

JUnit 5 用户指南

2018-01-01

袁慎建 · 王亚鑫 · 何疆乐 @ ThoughtWorks - Version 5.0.2

- [1. 概述](#)
 - [1.1. JUnit 5 是什么?](#)
 - [1.2. 支持的Java版本](#)
 - [1.3. 获取帮助](#)
- [2. 安装](#)
 - [2.1. 依赖元数据](#)
 - [2.2. 依赖关系图](#)
 - [2.3 JUnit Jupiter示例工程](#)
- [3. 编写测试](#)
 - [3.1. 注解](#)
 - [3.2. 标准测试类](#)
 - [3.3. 显示名称](#)
 - [3.4. 断言](#)
 - [3.5. 假设](#)
 - [3.6. 禁用测试](#)
 - [3.7. 标记和过滤](#)
 - [3.8. 测试实例生命周期](#)
 - [3.9. 嵌套测试](#)
 - [3.10. 构造函数和方法的依赖注入](#)
 - [3.11. 测试接口和默认方法](#)
 - [3.12. 重复测试](#)
 - [3.13. 参数化测试](#)
 - [3.14. 测试模板](#)
 - [3.15. 动态测试](#)
- [4. 运行测试](#)
 - [4.1. IDE支持](#)
 - [4.2. 构建工具支持](#)
 - [4.3. 控制台启动器](#)
 - [4.4. 使用JUnit 4运行JUnit Platform](#)

-
- [4.5. 配置参数](#)
 - [5. 扩展模型](#)
 - [5.1. 概述](#)
 - [5.2. 注册扩展](#)
 - [5.3. 条件测试执行](#)
 - [5.4. 测试实例后处理](#)
 - [5.5. 参数解析](#)
 - [5.6. 测试生命周期回调](#)
 - [5.7. 异常处理](#)
 - [5.8. 为测试模板提供调用上下文](#)
 - [5.9. 在扩展中保持状态](#)
 - [5.10. 在扩展中支持的实用程序](#)
 - [5.11. 用户代码和扩展的相对执行顺序](#)
 - [6. 从JUnit4迁移](#)
 - [6.1. 在 JUnit Platform 上运行JUnit4 测试](#)
 - [6.2. 迁移技巧](#)
 - [6.3. 对JUnit4规则的有限支持](#)
 - [7. 高级主题](#)
 - [7.1 JUnit Platform启动器API](#)
 - [8. API演变](#)
 - [8.1. API 版本和状态](#)
 - [8.2. 试验性API](#)
 - [8.3. @API工具支持](#)
 - [9. 贡献者](#)
 - [10. 发布记录](#)
 - [5.0.2](#)
 - [5.0.1](#)
 - [5.0.0](#)
 - [5.0.0-RC3](#)
 - [5.0.0-RC2](#)
 - [5.0.0-RC1](#)
 - [5.0.0-M6](#)
 - [5.0.0-M5](#)
 - [5.0.0-M4](#)
 - [5.0.0-M3](#)
 - [5.0.0-M2](#)
 - [5.0.0-M1](#)
 - [5.0.0-ALPHA](#)

1. 概述

本文档的目标是为那些编写测试的程序员、扩展开发人员（extension authors）和引擎开发人员（engine authors）以及构建工具和IDE供应商提供综合全面的参考。

本文档中文PDF文档 [即将发布](#)。

1.1. JUnit 5 是什么？

JUnit 5跟以前的JUnit版本不一样，它由几大不同的模块组成，这些模块分别来自三个不同的子项目。

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可用于开发在平台上运行的测试框架。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，它是由在JUnit 5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](#) 上跟我们交流。

2. 安装

最终版本和里程碑版本已经被部署到Maven仓库中心。

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

JUnit 内部通用类库/实用工具，它们仅用于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

支持从控制台中发现和执行JUnit Platform上的测试。详情请参阅 [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。详情请参阅 [控制台启动器](#)。

junit-platform-engine

测试引擎的公共API。详情请参阅 [插入你自己的测试引擎](#)。

junit-platform-gradle-plugin

支持使用 [Gradle](#) 来发现和执行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。详情请参阅 [JUnit Platform启动器API](#)。

junit-platform-runner

在一个JUnit 4环境中的JUnit Platform上执行测试和测试套件的运行器。详情请参阅 [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](#) 来发现和执行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 [参数化测试](#)。

junit-jupiter-migration-support

支持从JUnit 4迁移到JUnit Jupiter，仅在使用了JUnit 4规则的测试中才需要。

2.1.3. JUnit Vintage

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

junit-vintage-engine

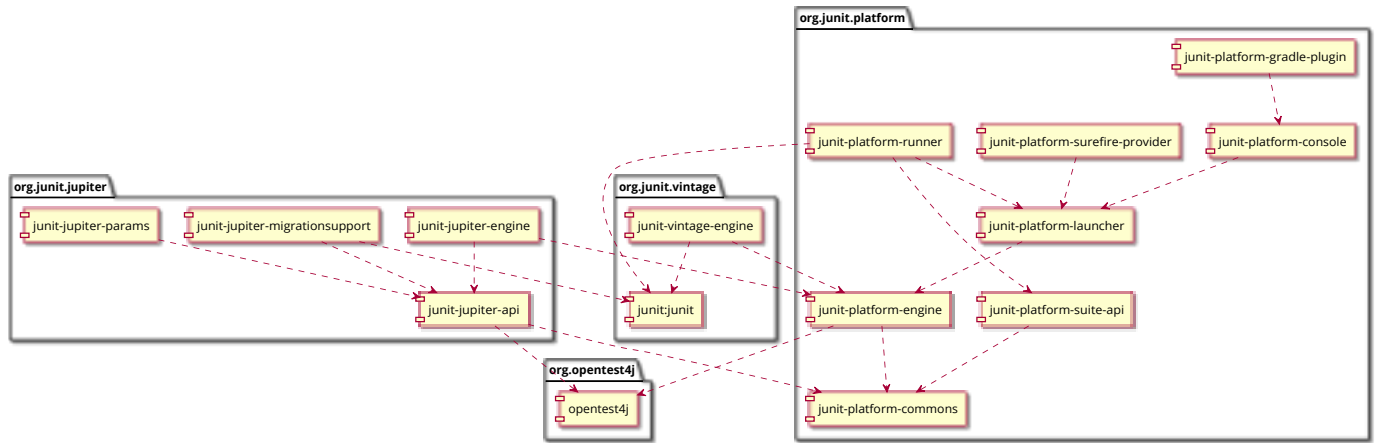
JUnit Vintage测试引擎实现，允许在新的JUnit Platform上运行低版本的JUnit测试，即那些以JUnit 3或JUnit 4风格编写的测试。

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它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。 |
| @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。 |
| @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。 |
| @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。 |
| @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 测试模板 ，它会依据注册的 提供者 所返回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。 |
| @DisplayName | 为测试类或测试方法声明一个自定义的显示名称。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。 |
| @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。 |
| @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” 测试实例生命周期 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” 测试实例生命周期 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” 测试实例生命周期 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TestNG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试验阶段。详细信息请参阅 [试验性API](#) 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("J°□° J")
    void testWithDisplayNameContainingSpecialCharacters() {
    }

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

}
```

3.4. 断言

JUnit Jupiter附帶了很多JUnit 4就已经存在的断言方法，并增加了一些适合与Java8 Lambda一起使用的断言。所有的JUnit Jupiter断言都是 [org.junit.jupiter.Assertions](https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html) 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许多测试场景，但有时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。因此，开发人员可以自由使用他们选择的断言类库。

举个例子，**匹配器**和流式调用的API组合起来使用可以让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中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。那些标记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的default方法上声明`@BeforeAll`和`@AfterAll`。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上没有使用`@TestInstance`注解，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 设置为定义在`TestInstance.Lifecycle`中的枚举常量名称即可，名称忽略大小写。它也作为一个JVM系统属性、作为一个传递给Launcher的`LauncherDiscoveryRequest`中的配置参数、或通过JUnit Platform配置文件来提供（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配置文件来设置默认的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`，你需要在类路径的根目录（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
```

 如果没有做到应用一致的配置，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。例如，如果构建将“per-class”语义配置为默认值，但是IDE中的测试却使用“per-method”的语义来执行，这样会增加在构建服务器上调试错误的难度。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

嵌套测试让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());
        }
    }
}
```

 `@Nested` 测试类必须是 **非静态嵌套类**（即内部类），并且可以有任意多层的嵌套。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置解析器。

- [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Tag。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，以便实现了该接口的类自动继承它的tags和扩展。请参阅 [测试执行之前和之后的回调](#) 章节的 [TimingExtension](#) 源代码。

```

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

```

在测试类中，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场景是为接口契约编写测试。例如，你可以编写测试，以了解Object.equals或Comparable.compareTo的工作原理。

```
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();

    @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。

```
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，我们可以使用`@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就像执行常规的`@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一个会自动重复执行10次的测试方法`repeatedTest()`。

```
@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，显示名称可以是由静态文本和动态占位符的组合而组成的模式。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。

一个特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. 重复测试示例

本节末尾的`RepeatedTestsDemo`类将演示几个重复测试的示例。

`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，我们就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
```

```
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 {totalRepetition s}")
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参数源来给每次调用提供参数。

 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 ar
gument) {
    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，参数源的值不会被生命周期方法（例如@BeforeEach）和测试类构造函数解析。

```
@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 Jupiter的 [注解](#) 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。这种新的测试是一个动态测试，它们由一个使用了`@TestFactory`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`或`Iterator`。`DynamicNode`的两个可实例化子类是`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;
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

4.1. IDE支持

4.1.1. IntelliJ IDEA

IntelliJ IDEA 从 2016.2 版本开始支持在JUnit Platform上运行测试。详情请参阅 [IntelliJ IDEA的相关博客](#)。

表格1. Junit5 版本对应的 IntelliJ IDEA

| IntelliJ IDEA 版本 | 捆绑的 JUnit 5 版本 |
|------------------|----------------|
| 2016.2 | M2 |
| 2016.3.1 | M3 |
| 2017.1.2 | M4 |
| 2017.2.1 | M5 |
| 2017.2.3 | RC2 |

⚠️ IntelliJ IDEA 与 JUnit5 的特定版本绑定，也就是说，如果你使用了Jupiter API更新的里程碑版本，执行测试时可能不起作用。这种情况一致持续到JUnit 5第一个GA版本发布才得到改善。在这之前，你可以在IntelliJ IDEA中按照下面所示的方法使用JUnit 5的新版本。

要想使用JUnit 5的不同版本，你需要在classpath中手动添加junit-platform-launcher、junit-jupiter-engine和junit-vintage-engine的JAR文件。

添加Gradle依赖

```
// Only needed to run tests in an IntelliJ IDEA that bundles an older version
testRuntime("org.junit.platform:junit-platform-launcher:1.0.2")
testRuntime("org.junit.jupiter:junit-jupiter-engine:5.0.2")
testRuntime("org.junit.vintage:junit-vintage-engine:4.12.2")
```

添加Maven依赖

```
!-- Only required to run tests in an IntelliJ IDEA that bundles an older version -->
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-launcher</artifactId>
  <version>1.0.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.0.2</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.junit.vintage</groupId>
  <artifactId>junit-vintage-engine</artifactId>
  <version>4.12.2</version>
  <scope>test</scope>
</dependency>
```

4.1.2. Eclipse 测试版支持

Eclipse 4.7（Oxygen）的测试版支持JUnit Platform和JUnit Jupiter。关于如何配置，请参阅 [Eclipse JDT UI/JUnit 5](#) wiki页面。

4.1.3. 其他 IDE

在本文写作之时，并没有其他任何IDE可以像IntelliJ IDEA或Eclipse的测试版一样可以直接在JUnit Platform上运行Java测试。但是，JUnit团队提供了另外两种折中的方法让JUnit 5可以在其他的IDE上使用。你可以尝试手动使用 [控制台启动器](#) 或者通过 [基于JUnit 4的Runner](#) 来执行测试。

4.2. 构建工具支持

4.2.1. Gradle

JUnit开发团队已经开发了一款非常基础的Gradle插件，它允许你运行被TestEngine（例如，JUnit3、JUnit4、JUnit Jupiter以及 [Specsy](#) 等）支持的任何种类的测试。关于插件的使用示例请参阅 [junit5-gradle-consumer](#) 项目中的build.gradle文件。

启用JUnit Gradle插件

要使用JUnit Gradle插件，你首先要确保使用了Gradle 2.5或更高的版本，然后你可以按照下面的模板来配置项目中的build.gradle文件。

```
buildscript {
    repositories {
        mavenCentral()
        // The following is only necessary if you want to use SNAPSHOT releases.
        // maven { url 'https://oss.sonatype.org/content/repositories/snapshots' }
    }
    dependencies {
        classpath 'org.junit.platform:junit-platform-gradle-plugin:1.0.2'
    }
}

apply plugin: 'org.junit.platform.gradle.plugin'
```

配置JUnit Gradle插件

一旦应用了JUnit Gradle插件，你就可以按照下面的方式进行配置。

```
junitPlatform {
    platformVersion '1.0.2' // optional, defaults to plugin version
    logManager 'org.apache.logging.log4j.jul.LogManager'
```

