

# 6.6 开发测试

---

- 开发测试概念
- Spring 中的测试



# 目录

1

开发测试概念

2

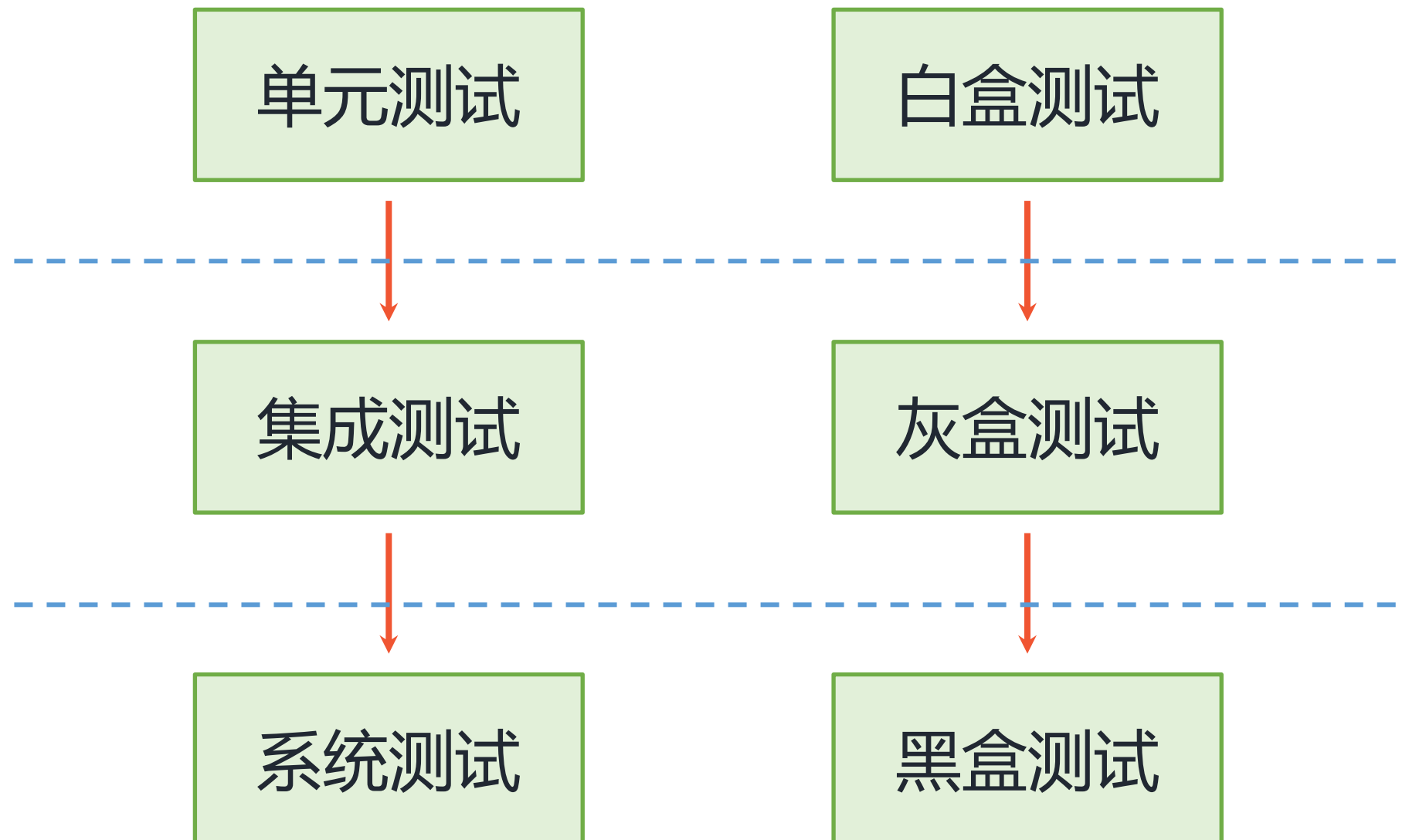
Spring 中的测试

# 开发测试

- 在软件开发过程中，我们写完某一段代码后，需要在各种可能的情况下运行它，对其正确性进行测试。然而，人力测试费时费力，还可能出现遗漏某些特殊情况的问题。
- **开发测试**[開発テスト]，就是指专门编写另外一些用于测试的代码，检测已编写的代码的正确性。
- 我们需要编写一些代码模拟可能的运行情况（**测试用例**），并根据对功能的要求给出每种情况下应有的运行结果。
- 运行测试代码后，我们会知道哪些状况下代码可以正确运行，满足功能需求；哪些情况下代码的运行结果和预期不符。在后一种情况出现时，我们便可以尝试对现有代码进行修改，并重复测试，直到所有测试都通过为止。