```
reportsDir file('build/test-results/junit-platform') // this is the default
// enableStandardTestTask true
// selectors (optional)
// filters (optional)
}
```

设置logManager会让JUnit Gradle插件将java.util.logging.manager系统属性设置为当前所提供的java.util.logging.LogManager实现类的全类名。上面的示例演示了如何将log4j配置为LogManager。

JUnit Gradle插件在默认情况下会禁用标准的Gradle test任务，但可以通过enableStandardTestTask标志来启用。

配置选择器

默认情况下，插件会扫描项目中所有测试的输出目录。不过，你可以使用一个叫selectors的扩展元素来显式指定要执行哪些测试。

```
junitPlatform {
    // ...
    selectors {
        uris 'file:///foo.txt', 'http://example.com/'
        uri 'foo:resource' ①
        files 'foo.txt', 'bar.csv'
        file 'qux.json' ②
        directories 'foo/bar', 'bar/qux'
        directory 'qux/bar' ③
        packages 'com.acme.foo', 'com.acme.bar'
        aPackage 'com.example.app' ④
        classes 'com.acme.Foo', 'com.acme.Bar'
        aClass 'com.example.app.Application' ⑤
        methods 'com.acme.Foo#a', 'com.acme.Foo#b'
        method 'com.example.app.Application#run(java.lang.String[])' ⑥
        resources '/bar.csv', '/foo/input.json'
        resource '/com/acme/my.properties' ⑦
    }
    // ...
}
```

- ① URIs
- ② 本地文件
- ③ 本地目录
- ④ 包
- ⑤ 类，全类名
- ⑥ 方法，全方法名（请参阅 [DiscoverySelectors中的selectMethod\(String\)方法](#)）
- ⑦ 类路径资源

配置过滤器

你可以使用`filters`扩展来配置测试计划的过滤器。默认情况下，所有的引擎和标记都会被包含在测试计划中。但只有默认的`includeClassNamePattern(^.*Tests?$)`会被应用。你可以重写默认的匹配模式，例如下面示例。当你使用了多种匹配模式时，JUnit Platform会使用逻辑 或 将它们合并起来使用。

```
junitPlatform {
    // ...
    filters {
        engines {
            include 'junit-jupiter'
            // exclude 'junit-vintage'
        }
        tags {
            include 'fast', 'smoke'
            // exclude 'slow', 'ci'
        }
        packages {
            include 'com.sample.included1', 'com.sample.included2'
            // exclude 'com.sample.excluded1', 'com.sample.excluded2'
        }
        includeClassNamePattern '.*Spec'
        includeClassNamePatterns '.*Test', '.*Tests'
    }
    // ...
}
```

如果你通过`engines {include ...}`或`engines {exclude ...}`来提供一个测试引擎ID，那么JUnit Gradle插件将只运行你希望运行的那个测试引擎。同样，如果你通过`tags {include ...}`或者`tags {exclude ...}`提供一个标记，JUnit Gradle插件将只运行相应标记的测试（例如，通过JUnit Jupiter测试的`@Tag`注解来过滤）。同理，关于包名，可以通过`packages {include ...}`或者`packages {exclude ...}`配置要包含或排除的包名。

配置参数

你可以使用`configurationParameter`或者`configurationParameters` DSL来设置配置参数，从而影响测试发现和执行。前者可以配置单独的配置参数，后者可以使用一个配置参数的map来一次性配置多个键-值对。所有的key和value都必须是String类型。

```
junitPlatform {  
    // ...  
    configurationParameter 'junit.jupiter.conditions.deactivate', '*'  
    configurationParameters([  
        'junit.jupiter.extensions.autodetection.enabled': 'true',  
        'junit.jupiter.testinstance.lifecycle.default': 'per_class'  
    ])  
    // ...  
}
```

配置测试引擎

为了让JUnit Gradle插件运行所有测试，类路径中必须存在一个TestEngine的实现。

要支持基于JUnit Jupiter的测试，你需要配置一个JUnit Jupiter API的 `testCompile`依赖以及JUnit Jupiter TestEngine实现的`testRuntime`依赖，具体配置如下。

```
dependencies {  
    testCompile("org.junit.jupiter:junit-jupiter-api:5.0.2")  
    testRuntime("org.junit.jupiter:junit-jupiter-engine:5.0.2")  
}
```

只要你配置了一个JUnit 4的`testCompile`依赖以及JUnit Vintage TestEngine实现的`testRuntime`依赖，JUnit Gradle插件就可以运行基于JUnit 4的测试，具体配置如下。

```
dependencies {  
    testCompile("junit:junit:4.12")  
    testRuntime("org.junit.vintage:junit-vintage-engine:4.12.2")  
}
```

使用JUnit Gradle插件

一旦应用并配置了JUnit Gradle插件，你就可以使用新的junitPlatformTest任务（在可用的Gralde task中会多出一个名为junitPlatformTest的Task）。

在命令行中调用gradlew junitPlatformTest（或者gradlew test）指令，项目中所有满足当前includeClassNamePattern（默认匹配`^.*Tests?$`）配置的测试会被执行。

在 [junit5-gradle-consumer](#) 项目中执行 junitPlatformTest任务会看到类似下面的输出。

```
:junitPlatformTest
```

```
Test run finished after 93 ms
```

```
[      3 containers found      ]
[      0 containers skipped    ]
[      3 containers started    ]
[      0 containers aborted    ]
[      3 containers successful ]
[      0 containers failed     ]
[      3 tests found          ]
[      1 tests skipped         ]
[      2 tests started         ]
[      0 tests aborted         ]
[      2 tests successful      ]
[      0 tests failed          ]
```

```
BUILD SUCCESSFUL
```

如果测试失败，build会失败，并且会输出类似下面的信息。

```
:junitPlatformTest
```

```
Test failures (1):
```

```
  JUnit Jupiter:SecondTest:mySecondTest()
```

```
    MethodSource [className = 'com.example.project.SecondTest', methodName = 'mySecondTest',
methodParameterTypes = '']
```

```
    => Exception: 2 is not equal to 1 ==> expected: <2> but was: <1>
```

```
Test run finished after 99 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         ]
[      2 tests successful      ]
[      1 tests failed          ]

:junitPlatformTest FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':junitPlatformTest'.
> Process 'command' '/Library/Java/JavaVirtualMachines/jdk1.8.0_92.jdk/Contents/Home/bin/java'
   finished with non-zero exit value 1

```

 当任何一个容器或测试失败时，退出值为1；否则，退出值为0。

当前JUnit Gradle插件的限制

任何通过JUnit Gradle插件运行的测试结果都不会包含在Gradle生成的标准测试报告中；但通常可以在持续集成服务器上汇总测试结果。详情请参阅插件的`reportsDir`属性。

4.2.2. Maven

JUnit团队已经为Maven Surefire开发了一个非常基础的provider，它允许你使用`mvn test`运行JUnit 4和JUnit Jupiter测试。[junit5-maven-consumer](#) 项目中的`pom.xml`文件演示了如何使用它，你可以以它作为一个起点。

 由于Surefire2.20存在内存泄漏的漏洞，`junit-platform-surefire-provider`目前仅适用于Surefire 2.19.1。

```

...
<build>
  <plugins>
    ...
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.19</version>

```

```
        <dependencies>
            <dependency>
                <groupId>org.junit.platform</groupId>
                <artifactId>junit-platform-surefire-provider</artifactId>
                <version>1.0.2</version>
            </dependency>
        </dependencies>
    </plugin>
</plugins>
</build>
...

```

配置测试引擎

为了让Maven Surefire运行所有测试，必须将TestEngine实现添加到运行时类路径中。

要支持基于JUnit Jupiter的测试，你需要配置一个JUnit Jupiter API的test依赖，并将JUnit Jupiter TestEngine的实现添加到maven-surefire-plugin的依赖项中，如下所示。

```
...
<build>
    <plugins>
        ...
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.19</version>
            <dependencies>
                <dependency>
                    <groupId>org.junit.platform</groupId>
                    <artifactId>junit-platform-surefire-provider</artifactId>
                    <version>1.0.2</version>
                </dependency>
                <dependency>
                    <groupId>org.junit.jupiter</groupId>
                    <artifactId>junit-jupiter-engine</artifactId>
                    <version>5.0.2</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</build>

```

```
...
<dependencies>
  ...
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.0.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
...
```

只要你配置了JUnit 4的test依赖，并将JUnit Vintage TestEngine的实现添加到maven-surefire-plugin的依赖项中，JUnit Platform Surefire Provider 就可以运行基于JUnit 4的测试。具体配置如下。

```
...
<build>
  <plugins>
    ...
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.19</version>
      <dependencies>
        <dependency>
          <groupId>org.junit.platform</groupId>
          <artifactId>junit-platform-surefire-provider</artifactId>
          <version>1.0.2</version>
        </dependency>
        ...
        <dependency>
          <groupId>org.junit.vintage</groupId>
          <artifactId>junit-vintage-engine</artifactId>
          <version>4.12.2</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
</build>
...
<dependencies>
  ...
```

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
</dependencies>
...
```

按Tag过滤

使用以下配置属性，你可以通过Tag来过滤测试。

- 要包含一个 tag，可以使用groups或者includeTags
- 要排除一个 tag，可以使用excludedGroups或者excludeTags

```
...
<build>
  <plugins>
    ...
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.19</version>
      <configuration>
        <properties>
          <includeTags>acceptance</includeTags>
          <excludeTags>integration, regression</excludeTags>
        </properties>
      </configuration>
      <dependencies>
        ...
      </dependencies>
    </plugin>
  </plugins>
</build>
...
```

配置参数

你可以使用`configurationParameters`属性并以Java Properties文件的语法提供键值对来设置配置参数，从而影响测试发现和执行。

```
...
<build>
  <plugins>
    ...
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.19</version>
      <configuration>
        <properties>
          <configurationParameters>
            junit.jupiter.conditions.deactivate = *
            junit.jupiter.extensions.autodetection.enabled = true
            junit.jupiter.testinstance.lifecycle.default = per_class
          </configurationParameters>
        </properties>
      </configuration>
      <dependencies>
        ...
      </dependencies>
    </plugin>
  </plugins>
</build>
...
```

4.3. 控制台启动器

[ConsoleLauncher](#) 是一个Java的命令行应用程序，它允许你通过命令行来启动JUnit Platform。例如，它可以用来运行JUnit Vintage和JUnit Jupiter测试，并在控制台中打印测试结果。

`junit-platform-console-standalone-1.0.2.jar`这个包含了所有依赖的可执行的jar包已经被发布在Maven仓库中，它位于 [junit-platform-console-standalone](#) 目录下，你可以 [运行](#) 独立的ConsoleLauncher，如下所示。

```
java -jar junit-platform-console-standalone-1.0.2.jar<Options>
```

如下所示为一个输出的例子。

```
|─ JUnit Vintage
|  └─ example.JUnit4Tests
|     └─ standardJUnit4Test ✓
```

```

└─ JUnit Jupiter
  └─ StandardTests
    │ └─ succeedingTest() ✓
    │ └─ skippedTest() ~ for demonstration purposes
  └─ A special test case
    │ └─ Custom test name containing spaces ✓
    │ └─ J°□°) J ✓
    └─ 🤖 ✓

```

Test run finished after 64 ms

```

[      5 containers found      ]
[      0 containers skipped    ]
[      5 containers started    ]
[      0 containers aborted    ]
[      5 containers successful ]
[      0 containers failed     ]
[      6 tests found          ]
[      1 tests skipped         ]
[      5 tests started         ]
[      0 tests aborted         ]
[      5 tests successful      ]
[      0 tests failed          ]

```

📖 退出码

如果任何容器或测试失败, [ConsoleLauncher](#) 就会以状态码1退出, 否则退出码为0.

4.3.1. Options

| Option | Description |
|---|---|
| ----- | ----- |
| <code>-h, --help</code> | Display help information. |
| <code>--disable-ansi-colors</code> | Disable ANSI colors in output (not supported by all terminals). |
| <code>--details <[none,flat,tree,verbose]></code> | Select an output details mode for when tests are executed. Use one of: [none, flat, tree, verbose]. If 'none' is selected, then only the summary and test failures are shown. (default: tree) |
| <code>--details-theme <[ascii,unicode]></code> | Select an output details tree theme for when tests are executed. Use one of: |

| | |
|--|---|
| <code>--class-path, --classpath, --cp <Path: path1:path2:...></code> | [ascii, unicode] (default: unicode) Provide additional classpath entries <code>--for</code> example, <code>for</code> adding engines and their dependencies. This option can be repeated. |
| <code>--reports-dir <Path></code> | Enable report output into a specified <code>local</code> directory (will be created <code>if</code> it does not exist). |
| <code>--scan-class-path, --scan-classpath [Path: path1:path2:...]</code> | Scan all directories on the classpath or explicit classpath roots. Without arguments, only directories on the system classpath as well as additional classpath entries supplied via <code>-cp</code> (directories and JAR files) are scanned. Explicit classpath roots that are not on the classpath will be silently ignored. This option can be repeated. |
| <code>-u, --select-uri <URI></code> | Select a URI <code>for test</code> discovery. This option can be repeated. |
| <code>-f, --select-file <String></code> | Select a file <code>for test</code> discovery. This option can be repeated. |
| <code>-d, --select-directory <String></code> | Select a directory <code>for test</code> discovery. This option can be repeated. |
| <code>-p, --select-package <String></code> | Select a package <code>for test</code> discovery. This option can be repeated. |
| <code>-c, --select-class <String></code> | Select a class <code>for test</code> discovery. This option can be repeated. |
| <code>-m, --select-method <String></code> | Select a method <code>for test</code> discovery. This option can be repeated. |
| <code>-r, --select-resource <String></code> | Select a classpath resource <code>for test</code> discovery. This option can be repeated. |
| <code>-n, --include-classname <String></code> | Provide a regular expression to include only classes whose fully qualified names match. To avoid loading classes unnecessarily, the default pattern only includes class names that end with "Test" or "Tests". When this option is repeated, all patterns will be combined using OR semantics. (default: <code>^.*Tests?\$</code>) |
| <code>-N, --exclude-classname <String></code> | Provide a regular expression to exclude those classes whose fully qualified |

```
--include-package <String>
```

```
--exclude-package <String>
```

```
-t, --include-tag <String>
```

```
-T, --exclude-tag <String>
```

```
-e, --include-engine <String>
```

```
-E, --exclude-engine <String>
```

```
--config <key=value>
```

names match. When this option is repeated, all patterns will be combined using OR semantics.

Provide a package to be included **in** the **test** run. This option can be repeated.

Provide a package to be excluded from the **test** run. This option can be repeated.

Provide a tag to be included **in** the **test** run. This option can be repeated.

Provide a tag to be excluded from the **test** run. This option can be repeated.

Provide the ID of an engine to be included **in** the **test** run. This option can be repeated.


Provide the ID of an engine to be excluded from the **test** run. This option can be repeated.

Set a configuration parameter **for test** discovery and execution. This option can be repeated.

4.4. 使用JUnit 4运行JUnit Platform

JUnitPlatform 运行器是一个基于JUnit 4的Runner，它让你能够在一个JUnit 4环境中的JUnit Platform上运行那些编程模型被支持的任何测试。例如一个JUnit Jupiter测试类。

如果某个类被标注了@RunWith(JUnitPlatform.class)注解，它就可以在那些支持JUnit 4但是还不支持JUnit Platform的IDE和构建系统中直接运行。

 由于JUnit Platform具备一些JUnit 4不具备的功能，因此运行器只能部分支持JUnit Platform的功能，特别是在报告方面（请参阅 [显示名称与技术名称](#)）。但是就目前来说，JUnitPlatform运行器是一个简单的入门方式。

4.4.1. 设置

你需要在类路径中添加以下的组件和它们的依赖。可以在 [依赖元数据](#) 中查看关于group ID, artifact ID 和版本的详细信息。

显式依赖

- junit-4.12.jar 在**test**作用域内：使用JUnit 4运行测试。

- `junit-platform-runner` 在 *test* 作用域内：JUnitPlatform运行器的位置。
- `junit-jupiter-api` 在 *test* 作用域内：编写测试的API，包括 `@Test` 等。
- `junit-jupiter-engine` 在 *test runtime* 范围内：JUnit Jupiter引擎API的实现。

可传递的依赖

- `junit-platform-launcher` 在 *test* 作用域内
- `junit-platform-engine` 在 *test* 作用域内
- `junit-platform-commons` 在 *test* 作用域内
- `opentest4j` 在 *test* 作用域内

4.4.2. 展示名称与技术名称

默认情况下，*显示名称* 会被使用在测试产出物上，但是当JUnitPlatform运行器使用Gradle或者Maven等构建工具来运行测试时，生成的测试报告通常需要包含测试产出物的*技术名称*（例如，使用完整类名），而不是像测试类的简单名称或包含特殊字符的自定义显示名称这种较短的显示名称。为了在测试报告中使用的技术名称，在 `@RunWith(JUnitPlatform.class)` 注解旁声明 `@UseTechnicalNames` 注解即可。

4.4.3. 单一测试类

使用JUnitPlatform运行器的方式之一是直接在测试类上添加 `@RunWith(JUnitPlatform.class)` 注解。请注意，以下示例中的测试方法使用的注解是 `org.junit.jupiter.api.Test`（JUnit Jupiter），而不是 `org.junit.Test`（JUnit Vintage）。同时，这个类中的测试用例必须为 `public`，否则，IDE不能将其识别为一个JUnit 4的测试类。

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

import org.junit.jupiter.api.Test;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
public class JUnit4ClassDemo {

    @Test
    void succeedingTest() {
        /* no-op */
    }
}
```

```
@Test
void failingTest() {
    fail("Failing for failing's sake.");
}

}
```

4.4.4. 测试套件

如果你有多个测试类，你可以创建一个测试套件，如下例子所示。

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectPackages;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectPackages("example")
public class JUnit4SuiteDemo {
}
```

JUnit4SuiteDemo类会发现并运行所有位于example包及其子包下的测试。默认情况下，它只包含类名符合正则表达式`^.*Tests?$`的测试类。

附加配置选项

除了@SelectPackages之外，还有很多配置选项可以用来发现和过滤测试。详细内容请参考 [Javadoc](#)。

4.5. 配置参数


除了告诉平台要包含哪些测试类、测试引擎以及要扫描哪些包等之外，有时还需要提供额外的自定义配置参数，该参数特定于特定的测试引擎。例如，JUnit Jupiter TestEngine支持以下用例中的配置参数。

- [更改默认的测试实例生命周期](#)
- [启用自动扩展检测](#)
- [禁用条件](#)

配置参数是一种基于文本的键值对，可以通过以下任何一种机制将其提供给运行在JUnit Platform上的测试引擎。

1. LauncherDiscoveryRequestBuilder中的configurationParameter()和configurationParameters()方法可以用来构建提供给 [Launcher API](#) 的请求。当使用JUnit Platform提供的某一种工具运行测试时，你可以采用如下所示的方式指定配置参数：

- [控制台启动器](#): 使用`--config`命令行选项。
 - [Gradle插件](#): 使用`configurationParameter`或者`configurationParametersDSL`。
 - [Maven Surefire 提供者](#): 使用 `configurationParameters` 属性。
2. JVM 系统属性。
 3. JUnit Platform配置文件: 该文件命名为`junit-platform.properties`, 位于类路径根目录下, 并遵循Java Properties文件的语法。

 配置参数会按照上面定义的顺序查找。所以, 直接提供给Launcher的配置参数优先于通过系统属性和配置文件提供的配置参数。同样, 通过系统属性提供的配置参数优先于通过配置文件提供的参数。

5. 扩展模型

5.1. 概述

不同于JUnit4中的Runner、@Rule以及@ClassRule等多个扩展点, JUnit Jupiter的扩展模型由一个连贯的概念组成: ExtensionAPI。但是, 需要注意的是 Extension本身也只是一个标记接口。

5.2. 注册扩展

JUnit Jupiter中的扩展可以通过 [@ExtendWith](#) 注解显式注册, 或者通过Java的 [ServiceLoader机制](#) 自动注册。

5.2.1. 声明式扩展注册

开发者可以通过在测试接口、测试类、测试方法或者自定义的 [组合注解](#) 上标注`@ExtendWith(...)`并提供要注册扩展类的引用, 从而以*声明式*的方式注册一个或多个扩展。

例如, 要给某个测试方法注册一个自定义的MockitoExtension, 你可以参照如下的方式标注该方法。

```
@ExtendWith(MockitoExtension.class)
@Test
void mockTest() {
    // ...
}
```

若要为某个类或者其子类注册一个自定义的MockitoExtension, 将注解添加到测试类上即可。

```
@ExtendWith(MockitoExtension.class)
class MockTests {
```

```
// ...  
}
```

多个扩展类的注册可以通过如下形式完成。

```
@ExtendWith({ FooExtension.class, BarExtension.class })  
class MyTestsV1 {  
    // ...  
}
```

还有另外一种方式来注册多个扩展类，如下面代码所示。

```
@ExtendWith(FooExtension.class)  
@ExtendWith(BarExtension.class)  
class MyTestsV2 {  
    // ...  
}
```

上述两种扩展注册方式是等价的，MyTestV1和MyTestV2都会被FooExtension和BarExtension进行扩展，且扩展顺序跟声明顺序一致。

5.2.2. 自动扩展注册

除了 [声明式扩展注册](#) 支持使用注解外，JUnit Jupiter同样也支持通过Java的java.util.ServiceLoader机制来做全局的扩展注册。采用这种机制后会自动的检测classpath下的第三方扩展，并自动完成注册。

具体来说，自定义扩展可以通过在org.junit.jupiter.api.extension.Extension文件中提供其全类名来完成注册，该文件位于其封闭的JAR文件中的/META-INF/services目录下。

启用自动扩展检测

自动检测是一种高级特性，默认情况下它是关闭的。要启用它，只需要在配置文件中将junit.jupiter.extensions.autodetection.enabled的配置参数 设置为 true即可。该参数可以作为JVM系统属性、或作为一个传递给Launcher的LauncherDiscoveryRequest中的配置参数、又或者通过JUnit Platform配置文件（详情请参阅 [配置参数](#)）来提供。

例如，要启用扩展的自动检测，你可以在启动JVM时传入如下系统参数。

```
-Djunit.jupiter.extensions.autodetection.enabled=true
```

启用自动检测功能后，通过ServiceLoader机制发现的扩展将在JUnit Jupiter的全局扩展（例如对TestInfo，TestReporter等的支持）之后被添加到扩展注册表中。

5.2.3. 扩展继承

扩展在测试类层次结构中以自顶向下的语义被继承。同样，在类级别注册的扩展会被方法级的扩展继承。此外，特定的扩展实现只能针对给定的扩展上下文及其父上下文进行一次注册。因此，任何尝试注册重复的扩展实现都将被忽略。

5.3. 条件测试执行

[ExecutionCondition](#) 定为程序化的条件测试执行定义了ExtensionAPI。

每个容器（例如测试类）都会对ExecutionCondition进行解析，从而确定是否应该根据提供的ExtensionContext执行其包含的所有测试。类似地，ExecutionCondition会被每个测试解析，从而确定是否应该根据提供的ExtensionContext执行给定的测试方法。

当多个ExecutionCondition扩展被注册时，只要有一个条件被禁用，容器或测试就会被禁用。所以，不能保证每个条件都会被解析，因为其中某个扩展可能已经导致容器或测试被禁用了。也就是说，条件的解析机制类似于短路 或 (符号为||)操作。

有关具体示例，请参阅 [DisabledCondition](#) 和 [@Disable](#) 的源码。

5.3.1. 禁用条件

有时候，在没有明确的条件被激活的情况下运行测试套件可能更有用。例如，你可能想要运行某些即便被标注了@Disable的测试，从而观察这些测试是否一直是失败的。此时只需为junit.jupiter.conditions.deactivate配置参数提供一个匹配模式，以指定当前测试运行应停用哪些条件（即不被解析）。该匹配模式可以作为JVM系统属性、或作为一个传递给Launcher的LauncherDiscoveryRequest中的配置参数、又或者通过JUnit Platform配置文件（详情请参阅 [配置参数](#)）来提供。

例如，要停用JUnit的@Disable条件，你可以在JVM启动时传入系统参数完成：

```
-Djunit.jupiter.conditions.deactivate=org.junit.*DisabledCondition
```

模式匹配语法

如果junit.jupiter.conditions.deactivate模式仅由星号（*）组成，则所有条件都将被禁用。否则，该模式将用于匹配每个注册的条件的完整的类名（FQCN）。模式中的点（.）会匹配FQCN中的点（.）或美元符号（\$）。星号（*）匹配FQCN中的一个或多个字符。模式中的所有其他字符将与FQCN一对一匹配。

例如：

- *: 禁用所有条件。
- org.junit.*: 禁用org.junit基础包及子包下的所有条件。
- *.MyCondition: 禁用MyCondition类中的每个条件。
- *System*: 禁用其简单类名包含System的类中的每个条件。
- org.example.MyCondition: 禁用FQCN为org.example.MyCondition的条件。

5.4. 测试实例后处理

[TestInstancePostProcessor](#) 为希望发布流程测试实例的Extensions定义了API。

常见的用法涵盖了诸如将依赖注入到测试实例中，在测试实例中调用自定义的初始化方法等。

关于具体示例，请查阅 [MockitoExtension](#) 和 [SpringExtension](#) 的源代码。

5.5. 参数解析

[ParameterResolver](#) 定义了用于在运行时动态解析参数的ExtensionAPI。

如果测试构造器或者@Test、@TestFactory、@BeforeEach、@AfterEach、@BeforeAll或者@AfterAll方法接收参数，则必须在运行时通过ParameterResolver解析该参数。开发人员可以使用内置的ParameterResolver（参考[TestInfoParameterResolver](#)）或[自己注册](#)。一般而言，参数可能被按照其名称、类型、注解或任何一种上述方式的组合所解析。具体示例可以参照[CustomTypeParameterResolver](#) 和 [CustomAnnotationParameterResolver](#) 的源码。