# 测试的三个阶段

- 概括来说，开发测试主要可以分为三个阶段，对应三种测试类型：





# 单元测试

- **单元测试**[单体テスト] (Unit Test, UT) , 就是指对最小单位量的代码进行独立的测试。
- 在传统面向过程开发中, 最小的单位可能是一个单独的程序或函数, 而在现代面向对象开发中, 最小单位一般指**对象的一个方法**。
- 单元测试的是证明组成软件的各个部分至少独立运行起来是正确的。因此, 单元测试一般是**最先**进行的测试。
- Java (Spring) 中提供了很方便的单元测试框架。我们几乎总是使用这些框架编写单元测试代码。

# 单元测试的重要性

- 一个尽责的单元测试方法将会在产品开发**初期**的某个阶段就发现很多的 Bug，使我们修改它们的成本不至于太高。系统开发的后期阶段，Bug 的检测和修改将会变得更加困难，并要消耗大量的时间和开发费用。
- 因此，无论何时，当我们对产品代码做出修改时，都应该进行完整的回归测试。在生命周期中尽早的对产品代码进行测试将是开发效率和产品质量最好的保证。
- 完善的单元测试能够大大简化**系统集成**的过程。开发人员可以将精力集中在单元之间的交互作用和全局的功能实现上，而不会陷入充满 Bug 的各个单元之中不能自拔。

# 优秀的单元测试

- 优秀的单元测试应该是**及时的、自动化的、可重复的**。
- 我们应该尽早地进行软件单元测试，以保障之后开发和测试的顺利进行。
- 我们应该尽可能地采用自动化的测试手段来进行单元测试活动，以加快单元测试的效率，减少重复劳作。
- 我们应该保证单元测试的可重复性，以应对代码的扩展或修改。
- 使测试工作效率最大化的关键在于选择正确的测试策略：这不仅要求我们完全理解单元测试的概念，还要对测试过程进行良好管理，并适当地使用好工具来支持测试过程。

# 单元测试与封装

- 我们知道 (↩ § 2.3.2)，面向对象的核心概念之一是**封装**。某些方法、属性不应该被外部类调用和访问，即私有方法、属性。
- 然而，在单元测试的过程中，我们需要了解这些私有方法的底层实现，因为这些代码恰恰可能是单元测试失败的原因所在。
- 因此，我们在单元测试的过程中一般会采取**白盒测试**[ホワイトボックステスト]的模式。单元测试代码将由**实际开发该单元的程序员亲自书写**。这些程序员了解、负责这些单元的底层实现，可以根据测试结果修改私有方法中的代码。



# 覆盖率

- 白盒测试中的**覆盖率**[カバレッジ]，指在测试过程中运行到的代码，占程序全部代码的比例。
- 一般来说，我们希望单元测试的覆盖率尽量达到 100%。如果代码的覆盖率没有达到 100%，就说明测试用例没有覆盖到我们开发时考虑到的所有情况，我们可能需要编写一些新的测试用例以检测这些代码，或者考虑这些代码是不是多余的，可以被删去。
- 需要注意，覆盖率为 100% 仅仅是测试完成的参考标准之一，并不代表代码一定没有问题。比如，可能存在一些代码和测试用例都没有考虑到的特殊情况。同时，有些测试（尤其是黑盒测试）即使覆盖率达不到 100% 也能保证应有的功能的正确性。

# 覆盖率的标准

- 我们可以把“覆盖”这一概念的标准大体分为 3 种：
  1. **命令覆盖**[命令網羅]：每一行语句都被执行过。
  2. **分支覆盖**[分歧網羅]：每一行语句都被执行过，并且每个分支语句判断条件为真、伪时的情况都出现过。
  3. **条件覆盖**[条件網羅]：在分支覆盖的基础上，当判断条件为多个条件组合而成时，每一种条件真伪的**组合**都出现过。

- 比如对于这段代码：

```
1 if (x > 3 && y < 4) {  
2     System.out.println(x - y);  
3 }  
4 System.out.println(x + y);
```

- 要做到命令覆盖，我们只需要测试  $x = 4$ 、 $y = 3$  这种情况就可以。要做到分支覆盖，我们至少需要再测试  $x = 3$ 、 $y = 3$  这种使条件为伪的情况。要做到条件覆盖，我们可能还需要测试  $x = 4$ 、 $y = 5$  这种其它的条件组合。