5.6. 测试生命周期回调

下列接口定义了用于在测试执行生命周期的不同阶段来扩展测试的API。关于每个接口的详细信息，可以参考后续章节的示例，也可以查阅 [org.junit.jupiter.api.extension](#) 包中的Javadoc。

- [BeforeAllCallback](#)
 - [BeforeEachCallback](#)
 - [BeforeTestExecutionCallback](#)
 - [AfterTestExecutionCallback](#)
 - [AfterEachCallback](#)
- [AfterAllCallback](#)

实现多个扩展API

扩展开发人员可以选择在单个扩展中实现任意数量的上述接口。具体示例请参阅 [SpringExtension](#) 的源代码。

5.6.1. 测试执行之前和之后的回调

[BeforeTestExecutionCallback](#) 和 [AfterTestExecutionCallback](#) 分别为Extensions定义了添加行为的API，这些行为将在执行测试方法之前和之后立即执行。因此，这些回调非常适合于定时器、跟踪器以及其他类似的场景。如果你需要实现围绕@BeforeEach和@AfterEach方法调用的回调，实现BeforeEachCallback和AfterEachCallback即可。

以下示例展示了如何使用这些回调来统计和记录测试方法的执行时间。TimingExtension同时实现了BeforeTestExecutionCallback和AfterTestExecutionCallback接口，从而给测试执行进行计时和记录。

一个为测试方法执行计时和记录的扩展

```
import java.lang.reflect.Method;
import java.util.logging.Logger;

import org.junit.jupiter.api.extension.AfterTestExecutionCallback;
import org.junit.jupiter.api.extension.BeforeTestExecutionCallback;
import org.junit.jupiter.api.extension.ExtensionContext;
import org.junit.jupiter.api.extension.ExtensionContext.Namespace;
import org.junit.jupiter.api.extension.ExtensionContext.Store;

public class TimingExtension implements BeforeTestExecutionCallback, AfterTestExecutionCallback {

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

    @Override
    public void beforeTestExecution(ExtensionContext context) throws Exception {
        getStore(context).put(context.getRequiredTestMethod(), System.currentTimeMillis());
    }

    @Override
    public void afterTestExecution(ExtensionContext context) throws Exception {
        Method testMethod = context.getRequiredTestMethod();
        long start = getStore(context).remove(testMethod, long.class);
        long duration = System.currentTimeMillis() - start;

        LOG.info(() -> String.format("Method [%s] took %s ms.", testMethod.getName(), duration));
    }

    private Store getStore(ExtensionContext context) {
```

```
        return context.getStore(Namespace.create(getClass(), context));
    }

}
```

由于`TimingExtensionTests`类通过`@ExtendWith`注册了`TimingExtension`，所以，测试将在执行时应用这个计时器。

一个使用示例`TimingExtension`的测试类

```
@ExtendWith(TimingExtension.class)
class TimingExtensionTests {

    @Test
    void sleep20ms() throws Exception {
        Thread.sleep(20);
    }

    @Test
    void sleep50ms() throws Exception {
        Thread.sleep(50);
    }

}
```

以下是运行`TimingExtensionTests`时生成的日志记录示例。

```
INFO: Method [sleep20ms] took 24 ms.
INFO: Method [sleep50ms] took 53 ms.
```

5.7. 异常处理

[TestExecutionExceptionHandler](#) 为`Extensions`定义了异常处理的API，从而可以处理在执行测试时抛出的异常。

下面的例子展示了一个扩展，它将吃掉所有的`IOException`，但会重新抛出任何其他类型的异常。

一个异常处理扩展

```
public class IgnoreIOExceptionExtension implements TestExecutionExceptionHandler {

    @Override
    public void handleTestExecutionException(ExtensionContext context, Throwable throwable)
        throws Throwable {

        if (throwable instanceof IOException) {
            return;
        }
        throw throwable;
    }
}
```

5.8. 为测试模板提供调用上下文

当至少有一个 [TestTemplateInvocationContextProvider](#) 被注册时，标注了 `@TestTemplate` 的方法才能被执行。每个这样的provider负责提供一个 [TestTemplateInvocationContext](#) 实例的Stream。每个上下文都可以指定一个自定义的显示名称和一个额外的扩展名列表，这些扩展名仅用于下一次调用 `@TestTemplate` 方法。

以下示例展示了如何编写测试模板以及如何注册和实现一个 [TestTemplateInvocationContextProvider](#)。

一个附带扩展名的测试模板

```
@TestTemplate
@ExtendWith(MyTestTemplateInvocationContextProvider.class)
void testTemplate(String parameter) {
    assertEquals(3, parameter.length());
}

static class MyTestTemplateInvocationContextProvider implements TestTemplateInvocationContext
Provider {

    @Override
    public boolean supportsTestTemplate(ExtensionContext context) {
        return true;
    }

    @Override
    public Stream<TestTemplateInvocationContext> provideTestTemplateInvocationContexts(ExtensionContext context) {
```

```
        return Stream.of(invocationContext("foo"), invocationContext("bar"));
    }

    private TestTemplateInvocationContext invocationContext(String parameter) {
        return new TestTemplateInvocationContext() {
            @Override
            public String getDisplayName(int invocationIndex) {
                return parameter;
            }

            @Override
            public List<Extension> getAdditionalExtensions() {
                return Collections.singletonList(new ParameterResolver() {
                    @Override
                    public boolean supportsParameter(ParameterContext parameterContext,
                                                    ExtensionContext extensionContext) {
                        return parameterContext.getParameter().getType().equals(String.class);
                    }

                    @Override
                    public Object resolveParameter(ParameterContext parameterContext,
                                                  ExtensionContext extensionContext) {
                        return parameter;
                    }
                });
            }
        };
    }
}
```

在这个例子中，测试模板将被调用两次。调用的显示名称是调用上下文指定的“foo”和“bar”。每个调用都会注册一个自定义的 [ParameterResolver](#) 用于解析方法参数。下面是使用ConsoleLauncher时产生的输出信息。

```
└─ testTemplate(String) ✓
   └─ foo ✓
      └─ bar ✓
```

[TestTemplateInvocationContextProvider](#) 扩展API主要用于实现不同类型的测试，这些测试依赖于某个类似于测试的方法的重复调用（尽管它们不在同一个上下文中）。例如，使用不同的参数，以不同的方式准备测试类实

例，或多次调用而不修改上下文。请参阅 [重复测试](#) 或 [参数化测试](#) 的实现，它们都使用了该扩展点来提供其相关的功能。

5.9. 在扩展中保持状态

通常，扩展只实例化一次。随之而来的相关问题是：开发者如何能够在两次调用之间保持扩展的状态？

`ExtensionContext` API 提供了一个 `Store` 用来解决这一问题。扩展可以将值放入 `Store` 中供以后检索。请参阅 [TimingExtension](#) 了解如何使用具有方法级作用域的 `Store`。要注意，在测试执行期间，被存储在一个 `ExtensionContext` 中的值在周围其他的 `ExtensionContext` 中是不可用的。由于 `ExtensionContexts` 可能是嵌套的，因此内部上下文的范围也可能受到限制。请参阅相应的 Javadoc 来了解有关通过 [Store](#) 存储和检索值的方法的详细信息。

5.10. 在扩展中支持的实用程序

JUnit Platform Commons 公开了一个名为 [org.junit.platform.commons.support](#) 的包，它包含了用于处理注解、反射和类路径扫描任务且正在维护中的实用工具方法。`TestEngine` 和 `Extension` 开发人员（authors）应该被鼓励去使用这些方法，以便与 JUnit Platform 的行为保持一致。

5.11. 用户代码和扩展的相对执行顺序

当执行包含一个或多个测试方法的测试类时，除了用户提供的测试和生命周期方法外，还会调用大量的回调函数。下图说明了用户提供的代码和扩展代码的相对顺序。

```

BeforeAllCallback (1)
  @BeforeAll (2)
    BeforeEachCallback (3)
      @BeforeEach (4)
        BeforeTestExecutionCallback (5)
          @Test (6)
            TestExecutionExceptionHandler (7)
              AfterTestExecutionCallback (8)
                @AfterEach (9)
                  AfterEachCallback (10)
                    @AfterAll (11)
                      AfterAllCallback (12)

```

Lifecycle Callbacks (`@ExtendWith(Extension)`)

User code: methods of the test class

用户代码和扩展代码

用户提供的测试和生命周期方法以橙色表示，扩展提供的回调代码由蓝色显示。灰色框表示单个测试方法的执行，并将在测试类中对每个测试方法重复执行。

下表进一步解释了 [用户代码和扩展代码](#) 图中的十二个步骤。

| 步骤 | 接口/注解 | 描述 |
|----|---|--------------------|
| 1 | 接口org.junit.jupiter.api.extension.BeforeAllCallback | 执行所有容器测试之前执行的扩展代码 |
| 2 | 注解org.junit.jupiter.api.BeforeAll | 执行所有容器测试之前执行的用户代码 |
| 3 | 接口org.junit.jupiter.api.extension.BeforeEachCallback | 每个测试执行之前执行的扩展代码 |
| 4 | 注解org.junit.jupiter.api.BeforeEach | 每个测试执行之前执行的用户代码 |
| 5 | 接口org.junit.jupiter.api.extension.BeforeTestExecutionCallback | 测试执行之前立即执行的扩展代码 |
| 6 | 注解org.junit.jupiter.api.Test | 真实测试方法的用户代码 |
| 7 | 接口org.junit.jupiter.api.extension.TestExecutionExceptionHandler | 用于处理测试期间抛出的异常的扩展代码 |
| 8 | 接口org.junit.jupiter.api.extension.AfterTestExecutionCallback | 测试执行后立即执行的扩展代码 |
| 9 | 注解org.junit.jupiter.api.AfterEach | 每个执行测试之后执行的用户代码 |
| 10 | 接口org.junit.jupiter.api.extension.AfterEachCallback | 每个执行测试之后执行的扩展代码 |
| 11 | 注解org.junit.jupiter.api.AfterAll | 执行所有容器测试之后执行的用户代码 |
| 12 | 接口org.junit.jupiter.api.extension.AfterAllCallback | 执行所有容器测试之后执行的扩展代码 |

在最简单的情况下，只有实际的测试方法被执行（步骤6）；所有其他步骤都是可选的，具体包含的步骤将取决于是否存在用户代码或对相应生命周期回调的扩展支持。有关各种生命周期回调的更多详细信息，请参阅每个注解和扩展各自的JavaDoc。

6. 从JUnit4迁移

虽然JUnit Jupiter编程模型和扩展模型本身不支持JUnit 4中的Rules和Runners等特性，但我们不期望源码维护者为了迁移到JUnit Jupiter，而必须更新其现有的所有测试、测试扩展以及自定义构建测试基础设施。

然而，JUnit通过一个JUnit Vintage测试引擎提供了一条平缓的迁移路径，该引擎允许使用JUnit Platform基础设施执行基于JUnit 3和JUnit 4的现有测试。由于所有JUnit Jupiter特有的类和注解都位于新的org.junit.jupiter基础包中，因此在类路径中同时使用JUnit 4和JUnit Jupiter不会导致任何冲突。所以，保持现有的JUnit 4测试和JUnit Jupiter测试是安全的。除此之外，JUnit团队会持续为JUnit 4.x 基线提供维护和错误修复的版本，所以开发人员有足够的时间按照自己的计划迁移到JUnit Jupiter。

6.1. 在 JUnit Platform 上运行JUnit4 测试

只要确保junit-vintage-engine包存在于你的测试运行时路径下，基于JUnit 3和 JUnit 4的测试将自动被JUnit Platform启动器拾取。

要想了解如何使用Gradle和Maven完成此操作，请参阅示例工程 [junit5-samples](#) 。

6.2. 迁移技巧

以下是在将现有JUnit 4测试迁移到JUnit Jupiter时必须注意的事项。

- `org.junit.jupiter.api`包中的注解。
- `org.junit.jupiter.api.Assertions`类中的断言。
- `org.junit.jupiter.api.Assumptions`类中的假设。
- `@Before`和`@After`已经不存在；取而代之的是`@BeforeEach`和`@AfterEach`。
- `@BeforeClass`和`@AfterClass`已经不存在；取而代之的是`@BeforeAll`和`@AfterAll`。
- `@Ignore` 已经不存在：取而代之的是 `@Disabled`。
- `@Category` 已经不存在：取而代之的是 `@Tag`。
- `@RunWith` 已经不存在：取而代之的是`@ExtendWith`。
- `@Rule`和 `@ClassRule`已经不存在；取而代之的是`@ExtendWith`；关于部分规则的支持请参阅后续章节。

6.3. 对JUnit4规则的有限支持

如前文所述，JUnit Jupiter本身不支持JUnit 4的Rule。然而，JUnit团队也意识到：很多组织，尤其是大型组织，很可能拥有使用自定义规则的大型JUnit 4代码库。为了给这些组织提供服务并实现平缓地迁移，JUnit团队决定在JUnit Jupiter中逐步地支持部分JUnit 4的Rule。这种支持是基于适配器的，并且仅限于那些与JUnit Jupiter扩展模型在语义上兼容的Rule，即那些不会完全改变测试总体执行流程的Rule。

JUnit Jupiter中的junit-jupiter-migrationsupport模块目前支持以下三种Rule类型以及它们的子类。

- `org.junit.rules.ExternalResource` (包含 `org.junit.rules.TemporaryFolder`)
- `org.junit.rules.Verifier` (包含`org.junit.rules.ErrorCollector`)
- `org.junit.rules.ExpectedException`

跟在JUnit 4中一样，规则注解的字段跟方法一样是被支持的。通过在测试类使用这些类级别的扩展，可以保留 遗留 代码库中的规则实现，其中包括JUnit4规则导入语句。

这种有限的Rule支持形式可以通过类级的注解

`org.junit.jupiter.migrationsupport.rules.EnableRuleMigrationSupport`来开启。该注解是一个组合注

解，它会启用所有支持迁移的扩展：VerifierSupport、ExternalResourceSupport和ExpectedExceptionSupport。

然而，如果你打算开发一个新的JUnit 5扩展，请使用JUnit Jupiter的新扩展模型，而不要再去使用JUnit 4中基于Rule的模型。

⚠ JUnit Jupiter中的JUnit 4 Rule支持目前是一个实验性功能。详细信息请参阅 [试验性API](#)

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与其编程客户端（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）在很大程度上解决过去IDE和构建工具难以识别测试类和测试方法的问题。

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，测试发现的搜索范围涵盖了类、方法、包中的所有类，甚至所有类路径中的测试。测试发现会贯穿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*，可以用它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

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`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

JUnit 5的主要目标之一是提高维护者发展演进JUnit的能力，尽管它在许多项目被使用。在JUnit 4中，起初作为内部构造而被添加的大量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。

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

每个已发布的artifact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 | 不应该再使用；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 |

| 包名 | 类名 | 类型 |
|--------------------------------------|-------------------------|----|
| 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pull request的完整列表，请参阅GitHub上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。

新特性与改进

- JUnit Platform Maven Surefire provider现在支持redirectTestOutputToFile Surefire功能。
- JUnit Platform Maven Surefire provider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Rule。现在，一旦启用，它将实例化所有声明的Rule（字段和方法），并按照JUnit 4使用的顺序来调用它们。
- 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。

新特性与改进

- 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pull request的完整列表，请参阅GitHub上JUnit仓库中的 [5.0.1](#) 里程碑页面。

整体改进

- 所有的artifact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 Rule`，`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 Rule`。

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pull request的完整列表，请参阅GitHub上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。

弃用和彻底改变

- `@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。
- `Tag`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。
- `FilePosition.getColumn()`现在返回`Optional<Integer>`而不是`int`。
- 以下所列的`TestSource`几个具体实现类的构造函数已被替换为命名`from(...)`的静态工厂方法。
 - `ClasspathResourceSource`
 - `ClassSource`
 - `CompositeTestSource`
 - `DirectorySource`
 - `FileSource`
 - `MethodSource`
 - `PackageSource`
- `LoggingListener`的构造函数已被替换为名为`forBiConsumer(...)`的静态工厂方法。
- `AbstractTestDescriptor`中的`getParent()`方法现在是`final`的。

新特性与改进

- `AbstractTestDescriptor`中的`children`字段现在是`protected`的，从而允许子类访问。

JUnit Jupiter

Bug修复

- `AbstractExtensionContext.getRoot()`现在会遍历完整的层次结构并返回真正的根上下文。

JUnit Vintage

没有变化。

5.0.0-RC3

发布时间： 2017.08.23

范围：配置参数和错误修复。

⚠ 这是一个预发行版，包含一些重大更改。如果想在捆绑了旧版里程碑版本的IntelliJ IDEA中使用此版本，请参阅上面的 [说明](#)。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上JUnit仓库中的 [5.0 RC3](#) 里程碑页面。

JUnit Platform

Bug修复

- 源JAR文件不再包含每个源文件两次。
- 如果一个失败的测试不是抛出一个`AssertionError`的实例，Maven Surefire provider现在会将其报告为错误而不是由于兼容性引起的失败，

新特性与改进

- 现在可以通过许多新的方式提供配置参数：
 - 通过类路径根目录下的`junit-platform.properties`文件。详情请参阅 [配置参数](#)。
 - 通过 [控制台启动器](#) 中的`--config`命令行选项。

- 通过Gradle插件的`configurationParameter`或`configurationParameters DSL`。
- 通过Maven Surefire provider的`configurationParameters`属性。

JUnit Jupiter

Bug修复

- 源JAR文件不再包含每个源文件两次。
- `ExecutionContext.Store.getOrComputeIfAbsent`现在在计算值之前会在其祖父级上下文中查找值（并在其父级中递归）。
- `ExecutionContext.Store.getOrComputeIfAbsent()`现在是线程安全的。
- 如果唯一ID属于不同的测试引擎，`JupiterTestEngine`就不会再尝试解析通过其中一个`DiscoverySelectors.selectUniqueId()`方法选择的唯一ID。

弃用和彻底改变

- 恢复RC1中引入的更改：现在使用与Java类相同的默认测试实例生命周期模式（即“per-method”）执行使用Kotlin编程语言编写的测试类。
- `junit.conditions.deactivate`配置参数已被重命名为 `junit.jupiter.conditions.deactivate`。
- `junit.extensions.autodetection.enabled`配置参数已被重命名为 `junit.jupiter.extensions.autodetection.enabled`。
- `ExtensionContext`中的默认全局扩展名称空间常量已从`Namespace.DEFAULT`重命名为`Namespace.GLOBAL`。
- 默认的`getStore()`方法已经从`ExtensionContext`接口中移除。要访问全局存储，需要显式调用`getStore(Namespace.GLOBAL)`方法。

新特性与改进

- 现在可以通过名为`junit.jupiter.testinstance.lifecycle.default`的配置参数或JVM系统属性来设置默认的测试实例生命周期模式。详情请参阅 [更改默认的测试实例生命周期](#)。
- 在参数化测试中使用`@CsvSource`或`@CsvFileSource`时，如果CSV解析器没有从输入中读取到任何字符，并且输入位于引号内，则返回空字符串""而不是`null`。

JUnit Vintage

Bug修复

- 源JAR文件不再包含每个源文件两次。

- 现在可以通过DiscoverySelectors中的selectMethod()变体在JUnit 4参数化测试类中选择单个方法。

5.0.0-RC2

发布时间： 2017.07.30

范围：修复junit-jupiter-engine的Gradle消耗

⚠️ 这是一个预发行版，包含一些重大更改。如果想在捆绑了旧版里程碑版本的IntelliJ IDEA中使用此版本，请参阅上面的 [说明](#)。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上JUnit仓库中的 [5.0 RC2](#)里程碑页面。

JUnit Platform

没有变化。

JUnit Jupiter

Bug修复

- 修正junit-jupiter-engine的无效POM，排除test作用域依赖。

JUnit Vintage

没有变化。

5.0.0-RC1

发布时间： 2017.07.30

范围：5.0 GA之前的错误修复和文档改进

⚠️ 这是一个预发行版，包含一些重大更改。如果想在捆绑了旧版里程碑版本的IntelliJ IDEA中使用此版本，请参阅上面的 [说明](#)。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上JUnit仓库中的 [5.0 RC1](#) 里程碑页面。

JUnit Platform

Bug修复

- 现在可以通过类，方法名，参数类型或完全限定的方法名来选择 未被实现类覆盖的通用接口default方法。这适用于DiscoverySelectors中的方法选择器以及ReflectionSupport中的findMethod()变体。
- 在使用ReflectionSupport中的findMethods()搜索类层次结构中的方法时，现在可以正确发现一个非重写的接口default方法，其方法签名被本地声明的方法重载。
- 在使用ReflectionSupport中的findMethods()搜索类层次结构中的方法时，不再发现重写的接口default方法。

弃用和彻底改变

- 从Launcher类中删除了已弃用的方法execute(LauncherDiscoveryRequest)。已删除的方法在里程碑4中被execute(LauncherDiscoveryRequest, TestExecutionListener ...)方法替换。

JUnit Jupiter

Bug修复

- 与@BeforeAll、@AfterAll、@BeforeEach或@AfterEach注解标注的生命周期方法有关的配置错误在测试发现阶段不再停止执行整个测试计划。相反，现在在执行受影响的测试类时就会报告这样的错误。
- 测试计划中已经正确地包含了一个未覆盖的接口默认方法，其方法签名被本地声明的方法重载。这适用于使用了Jupiter注解（如@Test，@BeforeEach等）的default方法。
- 测试计划中不再包含重写的接口default方法。这适用于使用了Jupiter注解（如@Test，@BeforeEach等）的default方法。

新特性与改进

- Assertions.assertThrows()接受自定义失败消息的新变体作为String或Supplier<String>。
- 如果测试类使用了@TestInstance(Lifecycle.PER_CLASS)，则@MethodSource引用的方法不再必须是static的。
- 使用Kotlin编程语言编写的测试类现在默认使用@TestInstance(Lifecycle.PER_CLASS)语义来执行。

JUnit Vintage

除了内部重构之外没有变化。

5.0.0-M6

发布时间： 2017.07.18

范围：JUnit 5的第六次发布，主要解决Java 9的兼容性、验证（例如：Tag语法规则）和修复bug。

⚠️ 这是一次里程碑式的发布，包含重大更改。如果想在捆绑了旧版里程碑版本的IntelliJ IDEA中使用此版本，请参阅上面的 [说明](#)。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上JUnit仓库中的 [5.0 M6](#) 里程碑页面。

兼容Java 9

JUnit 5的运行时环境的主要目标是Java 8，因此，JUnit 5的发布版本不能描述Java 9的编译模块。然而，由于 [第5次发布](#) 的每个发布包在其JAR清单中声明了稳定的Automatic-Module-Name，这使得可以在测试模块中包含著名的JUnit模块名称，如下所示。

```
module foo.bar {  
    requires org.junit.jupiter.api;  
}
```

通常在类路径上就可以运行测试，这方面Java 8和Java 9没什么区别。只要支持JUnit平台，大多数的命令行工具和IDE都可以*开箱即用* JUnit 5。如果你选择的开发工具不支持JUnit平台，可以使用ConsoleLauncher，甚至可以使用可执行的junit-platform-console-standalone一站式jar包。

要在模块路径中运行JUnit Jupiter测试，可以通过一个Java 9兼容的构建工具 [pro](#) 来实现。

pro 支持黑盒测试和白盒测试。前者用来测试模块表面，只能访问应用程序模块的导出位。后者使用合并的模块描述符技术，允许访问protected和包私有类型以及非导出包。

测试模块的例子可以查看 [pro的GitHub仓库](#)：integration.pro是一个黑盒测试模块；但是com.github.forax.pro.api和com.github.forax.pro.helper是白盒测试模块。

JUnit Platform

Bug修复

- 为了删除前导和尾随的空白，所有的Tag都被修剪了。这适用于任何直接通过TagFilter.includeTags()和TagFilter.excludeTags()或者间接通过@IncludeTags,@ExcludeTags，JUnit Platform控制台启动器，JUnit Platform的Gradle插件和JUnit平台的Maven Surefire provider所提供的任何Tag。

弃用和彻底改变

- 所有的Tag现在都要满足以下语法规则：
 - 标签不能是null或者空白。
 - 修剪的标签不能包含空白。
 - 修剪的标签不能包含ISO控制字符。
- 如果提供的Tag在语法上是无效的，`TagFilter.includeTags()`、`TagFilter.excludeTags()`和`TestTag.create()`工厂方法现在会抛出一个`PreconditionViolationException`异常。
- `EngineDiscoveryRequest`的`getDiscoveryFiltersByType`方法已经被改名为`getFiltersByType`。
- `UniqueId`的`getSegments()`方法现在返回一个不可变的列表。
- `AbstractTestDescriptor`的方法`setSource`被删除，替代它的是一个包含`source`变量的构造函数。

新特性与改进

- 为语法有效的Tag增加了一个新的检查方法`TestTag.isValid(String)`
- 如果应用的元素是类，`AnnotationSupport`的方法`findAnnotation()`现在可以被一个类实现的接口中搜索
- `org.junit.platform.commons.util.ReflectionUtils`的下面这些方法现在通过`org.junit.platform.commons.support.ReflectionSupport`向外暴露：
 - `public static Optional<Class<?>> loadClass(String name)`
 - `public static Optional<Method> findMethod(Class<?> clazz, String methodName, String parameterTypeNames)`
 - `public static Optional<Method> findMethod(Class<?> clazz, String methodName, Class<?>... parameterTypes)`
 - `public static <T> T newInstance(Class<T> clazz, Object... args)`
 - `public static Object invokeMethod(Method method, Object target, Object... args)`
 - `public static List<Class<?>> findNestedClasses(Class<?> clazz, Predicate<Class<?>> predicate)`

JUnit Jupiter

Bug修复

- 为了删除前导和尾随的空白，所有通过`@Tag`声明的标记都被修剪了。
- 现在支持所有基本数组类型（从`boolean[]`到`short[]`）作为参数化测试的静态参数的返回类型。