# 集成测试

- **集成测试**[統合テスト] (Integration Testing, IT) , 指把两个或多个相依赖的软件模块作为整体进行测试。
- 通常, 集成测试是在单元测试之后进行的。一个通过了单元测试的单元, 其自身功能的正确性得到了保障, 但在与其它单元交互时仍可能产生问题。集成测试的目的就是确保不同的单元交互合作时, 仍然能正确地实现预期中的功能。
- 很多软件项目会有多个开发人员为不同的模块或单元编写代码。因此, 通过集成测试验证由不同人员编写的代码构成的系统能否正确运行是至关重要的。



# 集成测试关注的重点

- 集成测试应当优先关注以下 2 大（5 个）问题：

1. 模块间的接口（接口的覆盖率）：

- ① 在把各个模块连接起来的时候，穿越模块接口的数据是否会丢失？
- ② 全局数据结构是否有问题，会不会被异常修改？

2. 集成后的功能（参数的传递）：

- ① 各个子功能组合起来，能否达到预期要求的父功能？
- ② 一个模块的功能是否会对另一个模块的功能产生不利的影响？
- ③ 单个模块的误差积累起来，是否会放大，从而达到不可接受的程度？

# 集成测试的实行模式

- 集成测试的根本目的是为了检验功能的正确运行，因此一般会采取**黑盒**[ブラックボックス]或**灰盒**[グレーボックス]测试模式，编写测试代码者**不应该**了解模块的内部实现，而是基于**预期的功能**选择和编写测试用例。
- 另外，软件系统中存在大量的模块和层级，从一开始就把所有模块集成到一起进行测试（*Big-bang* 模式）将导致测试人员在出现问题时很难定位到有问题的模块。因此，实际测试过程中一般先集成少量模块进行测试，在该测试通过后再集成上新的模块，如此重复直至所有模块都通过集成测试。根据进行集成的模块的添加顺序，又可分为**自底而上**[ボトムアップ]、**自顶而下**[トップダウン]等模式。

# 系统测试

- **系统测试**[システムテスト] (System Testing, ST)，指将整个产品的代码放在实际的运行环境下进行的完整测试。
- 在系统测试中，经过集成测试的软件将会作为计算机系统的一部分，与系统中其他部分结合起来，在实际运行环境下进行一系列严格有效的测试。系统测试能够发现软件潜在的问题，保证系统的正常运行。
- 集成测试和系统测试之间的比较：
  - 测试内容：集成测试测试各个单元模块之间的接口，而系统测试则测试整个系统的功能和性能；
  - 测试角度：集成测试偏重于技术角度，而系统测试则偏重于业务角度。



Q & A

*Question and answer*

# 目录


1

开发测试概念

2

Spring 中的测试

# JUnit

- JUnit 是在 Java 中最流行的单元测试框架。使用 JUnit，我们可以统一高效地开发单元测试代码，并且测试结果同样清晰可读。虽然主要用作单元测试，但 JUnit 也可在一定程度上支持集成测试的编写。
- Spring Boot 支持并推荐使用最新版本 JUnit 5。
- JUnit 5 的完整 API 和详细使用教程可以查看官方文档：  
 <https://junit.org/junit5/docs/current/user-guide/>



# JUnit 的使用

- Spring Boot 中集成了 JUnit 的组件（spring-boot-starter-test）。我们无需添加依赖即可直接开始编写测试代码。
- 如果你要在一个 Spring 以外的项目使用 JUnit，添加对应的 Maven 依赖即可：

```
1 <dependency>
2     <groupId>org.junit.jupiter</groupId>
3     <artifactId>junit-jupiter-engine</artifactId>
4     <version>5.4.0</version>
5     <scope>test</scope>
6 </dependency>
```

# 测试类的声明

- 我们通常将测试代码放置于单独的**测试目录**中。对于单元测试，我们通常会为每一个被测试类创建一个对应的**测试类**。测试类的包名和类名也应该和被测试类相对应：

```
▼ src/main/java
  ▼ net.lighthouseplan.spring.junit
    ▼ controllers
      > LoginController.java
      > ReigisterController.java
    > models
    > repositories
  ▼ services
    > AccountService.java
    > DemoMethods.java
```

```
▼ src/test/java
  ▼ net.lighthouseplan.spring.junit
    ▼ controllers
      > LoginControllerTest.java
      > RegisterControllerTest.java
    > integration
  ▼ services
    > AccountServiceTest.java
    > DemoMethodsTest.java
```



# 测试方法的声明

- 单元测试的单位通常是类的方法。我们通常会为被测试类的每一个方法声明对应的**测试方法**。测试方法都可以设为 `public`，不需要传入参数，也不需要返回有返回值。
- 在 JUnit 中，我们可以使用 **@Test** 注解声明一个方法是测试方法，方便之后执行测试：

```
1 @Test
2 public void testIsOdd() {
3     // Test code
4 }
```



# 对应多种情况的测试方法

- 同一个被测试方法可能会根据不同的输入值产生不同的结果。为了测试所有情况下的正确性，我们应该**为每一种可能的输入情况单独声明一个方法**。比如，服务类中验证用户名和密码的方法，可能验证成功，也可能因为用户名或密码不正确而验证失败，我们应该分别定义测试这些情况的测试方法：

```
// 正しいユーザーネームとパスワードを入力した場合
public void testValidateAccount_CorrectInfo_ReturnTrue() { }
// 間違ったユーザーネームを入力した場合
public void testValidateAccount_WrongUsername_ReturnFalse() { }
// 間違ったパスワードを入力した場合
public void testValidateAccount_WrongPassword_ReturnFalse() { }
```

# 断言

- **断言** [アサーション] 指声明某一个变量或对象应该满足的**预期结果**。我们可以在一个测试方法中声明一个或多个断言，表示“运行完被测试方法后，这些变量应该满足这些条件”。
- 如果运行测试代码时这些断言都成立，则该方法通过测试。如果有一个断言没有成立，测试会立刻终止并抛出异常，使我们知道实际的运行结果以及它和预期结果的差别，以助我们修正代码。

# 常用断言方法

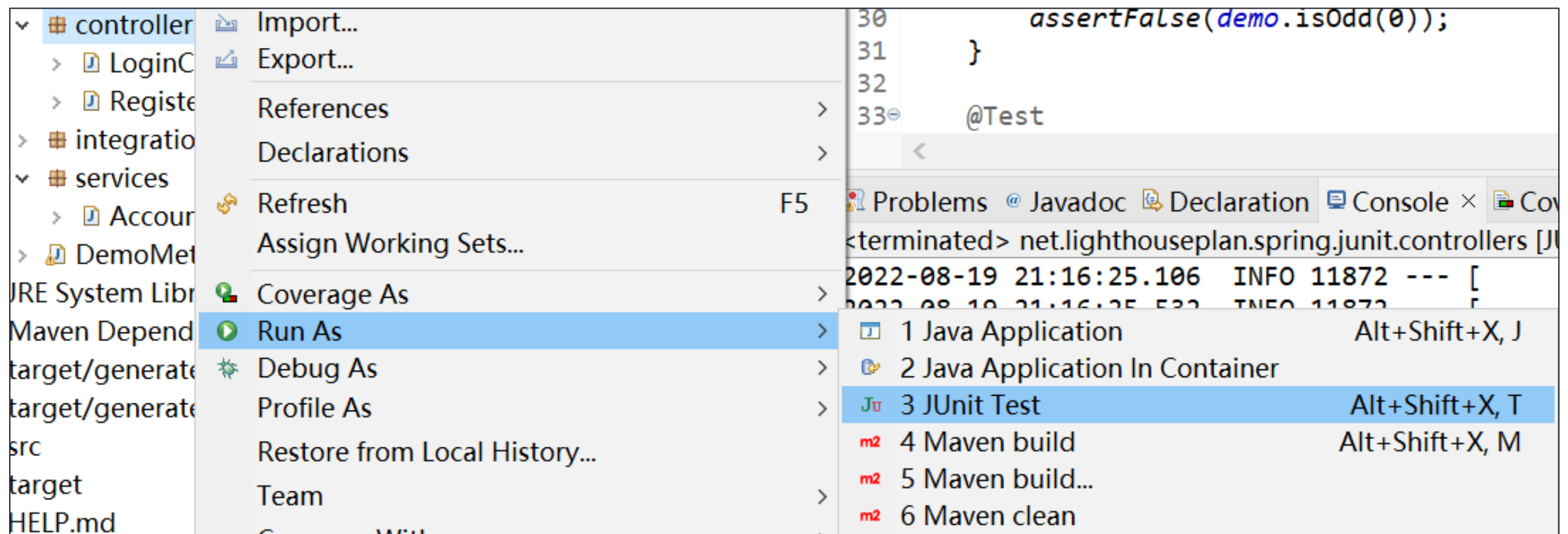
- 在 JUnit 中，我们可以使用 **assertXxx()** 系列方法声明某一个断言。以下列出常用的断言方法：