弃用和彻底改变

- `@Test`和有生命周期的方法现在都强制返回`void`类型。

新特性与改进

- 现在可以使用`Assertions` 中的所有`fail(...)`方法来实现单语句`lambda`表达式，从而避免需要实现具有显式返回值的代码块。
- `ExtensionContext`中的新方法`getRoot()`能够容易的获得最高的，`root` 扩展上下文。
- `ExtensionContext` API中的新方法`getRequiredTestClass()`、`getRequiredTestInstance()`和`getRequiredTestMethod()`更加便捷，可用于在需要这些元素的用例中检索测试类，测试实例和测试方法。
- 类似`@TestInstance`和`@Disabled`的类级注解现在可以被声明与测试接口上（也称为测试特征）。
- 如果针对单个方法解析了多个`TestDescriptor`，则现在会记录一条警告。这有助于调试由多个竞争注解（例如`@Test`，`@RepeatedTest`，`@ParameterizedTest`，`@TestFactory`等）同时注解的方法导致的错误。
- 包含无效标记语法的`@Tag`声明现在将被记录为警告，但会被有效地忽略掉。

JUnit Vintage

Bug修复

- 与`@Unroll`一起使用时，添加了对`Runners`的支持，这些`Runners`报告不属于`Description`树的测试事件，例如`Spock`的`Sputnik`。以前，这样的测试根本没有报告；现在它们被作为动态测试而报告。

5.0.0-M5

发布时间： 2017.07.04

范围：JUnit 5的第5个里程碑版本，重点关注三点：动态容器、测试实例生命周期管理以及次要的API更改。

⚠ 这是一次里程碑式的发布，包含重大更改。如果想在捆绑了旧版里程碑版本的IntelliJ IDEA中使用此版本，请参阅上面的 [说明](#)。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上的JUnit仓库中的[5.0 M5](#)里程碑页面。

- 所有已发布的JAR文件现在都包含一个`Automatic-Module-Name`清单属性，其值被用作该JAR文件放置在Java 9模块路径上时所定义的自动模块的名称。

表 2. 自动模块名称列表

| JAR 文件 | Automatic-Module-Name |
|---------------------------|--------------------------|
| junit-jupiter-api-.jar | org.junit.jupiter.api |
| junit-jupiter-engine-.jar | org.junit.jupiter.engine |

| JAR 文件 | Automatic-Module-Name |
|---------------------------------------|--------------------------------------|
| junit-jupiter-migrationsupport-.jar | org.junit.jupiter.migrationsupport |
| junit-jupiter-params-.jar | org.junit.jupiter.params |
| junit-platform-commons-.jar | org.junit.platform.commons |
| junit-platform-console-.jar | org.junit.platform.console |
| junit-platform-engine-.jar | org.junit.platform.engine |
| junit-platform-gradle-plugin-.jar | org.junit.platform.gradle.plugin |
| junit-platform-launcher-.jar | org.junit.platform.launcher |
| junit-platform-runner-.jar | org.junit.platform.runner |
| junit-platform-suite-api-.jar | org.junit.platform.suite.api |
| junit-platform-surefire-provider-.jar | org.junit.platform.surefire.provider |
| junit-vintage-engine-.jar | org.junit.vintage.engine |

JUnit Platform

Bug修复

- 如果选择器是通过不具有显式参数类型的`DiscoverySelectors`创建的，则 `MethodSelector.getMethodParameterTypes()` 不会为参数类型返回 `null`。具体来说，如果参数类型没有被提供并且不能推导，则返回一个空字符串。同样，如果参数类型没有明确提供，但可以推导（例如，通过提供的 `Method` 引用），`getMethodParameterTypes()` 现在返回一个包含推导的参数类型的字符串。
- `ConsoleLauncher` 现在打印对应异常的 `toString()` 方法的内容作为回退机制，而非在 `Details.TREE` 模式下抛出 `NullPointerException`。
- 现在，`DefaultLauncher` 会捕获并警告某个引擎在其发现和执行阶段生成的异常。其他引擎可被正常处理。
- 当生成字符串形式的唯一ID时，`UniqueId.Segment` 类型和值的字符串现在会部分进行URL编码。通过 `UniqueIdFormat` 所保留的字符（例如 `[`，`:`，`]` 和 `/`）会被编码。默认解析器也已经更新，从而可以解码这样的编码段。

弃用和彻底改变

- `ConsoleLauncher` 中弃用的 `--hide-details` 选项已被移除；使用 `--details none` 替代。
- 下列之前弃用的方法现在都已被移除。

- `junit-platform-engine:Node.execute(EngineExecutionContext)`
- `junit-platform-commons:ReflectionUtils.findAllClassesInClasspathRoot(Path, Predicate, Predicate)`
- `org.junit.platform.engine.support.hierarchical.Node`接口的 `isLeaf()` 方法已被移除。
- `TestDescriptor`类中的默认方法`pruneTree()`和`hasTests()`已被移除。

新特性与改进

- 当使用`DiscoverySelectors`通过完全限定的方法名称来选择一个方法，或者通过提供 `methodParameterTypes` 作为一个逗号分隔的字符串时，可以使用 *源代码语法* 来描述数组参数类型，如基本数组 `int[]` 和 `java.lang.String[]` 作为一个对象数组。此外，现在可以使用JVM的内部字符串表示来描述多维数组类型（例如，`[[[I`代表 `int[][][]`，`[[Ljava.lang.String;`代表 `java.lang.String[][]`等）或 *源代码语法*（例如，`boolean[][][]`，`java.lang.Double[][]`等）。
- JUnit的平台的Gradle插件的`junitPlatformTest`任务现在可以在构建的配置阶段直接访问。
- JUnit Platform Gradle插件与Gradle Kotlin DSL配合使用效果更佳。
- 当通过Java的`ServiceLoader`机制发现`TestEngine`时，现在将尝试确定引擎类的加载位置，如果位置URL可用，则将在配置级别进行记录。
- `mayRegisterTests()` 方法可用来表示一个`TestDescriptor`将在执行过程中注册动态测试。
- 现在可以查询`TestPlan`是否包含`Test()`。这被`Surefire provider`用来决定一个类是否是一个应该被执行的测试类。
- `ENGINE`枚举常量已从`TestDescriptor.Type`中移除。`EngineDescriptor`的默认类型现在是 `TestDescriptor.Type.CONTAINER`。

JUnit Jupiter

Bug修复

- `@ParameterizedTest`参数不再为生命周期方法和测试类构造函数解析，以提高与具有不同参数列表的常规测试方法和参数化测试方法的互操作性。

弃用和彻底改变

- 迁移支持模块现在名为`junit-jupiter-migrationsupport`，值得注意的是，在名称中，`migration`与`support`之间没有`-`。
- 为了确保JUnit Jupiter中所有受支持的扩展API的可组合性，现有API中的几个方法已被重命名。
 - 详细信息请参阅 [扩展API迁移](#)。

- `junit-jupiter-params` 模块的 `ArgumentsProvider` API 中的 `arguments()` 方法已被重命名为 `provideArguments()`。
- `junit-jupiter-params` 模块中的 `ObjectArrayArguments` 类已被删除; 通过 `Arguments.of (...)` 静态工厂方法现在可以创建 `Arguments` 实例的功能。
- `@MethodSource` 的名称属性已被重命名为值。
- `TestExtensionContext` API 中的 `getTestInstance()` 方法已被移至 `ExtensionContext` API。此外, 签名已从 `Object getTestInstance()` 更改为 `Optional <Object> getTestInstance()`。
- `TestExtensionContext` API 中的 `getTestException()` 方法已移至 `ExtensionContext` API 并重命名为 `getExecutionException()`。
- `TestExtensionContext` 和 `ContainerExtensionContext` 接口已被删除, 并且所有扩展接口已被更改为使用 `ExtensionContext`。
- `TestExecutionCondition` 和 `ContainerExecutionCondition` 已被一个用于条件测试执行的通用扩展 API 取代: `ExecutionCondition`。

表 3. 扩展API迁移对应表

| 扩展API | 曾用名 | 新名称/定位 |
|---------------------------------------|--------------------|--|
| ParameterResolver | supports() | supportsParameter() |
| ParameterResolver | resolve() | resolveParameter() |
| ContainerExecutionCondition | evaluate() | evaluateExecutionCondition() in ExecutionCondition |
| TestExecutionCondition | evaluate() | evaluateExecutionCondition() in ExecutionCondition |
| TestExtensionContext | getTestException() | getExecutionException() in ExtensionContext |
| TestExtensionContext | getTestInstance() | getTestInstance() in ExtensionContext |
| TestTemplateInvocationContextProvider | supports() | supportsTestTemplate() |
| TestTemplateInvocationContextProvider | provide() | provideTestTemplateInvocationContexts() |

新特性与改进

- 现在, 通过使用新的类级注解 `@TestInstance`, 可以使测试实例的生命周期从默认 `per-method` 模式转换到 `per-class` 模式。这使得在测试类中的测试方法之间以及测试类中的非静态 `@BeforeAll` 和 `@AfterAll` 方法之间可以共享测试实例状态。
 - 详细信息请参阅 [测试实例生命周期](#)。

- 如果测试类用`@TestInstance(Lifecycle.PER_CLASS)`进行注解，则`@BeforeAll`和`@AfterAll`方法可以不再是`static`的。同时启用以下新特性：
 - 在`@Nested`测试类中声明`@BeforeAll`和`@AfterAll`方法。
 - 在接口`default`方法上声明`@BeforeAll`和`@AfterAll`。
 - 用Kotlin编程语言实现的测试类中的`@BeforeAll`和`@AfterAll`方法的简化声明。
- `Assertions.assertAll()`现在将跟踪几乎所有类型的异常（而不是只跟踪`AssertionError`类型的异常），除非异常是一个被列入黑名单的异常，在这种情况下，它将被立即重新抛出。
- 如果`@ParameterizedTest`接受一个数组作为参数，那么当为参数化测试的调用生成显示名称时，数组的字符串表示形式将被转换为人类可读的格式。
- `@EnumSource`现在提供了一个枚举常量选择模式，用于控制如何解释提供的名称。支持的模式包括：`INCLUDE`和`EXCLUDE`，以及基于正则表达式的模式匹配，如`MATCH_ALL`模式和`MATCH_ANY`模式。
- 扩展现在可以通过使用新引入的引擎级`ExtensionContext`的`Store`来在顶级测试类中共享状态。
- 现在，`@MethodSource`引用的参数提供方法可以直接返回`DoubleStream`，`IntStream`和`LongStream`的实例。
- `@TestFactory`现在支持任意嵌套的动态容器。详情请参阅`DynamicContainer`和抽象基类`DynamicNode`。
- 现在，`ExtensionContext.getExecutionException()`向`AfterAllCallbacks`提供`@BeforeAll`方法或`BeforeAllCallbacks`中抛出的异常。

JUnit Vintage

Bug修复

- `VintageTestEngine`不再过滤掉声明为静态成员类的测试类，因为它们是有用的JUnit 4测试类。
- `VintageTestEngine`不再尝试将抽象类当作测试类来执行。相反，现在会有一个警告记录来说明这些类被排除在外。

5.0.0-M4

发布时间： 2017.04.01

范围：JUnit 5的第4个里程碑发布，主要关注点在于测试模板、重复测试和参数化测试。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上的JUnit仓库中的 [5.0 M4](#) 里程碑页面。

JUnit Platform

Bug修复

- 为了保证可重复的构建，JUnit Platform的Gradle插件为其依赖增加了一个修复版本（和插件版本相同），替换了默认的动态版本方案（即`1.+`）。

- JUnit平台的Gradle插件如今明确地应用在java插件的内置Gradle中，因为它对应用程序有一个隐含的依赖关系。
- ReflectionUtils中的所有findMethods()实现不再返回合成方法。结果中不再包含隐含的 [override-equal](#) 方法。
- 在TestIdentifier和TestDescriptor中引入getLegacyReportingName()。这使得JUnit Platform的Gradle插件和Surefire Provider能够通过getLegacyReportingName()解析类和方法名，而不再使用资源位置。
- 在路径中包含空格（%20）的JAR文件现在在用于类路径扫描例程之前已被正确解码。
- 更新了ReflectionUtils中的findNestedClasses()方法，以便搜索嵌套类也返回继承的嵌套类（无论它们是否为静态）。

弃用和彻底改变

- 所有在org.junit.platform.runner包中的测试套件注解都移到了新junit-platform-suite-api模块中的org.junit.platform.suite.api包。其中包括例如@SelectClasses,@SelectPackages等注解。
- 移除了DiscoverySelectors中基于名字的发现选择器selectNames()，推荐使用方法selectPackage(String),selectClass(String)和selectMethod(String)。
- ClassNameFilter.includeClassNamePattern被移除了，取而代之的请使用ClassNameFilter.includeClassNamePatterns。
- @IncludeClassNamePattern被移除了，请改用@IncludeClassNamePatterns。
- ConsoleLauncher的选项--hide-details被弃用了，请改用--details none
- ZIP发布包包含的控制台启动器不再提供，取代它的是一个独立的可执行的JAR发行版。更多细节见下面的“New Feature”部分。
- ReflectionUtils中的枚举类MethodSortOrder被重命名为HierarchyTraversalMode，产生的影响是应该使用ReflectionSupport来代替ReflectionUtils。
- Launcher中的方法execute(LauncherDiscoveryRequest launcherDiscoveryRequest)已经被弃用，并将在里程碑版本M5中删除。取代它的是新方法execute(LauncherDiscoveryRequest launcherDiscoveryRequest, TestExecutionListener... listeners)，除了已经注册的监听器之外，该方法还可以用于注册已提供的TestExecutionListener，但也仅限于已提供的LauncherDiscoveryRequest。