断言方法	描述
assertNull() assertNotNull()	断言对象为空 / 不为空
assertEquals() assertNotEquals()	断言变量值相等（接近） / 不相等（接近）
assertSame() assertNotSame()	断言对象（引用）相同 / 不同
assertTrue() assertFalse()	断言布尔表达式为真 / 伪
assertThat()	断言各种条件成立



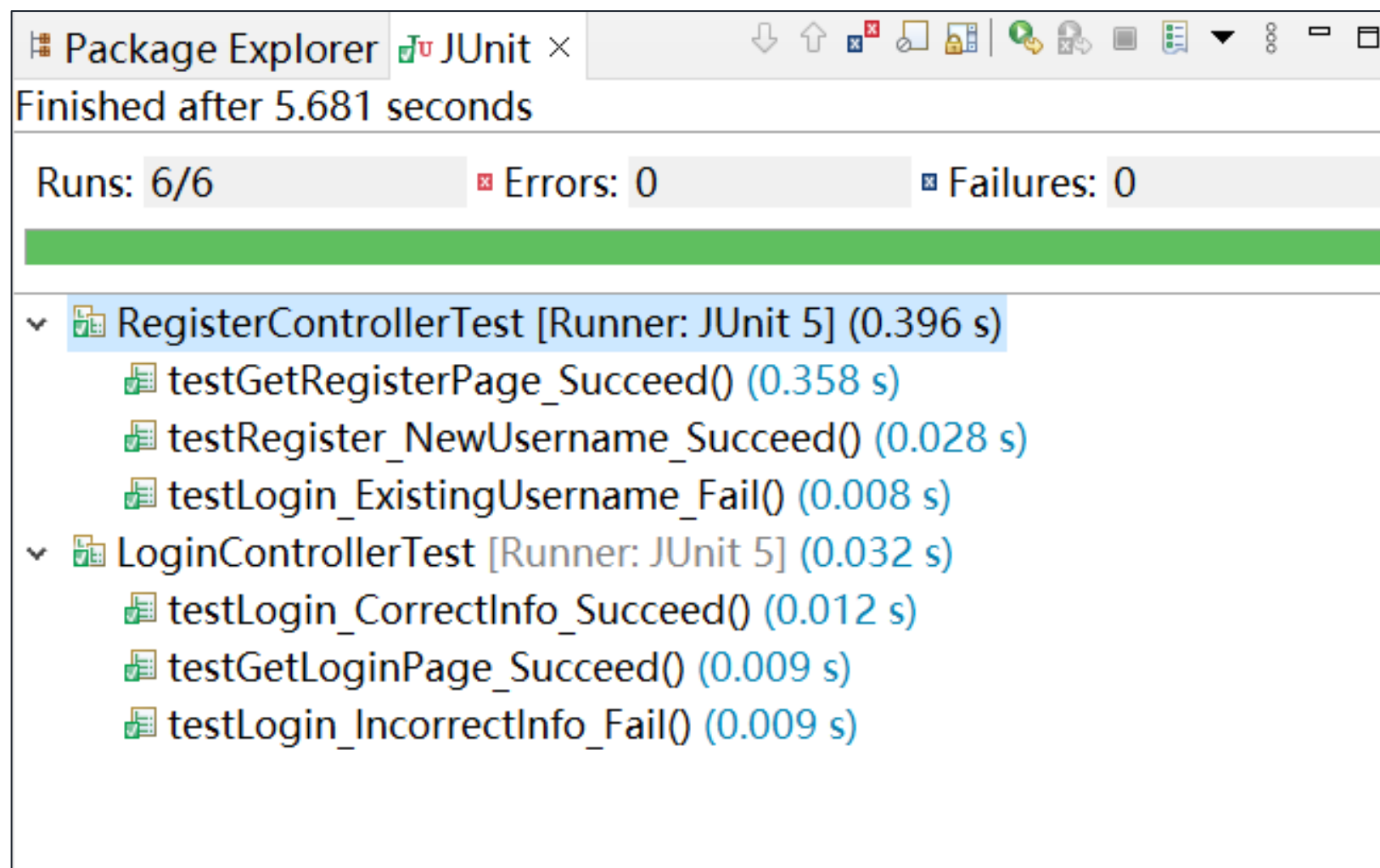
# 运行测试

- 当我们编写好测试方法后，可以选择单独运行某个测试方法、某个测试类中的全部方法或某个包中的全部方法。只要分别右键点击方法、类或包，选择 **Run As → JUnit Test**:



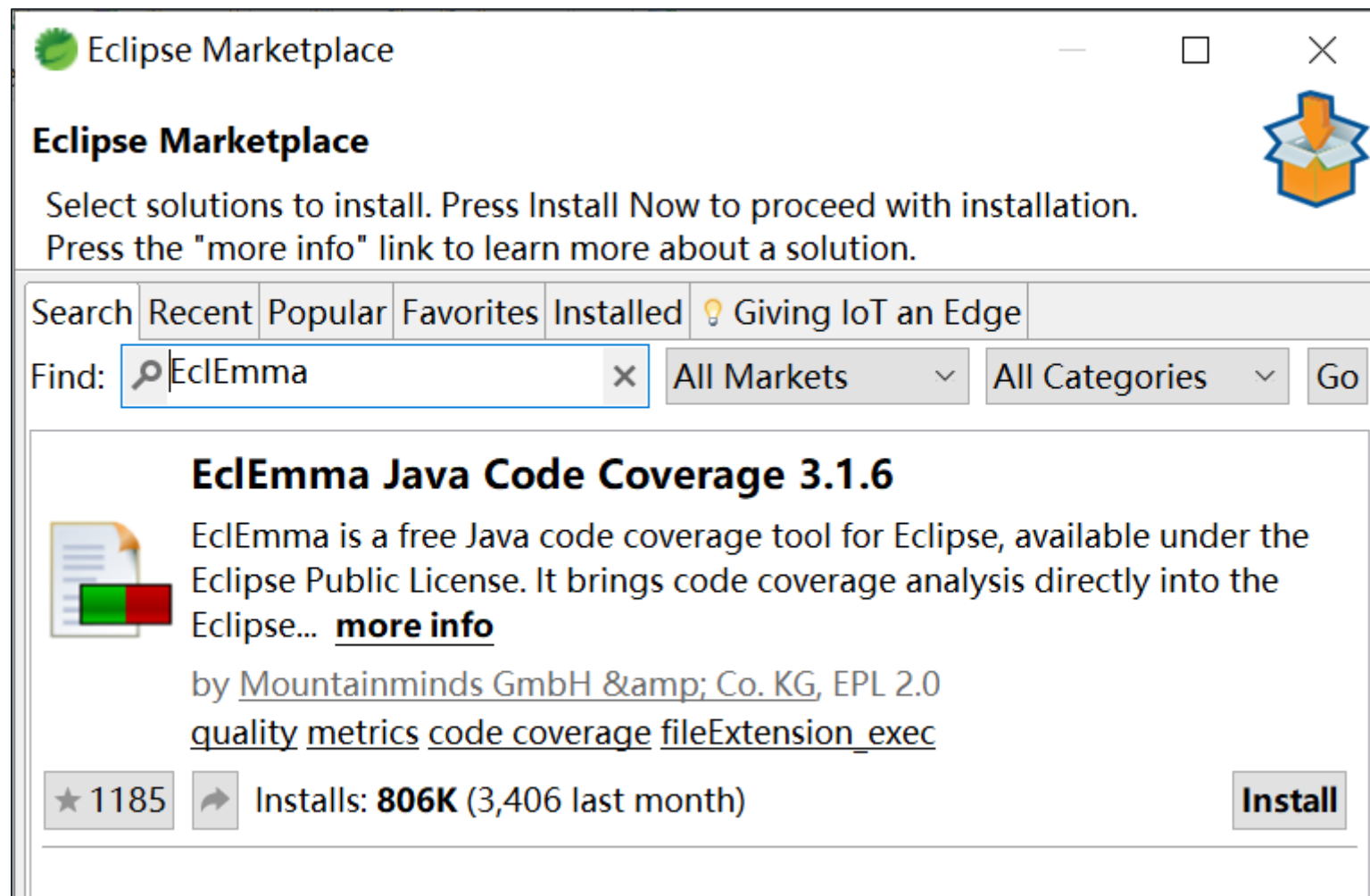
# 查看测试结果

- 点击包浏览器右侧的 JUnit 标签，可以查看各个测试方法的运行结果：



# 安装 EclEmma

- 要查看测试的覆盖率，需要安装额外插件。点击工具栏中的 Help → Eclipse Marketplace，搜索 **EclEmma** 并安装：

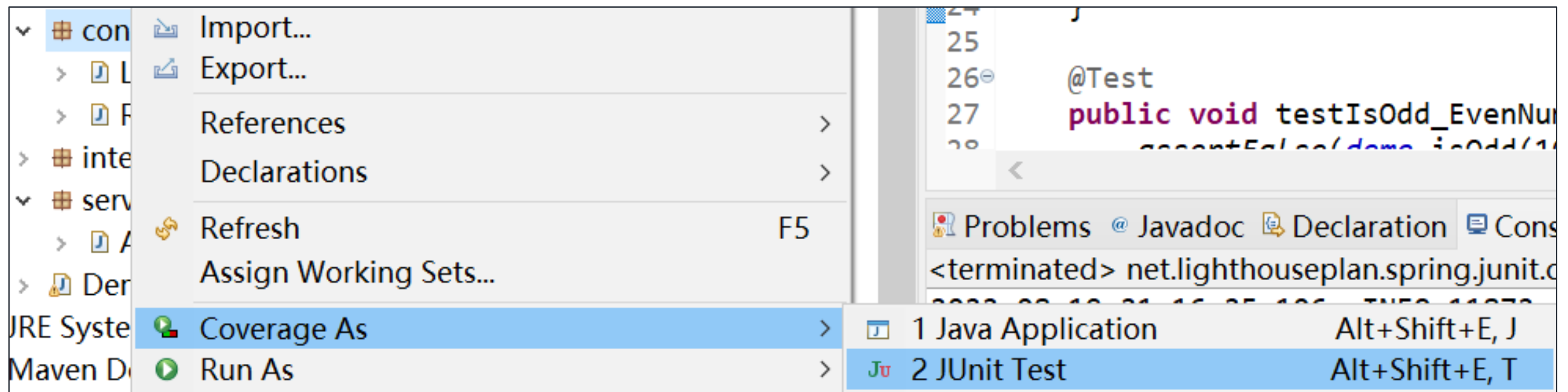


- 安装完成后需要重新启动 STS。



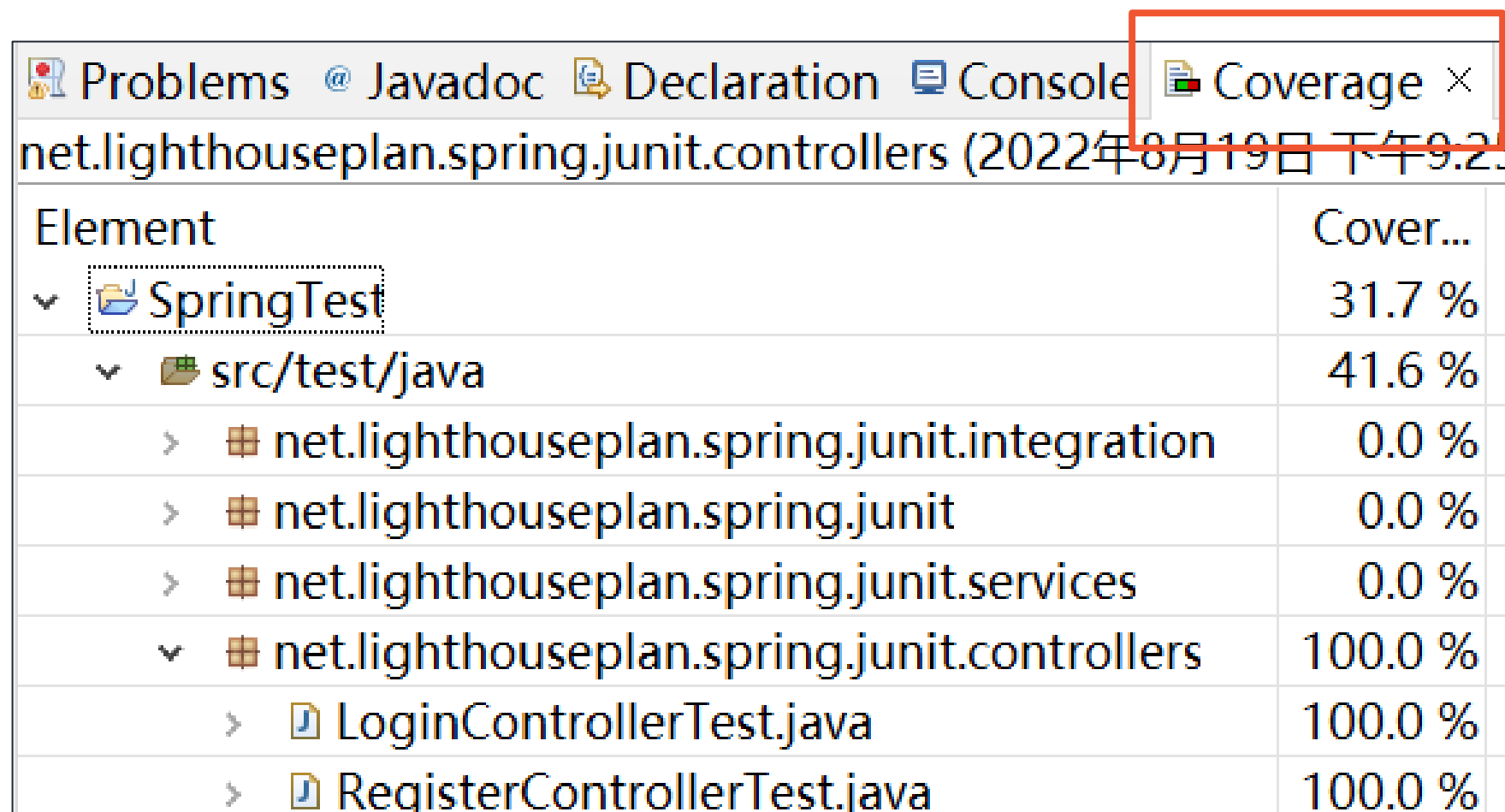
# 带有覆盖率的测试

- 能够查看覆盖率的测试方法和普通测试类似，只是要选择 Coverage As 而不是 Run As:



# 查看覆盖率

- 测试完成后，可以在点击控制台面板上的 Coverage 标签查看各个包、类或方法的代码覆盖率：



Problems @ Javadoc Declaration Console Coverage ×	
net.lighthouseplan.spring.junit.controllers (2022年8月19日 下午9:23)	
Element	Cover...
▼ SpringTest	31.7 %
▼ src/test/java	41.6 %
> net.lighthouseplan.spring.junit.integration	0.0 %