新特性与改进

- 自定义的TestExecutionListener实现现在可以通过Java的ServiceLoader机制自动注册。
- TestEngine API添加了新的默认方法getGroupId(),getArtifactId()和getVersion()，这些方法用来调试和报告。默认情况下，包属性（一般来自JAR属性清单）用来决定发行ID和版本号；而组ID默认是空的。更多细节请参阅Javadoc的 [TestEngine](#) 文档。
- 已发现的测试引擎的记录信息得到增强，包括group ID、artifact ID和每个测试引擎的版本，如果它们能够通过getGroupId(), getArtifactId(), 以及getVersion()方法获取。

- 现在，ConsoleLauncher 的`--scan-classpath`允许扫描JAR文件作为显式参数提供的测试（请参阅[Options](#)）。
- 现在，ConsoleLauncher中新式的`--details<Details>`选项允许在执行测试时选择一个输出细节的模式。可以使用`none`、`flat`、`tree`，或者`verbose`中的一个来指定该模式。如果没有指定，则只会输出汇总和测试失败信息（请参阅[Options](#)）。
- 可执行的`junit-platform-console-standalone-1.0.2.jar`包在默认的构建过程中生成，并存储于已在[Maven 中心库](#)发布的`junit-platform-console-standalone/build/libs`路径下。该jar包包含了Jupiter和Vintage测试引擎及其所有依赖。它在手动管理其依赖项的项目中提供了JUnit 5的无障碍使用，类似于JUnit 4中广为人知的[plain-old JAR](#)管理。
- ConsoleLauncher中新增加的`--exclude-classname` (`--N`)选项可以接受一个正则表达式来排除那些全类名匹配的类。当这个选项重复时，所有的模式将使用“或”语义进行组合。
- 新的JUnit Platform支持包`org.junit.platform.commons.support`包含了许多正在维护的工具方法，这些方法可用于注解、反射，以及类路径扫描任务。我们鼓励TestEngine和Extension开发人员（author）使用这些支持方法，以便与JUnit Platform的行为保持一致。
- 压缩了TestDescriptor.isTest()和TestDescriptor.isContainer()的内部逻辑，使得其返回值从两个独立的布尔属性值转变为名为TestDescriptor.Type的枚举类。详细信息请参阅下一小节。
- 引入了TestDescriptor.Type枚举类以及对应的方法TestDescriptor.getType()，用于定义所有可能的描述符类型。原来的TestDescriptor.isTest()和TestDescriptor.isContainer()现在委托给TestDescriptor.Type常量。
- 引入了TestDescriptor.prune()和TestDescriptor.pruneTree()方法，它们允许引擎作者自定义JUnit平台触发修剪时的处理办法。
- TestIdentifier现在使用新的TestDescriptor.Type枚举来存储底层类型。它可以通过新的TestIdentifier.getType()方法获取。此外，TestIdentifier.isTest()和TestIdentifier.isContainer()现在委托TestDescriptor.Type的常量。

JUnit Jupiter

Bug修复

- 修复了通过方法选择器选择时阻止在相同类中发现多个方法的错误。
- 当在超类中声明时，现在会检测到@Nested的非静态测试类。
- 现在，当在一个类层次结构中的多个级别声明时，重写的@BeforeEach和@AfterEach方法会以正确执行顺序会被强制执行，其顺序始终是`super.before->this.before->this.test->this.after->super.after`，即使编译器添加了合成（synthetic）方法。
- 现在，TestExecutionExceptionHandlers会被按照其注册顺序的反序执行，这类似于其他所有的“after”扩展。

弃用和彻底改变

- 删除已弃用的`Assertions.expectThrows()`方法，`Assertions.assertThrows()`取代。
- 由相同部分组成但顺序不同的`ExtensionContext.Namespaces`不再被认为是同一个。

新特性与改进

- 新增了`@ParameterizedTest` 注解，从而为参数化测试 提供了极好的支持。请参阅 [参数化测试](#)。
- 新增了`@RepeatedTest`注解和`RepetitionInfoAPI`,从而为 重复测试提供了极好的支持。请参阅 [重复测试](#)。
- 引入了新的注解`@TestTemplate`，并附带了对应的扩展点`TestTemplateInvocationContextProvider`。
- 现在，`Assertions.assertThrows()`方法在生成断言失败的消息时，会采用规范名称作为异常类型。
- 现在，在测试方法上注册的`TestInstancePostProcessors`会被调用。
- `Asserrions.fail`现在有了新的变体：`Assertions.fail(Throwable cause)`和`Assertions.fail(String message, Throwable cause)`。
- 新的`Assertions.assertLinesMatch()`方法会比较字符串列表，应用了`Object :: equals`和正则表达式。
`assertLinesMatch()`还提供了一个快速转发机制，可以跳过每个调用中预期会更改的行，例如持续时间，时间戳，堆栈跟踪等。详细信息请参阅 [org.junit.jupiter.Assertions](#) 的JavaDoc。
- 现在可以通过Java的`ServiceLoader`机制自动注册扩展。详情请参阅 [自动扩展注册](#)。

JUnit Vintage

Bug修复

- 修复了导致只有最后一次测试失败的报告。例如，使用`ErrorCollector`规则时，只报告最后一次失败的检查。现在，使用`org.opentest4j.MultipleFailuresError`会报告所有失败。

5.0.0-M3

发布时间： 2016.11.30

范围： JUnit 5 第三次发布，重点是JUnit 4的互操作性、附加的发现选择器以及文档。

关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上的JUnit仓库中的 [5.0 M3](#) 里程碑页面。

JUnit Platform

Bug修复

- 现在，在打印异常信息时，`ConsoleLauncher`所使用的`ColoredPrintingTestListener`输出实际异常类型以及堆栈跟踪。

- 在扫描classpath根目录时，现在通过类路径扫描来获取默认包中的测试类。例如，在没有选择显式包的情况下，会使用JUnit Platform Gradle插件。
- 类路径扫描不再加载类名过滤器排除的类。
- 类路径扫描不再尝试加载Java 9的module-info.class文件。

弃用与彻底改变

- 重命名ClasspathSelector为ClasspathRootSelector，以避免与ClasspathResourceSelector混淆。
- 重命名JavaPackageSource为PackageSource，以便于PackageSelector的命名方式保持一致。
- 重命名JavaMethodSource为MethodSource，以便于MethodSelector的命名方式保持一致。
- 现在，PackageSource，ClassSource，以及MethodSource直接实现TestSource接口，而不是已经移除的JavaSource接口。
- DiscoverySelectors中，基于名称的通用发现选择器（例如selectName()和selectNames()）已经被弃用，取而代之的是更加专用的selectPackage(String)，selectClass(String)，以及selectMethod(String)方法。
- 现在，ClassFliter已经被重命名为ClassNameFilter，并且实现了DiscoveryFilter<String>，而非之前的DiscoveryFilter<Class<?>>。因此可以在类路径扫描的期间，在类加载之前应用它。
- 现在，ClassNameFilter.includeClassNamePattern被弃用，并以ClassNameFilter.includeClassNamesPatterns取而代之。
- 现在，@IncludeClassNamePattern被弃用，并以@IncludeClassNamesPatterns取而代之。
- 用于配置ConsoleLauncher其他类路径条目的命令行选项-p已被重命名为-cp，以便与java其他可执行选项保持一致。另外，为--classpath命令行选项引入了一个新的别名--class-path，而原有命令行保持不变。
- 对ConsoleLauncher的命令行选项-a和--all已被重命名为--scan-class-path。
- 对ConsoleLauncher的短命令行选项-C，-D，以及-r都被弃用，取而代之的是对应的长命令行选项--disable-ansi-colors，--hide-details，以及--reports-dir，分别与之等价。
- ConsoleLauncher不再支持无选择参数的使用方式。请使用新的显式选择器选项以明确包名、类名或方法名。另外，--scan-class-path现在可以接受一个可选参数，该参数可被用来选择需要被扫描的类路径。涉及多条类路径时，可以使用系统自带的路径分隔符将之隔离（Windows使用;，Unix使用:）。
- LauncherDiscoveryRequestBuilder不再接收selectors，filters或配置参数映射的null值。
- 现在需要将Gradle插件配置中的类名称、引擎ID和标记过滤器包装在filters扩展元素中（请参阅[配置过滤器](#)）。

新特性

- DiscoverySelectors中新增的selectUri (...) 方法用于选择URI。TestEngine可以通过查询UriSelector的注册实例来检索这些值。

- 在DiscoverySelectors中新增的selectFile (...) 和selectDirectory (...) 方法可以用于选择文件系统上的文件和目录。TestEngine可以通过查询FileSelector和DirectorySelector的已注册实例来检索这些值。
- DiscoverySelectors中的新的selectClasspathResource (String) 方法，用于按名称选择类路径资源（如XML或JSON文件），其中名称是当前类路径中资源的分离路径名。TestEngine可以通过查询ClasspathResourceSelector的已注册实例来获取这些值。此外，可以通过新的ClasspathResourceSource将类路径资源作为TestIdentifier的TestSource。
- 现在，DiscoverySelectors中的selectMethod(String)方法支持完全限定的方法名作为参数的选择方法（例如"org.example.TestClass # testMethod(org.junit.jupiter.api.TestInfo)"）。
- 现在，ConsoleLauncher和JUnit Platform Gradle插件使用的TestExecutionSummary除了包含测试事件外，还包含所有容器事件的统计信息。
- 现在，TestExecutionSummary可以用来获取所有失败的测试用例列表。
- 现在，[ConsoleLauncher](#)、Gradle插件和[JUnitPlatform](#) 运行器使用`^.* Tests? $`正则表达式作为测试运行中所包含的类名的默认匹配模式。
- Gradle插件现在允许明确地选择应该执行哪些测试（请参阅 [配置选择器](#)）。
- 新的@Testable注解可以用来向IDE和开发工具供应商传递信息，用以说明被此注解所标注的元素是可测试的（即，它可以在JUnit Platform上作为测试被执行）。
- 现在，在使用ClassNameFilter.includeClassNamePatterns时，可以传递多个正则表达式，它们之间通过"或"语义组合。
- 现在，@IncludeClassNamePatterns注解可以将多个正则表达式以"或"的语义组合，并传递到JUnitPlatform运行器中。
- 现在，多个正则表达式可以通过"或"的语义，进行组合，并被传递给JUnit Platform Gradle 插件（参阅 [配置过滤器](#)）以及ConsoleLauncher（请参阅 [Options](#)）。
- 现在，可以通过使用PackageNameFilter.includePackageNames来指定被包含的包名或使用PackageNameFilter.excludePackageNames来指定被排除的包名。
- 现在，JUnitPlatform运行器可以结合@IncludePackages注解指定希望包含的包名，也可以结合@ExcludePackages来指定希望排除在外的包名。
- 现在，对于使用JUnit Platform Gradle插件和ConsoleLauncher的用户，可应通过配置过滤器完成包名的添加与排除。
- 包名现在可以通过JUnit Platform Gradle插件（请参阅 [配置过滤器](#)）和[ConsoleLauncher](#)（请参阅 [Options](#)）的过滤器配置来包含或排除。
- 现在，junit-platform-console不再强依赖于[JOptSimple](#)。因此，现在可以测试那些使用该库的不同版本的自定义代码。
- 现在，包选择器的解析会扫描JAR文件。
- Surefire provider现在支持forking。
- Surefire provider现在支持使用标记过滤，可以通过以下方法：
 - 包含: groups/includeTags
 - 排除: excludedGroups/excludeTags
- Surefire Provider现在获得了Apache License v2.0许可。

JUnit Jupiter

Bug修复

- 现在，`@AfterEach`标注的方法在类的层级结构中，将以自下而上的语义顺序执行。
- 现在，`DynamicTest.stream()`会为测试执行器接收一个`ThrowingConsumer`，而非以往的`Consumer`，从而允许定制动态测试流可能会抛出的检查异常。
- 现在，当为对应的测试方法调用`@BeforeEach`和`@AfterEach`方法时，该测试方法级的扩展注册会被使用。
- 现在，`JupiterTestEngine`支持通过接受数组或基本类型参数的唯一标识（方法的ID）来选择测试方法。
- 现在，`ExtensionContext.Store`是线程安全的。

弃用和彻底改变

- `Executable`函数式接口已经被搬移到了新的专用包`org.junit.jupiter.api.function`中。
- `Assertions.expectThrows()` 已经被弃用，并以`Assertions.assertThrows()`取而代之。

新特性与改进

- 支持`Assertions`中的`lambda`表达式的延迟和抢占超时。请参阅 [AssertionsDemo](#) 中的示例，更多详细信息请参阅 [org.junit.jupiter.Assertions](#) JavaDoc。
- 新的`assertIterableEquals()`断言会校验两个可迭代对象是否深度一致（查阅Java文档获取细节）。
- `Assertions.assertAll()`的新变体接收可执行流（即，`Stream<Executable>`）。
- 现在，`Assertions.assertThrows()`方法将返回抛出的异常。
- 现在，`@BeforeAll`与`@AfterAll`可以被声明在接口中的静态方法上。
- JUnit Jupiter现在支持JUnit 4中的`org.junit.rules.ExternalResource`、`org.junit.rules.Verifier`、`org.junit.rules.ExpectedException`规则及其子类，这更有利于JUnit 4代码库的迁移。

JUnit Vintage

自5.0.0-M2以来没有变化。

5.0.0-M2

发布时间： 2016.07.23

范围： JUnit 5的第2个里程碑发布

变更摘要

此版本主要是一个bug修复版本，修复了自5.0.0-M1以来所发现的bug。

以下是一个全局改变的列表。关于更改的详细信息，包括Platform、Jupiter和Vintage，请参阅下面的章节。关于此版本所有已关闭的问题和pull request的完整列表，请参阅GitHub上的JUnit仓库中的 [5.0 M2](#) 里程碑页面。

- JUnit 5的Gradle构建可以正常在Microsoft Windows下运行。
- 在AppVeyor上建立了针对Microsoft Windows的持续集成构建。

JUnit Platform

修复Bug

- 容器中的故障—例如，抛出异常的@BeforeAll方法—如今在使用ConsoleLauncher或JUnit Platform Gradle插件时会构建失败。
- JUnit Platform Surefire Provider不再默默地忽略纯动态测试类——例如只声明@TestFactory方法的测试类。
- 在junit-platform-console-<release version> TAR和ZIP发布包中包含的junit-platform-console和junit-platform-console.bat shell脚本能够正确引用ConsoleLauncher，而不是ConsoleRunner。
- 被ConsoleLauncher和JUnit Platform Gradle插件使用的TestExecutionSummary包含了失败的错误信息。
- 类路径扫描现在可以防止类加载和处理期间遇到的问题—例如，在处理格式错误的类时，潜在的异常被吞噬并记录在有问题的文件路径中，如果这个异常是一个列入黑名单的异常，比如OutOfMemoryError，那么它将被重新抛出。

弃用

- DiscoverySelectors中基于通用名称的发现选择器（即selectName()和selectNames()）已经被废弃，建议使用selectPackage(String)、selectClass(String)和selectMethod(String)方法。

新特性

- DiscoverySelectors中的新方法selectMethod(String)支持选择完全限定方法名。

JUnit Jupiter

修复Bug

- 在类级别和方法级别通过@ExtendWith声明的扩展实现将不再被多次注册。

JUnit Vintage

自5.0.0-M1以来没有变化。

5.0.0-M1

发布时间： 2016.07.07

范围： JUnit 5的第一个里程碑发布

变更摘要

下面首先给出整体改动。而有关Platform、Jupiter，以及Vintage的细节变动，可以查阅后面的专项信息。对于本次发布相关的commit信息，可以通过查看 [5.0 M1](#)里程碑页面在GitHub上的JUnit代码库了解。

- 在已发布的JAR清单中，包含了额外的元数据，例如Create-By、Built-By、Build-Date、Build-Time、Build-Revision、Implementation-Title、Implementation-Version、Implementation-Vendor等。
- 当前发布的工件中，包含了LICENSE.md和META-INF。
- 现在，JUnit参与到 [Up For Grabs](#) 运动中，以便为开源做贡献。
 - 请参阅GitHub上的 [up-for-grabs](#) 标签。
- 对于已发布的Artifact，其Group ID，Artifact ID，以及版本都已改变。
 - 查看 [工件迁移](#) 以及 [依赖元数据](#)。
- 所有的基础包都被重命名了。
 - 查看 [包迁移](#)

表 4. 工件迁移信息表

| 旧的Group ID | 旧的Artifact ID | 新的Group ID | 新的Artifact ID | 新的Base版本 |
|------------|------------------|--------------------|----------------------------------|----------|
| org.junit | junit-commons | org.junit.platform | junit-platform-commons | 1.0.0 |
| org.junit | junit-console | org.junit.platform | junit-platform-console | 1.0.0 |
| org.junit | junit-engine-api | org.junit.platform | junit-platform-engine | 1.0.0 |
| org.junit | junit-gradle | org.junit.platform | junit-platform-gradle-plugin | 1.0.0 |
| org.junit | junit-launcher | org.junit.platform | junit-platform-launcher | 1.0.0 |
| org.junit | junit4-runner | org.junit.platform | junit-platform-runner | 1.0.0 |
| org.junit | surefire-junit5 | org.junit.platform | junit-platform-surefire-provider | 1.0.0 |
| org.junit | junit5-api | org.junit.jupiter | junit-jupiter-api | 5.0.0 |
| org.junit | junit5-engine | org.junit.jupiter | junit-jupiter-engine | 5.0.0 |
| org.junit | junit4-engine | org.junit.vintage | junit-vintage-engine | 4.12.0 |

表 5. 包迁移信息表

| 旧基础包名 | 新基础包名 |
|------------------------------|--------------------------------------|
| org.junit.gen5.api | org.junit.jupiter.api |
| org.junit.gen5.common | org.junit.platform.common |
| org.junit.gen5.console | org.junit.platform.console |
| org.junit.gen5.engine.junit4 | org.junit.vintage.engine |
| org.junit.gen5.engine.junit5 | org.junit.jupiter.engine |
| org.junit.gen5.engine | org.junit.platform.engine |
| org.junit.gen5.gradle | org.junit.platform.gradle.plugin |
| org.junit.gen5.junit4.runner | org.junit.platform.runner |
| org.junit.gen5.launcher | org.junit.platform.launcher |
| org.junit.gen5.launcher.main | org.junit.platform.launcher.core |
| org.junit.gen5.surefire | org.junit.platform.surefire.provider |

JUnit Platform

- 重命名ConsoleRunner为ConsoleLauncher。
- 现在，ConsoleLauncher在退出时总会返回状态码，并且已经移除了激活退出码的标志位。
- junit-platform-console不再在junit-platform-runner、junit-jupiter-engine或junit-vintage-engine上定义传递依赖。
- 现在，JUnit5运行器已经被重命名为JUnitPlatform。
 - @Packages已经被重命名为@SelectPackages。
 - @Classes已经被重命名为@SelectClasses。
 - @UniqueIds已经被移除。
 - 引入了新的注解@UseTechnicalNames。
 - 详情请参阅 [显示名称与技术名称](#)
- Gradle的JUnit Platform插件已经全面修订。
 - 现在，JUnit Platform的Gradle插件要求Gradle版本在2.5或更高版本。
 - Gradle中名为junit5Test的任务已经被重命名为junitPlatformTest。
 - Gradle中名为junit5的配置已经被重命名为junitPlatform。
 - runJunit4已经被enableStandradTestTask替换。
 - version已经被platformVersion替换。
 - 详情请查阅 [Gralde](#)。

- XML测试报告已被全面修正。
 - 现在，XML报告包含换行。
 - 现在，在JUnit Platform中明确定义的，但是在标准XML文档属性中不包含的属性元素，会被包含在<system-out>元素中的CDATA块中。
 - 现在，测试报告中会用全限定类名和真正的方法名来取代先前的显示名称。
- 现在，TestIdentifier中的唯一标识符是String类型。
- 现在，TestSource是有着专用层级结构的接口，它由CompositeTestSource、JavaSource、JavaPackageSource、JavaClassSource、JavaMethodSource、UriSource、FileSystemSource、DirectorySource和FileSource构成。
- 新增了DiscoverySelectors类，用于集中管理所有的select方法，同时将所有DiscoverySelector 工厂方法转移到新增类中。
- Test.filter()已经被重命名为Filter.apply()。
- TestTag.of()已被重命名为TestTag.create()。
- TestDiscoveryRequestBuilder已被重命名为LauncherDiscoveryRequest。
- 现在，LauncherDiscoveryRequest是不可变的。
- TestDescriptor.allDescendants()已经被重命名为TestDescriptor.getAllDescendants()。
- TestEngine#discover(EngineDiscoveryRequest)已经被TestEngine#discover(EngineDiscoveryRequest, UniqueId)替换。
- 引入了ConfigurationParameters，Launcher可以通过EngineDiscoveryRequest和ExecutionRequest将配置参数传递给引擎。
- Container和Leaf抽象类已经从HierarchicalTestEngine中移除。
- getName()方法已经被从TestIdentifier和TestDescriptor中移除，取而代之的是通过TestSource获取一个实现类的具体名称。
- 现在，测试引擎在本质上是完全动态的。也就是说，TestEngine无需在发现阶段创建TestDescriptor条目；现在，TestEngine现在可以选择在执行阶段注册容器和测试。
- 包括和排除对引擎和Tag的支持已经完全修改。
 - 引擎和Tag不再是required，而是included。
 - 现在，ConsoleLauncher支持以下选项：t/include-tag、T/exclude-tag、e/include-engine、E/exclude-engine。
 - 现在，Gradle插件支持嵌套在include和exclude实体中的engines和tags配置块。
 - 现在，EngineFilter支持includeEngines()和excludeEngines()工厂方法。
 - 现在，JUnitPlatform运行器支持@IncludeTags、@ExcludeTags、@IncludeEngines以及@ExcludeEngines。

JUnit Jupiter

- junit5引擎ID已经被重命名为junit-jupiter。
- JUnit5TestEngine已经被重命名为JupiterTestEngine。
- 现在，Assertions提供了以下支持：

- `assertEquals()` 方法可以对基本类型使用
- `assertEquals()` 方法可以对包含增量的 `double` 类型和 `float` 类型的值使用
- `assertArrayEquals()`
- 现在期望值与实际值都被提供给 `AssertionFailedError`。
- [动态测试](#)：现在，测试可以通过 `lambda` 表达式在运行时被动态注册。
- 现在，`TestInfo` 通过 `getTags()` 方法提供了获取 `Tag` 的方法。
- 如果 `@Test`、`@BeforeEach` 或 `before` 回调引发异常，现在将调用 `@AfterEach` 方法和 `after` 回调函数。
- 现在，`@AfterAll` 注解所标注的方法以及 `after all` 回调一定会被调用。
- 现在，可以在测试类层次结构中的超类以及接口中发现可重复的注解，例如 `@ExtendWith` 和 `@Tag`。
- 现在，在测试类或接口的层级结构中，扩展将会被 *自上而下* 地注册。
- 现在，测试和容器的 [执行条件可以被禁用](#)。
- `InstancePostProcessor` 已被重命名为 `TestInstancePostProcessor`。
 - 现在，`TestInstancePostProcessor` 实现正确地应用在 `@Nested` 测试类层次结构中。
- `MethodParameterResolver` 已被重命名为 `ParameterResolver`。
 - 现在，`ParameterResolver` API 是基于 `java.lang.reflect.Executable` 的，因此可以被用来解析方法和构造器的参数。
 - 新的 `ParameterContext` 用来作为参数传递给 `ParameterResolver` 扩展的 `supports()` 和 `resolve()` 方法。
 - 现在，`ParameterResolver` 扩展支持基础类型的解析。
- `ExtensionPointRegistry` 和 `ExtensionRegistrar` 已经被移除，现在通过 `@ExtendWith` 注解完成声明式注册。
- `BeforeAllExtensionPoint` 已经被重命名为 `BeforeAllCallback`。
- `AfterAllExtensionPoint` 已经被重命名为 `AfterAllCallback`。
- `BeforeEachExtensionPoint` 已经被重命名为 `BeforeEachCallback`。
- `BeforeAllExtensionPoint` 已经被重命名为 `BeforeAllCallback`。
- 新增了 `BeforeTestExecutionCallback` 与 `AfterTestExecutionCallback` 扩展 API。
- `ExceptionHandlerExtensionPoint` 已经被重命名为 `TestExecutionExceptionHandler`。
- 现在，测试异常通过 `TestExtensionContext` 被提供给扩展。
- `ExtensionContext.Store` 现在支持许多方法的类型安全变体。
- 现在，`ExtensionContext.getElement()` 方法返回 `Optional` 类型。
- `Namespace.of()` 已经被重命名为 `Namespace.create()`。
- `TestInfo` 和 `ExtensionContext` 新增了 `getTestClass()` 和 `getTestMethod()` 方法。
- 移除了 `TestInfo` 和 `ExtensionContext` 中的 `getName()` 方法，现在通过当前的测试类或测试方法来获取上下文特定的名称。

JUnit Vintage

- `junit4` 引擎 ID 已经被重命名为 `junit-vintage`。
- `JUnit4TestEngine` 已经被重命名为 `VintageTestEngine`。

5.0.0-ALPHA

发布时间： 2016.02.01

范围： JUnit 5的Alpha版本

版本 5.0.2

最后更新 2017-09-16 20:47:17 CEST

Posted by [Yuan Shenjian](#) • January 1st, 2018 @ [ThoughtWorks®](#)

版权声明： 自由转载•非商用•非衍生•保持署名 | [Creative Commons BY-NC-ND 3.0](#)

原文链接： <http://sjyuan.cc/junit5/user-guide-cn>

[支持原创](#)