> net.lighthouseplan.spring.junit	0.0 %
> net.lighthouseplan.spring.junit.services	0.0 %
▼ net.lighthouseplan.spring.junit.controllers	100.0 %
> LoginControllerTest.java	100.0 %
> RegisterControllerTest.java	100.0 %

# JUnit 中的特殊注解

- 有时，在同一个类的各个方法的测试过程中，我们会做一些重复的操作。比如，我们需要在每个测试前创建一个测试类的对象。再如，我们需要在每个测试结束后关闭一些资源（如调用 `close()` 方法）。
- JUnit 提供了一些修饰方法的注解帮助我们简化测试代码：

注解	描述
@BeforeEach	方法会在每个测试进行前执行
@AfterEach	方法会在每个测试进行后执行
@BeforeAll	方法会在所有测试进行前执行
@AfterAll	方法会在所有测试进行前执行



Q & A

*Question and answer*

# Spring 中对控制器的测试

- MVC 模型中控制器（Controller）的各个方法通常只在接受某个 HTTP 请求时被调用。因此，我们需要特殊的方法模拟发送 HTTP 请求的过程。
- 要进行对控制器的测试，我们需要首先在测试类前添加注解，令 Spring 帮助我们进行 MVC 测试的配置：

```
1 @SpringBootTest
2 @AutoConfigureMockMvc
3 public class RegisterControllerTest {
4     // Test code
5 }
```

# MockMvc 对象

- 接下来，我们需要创建一个 **MockMvc** 对象。该对象将会模拟一个服务器，接受我们发送的 HTTP 请求并以方便我们验证的形式做出响应：

```
1 @SpringBootTest
2 @AutoConfigureMockMvc
3 public class RegisterControllerTest {
4     @Autowired
5     private MockMvc mockMvc;
6 }
```

- 和之前我们看到的一样，@Autowired 注解可以直接使用 mockMvc 对象而无需实例化它。



# 模拟 HTTP 请求

- 要模拟客户端发送的 HTTP 请求，我们需要通过 **MockMvcRequestBuilders** 提供的方法手动生成一个请求：

```
1 RequestBuilder request = MockMvcRequestBuilders
2     .post("/register")
3     .param("username", "Bob")
4     .param("password", "Bob54321");
```

- 这里第 2 行的 **post()** 表示该 HTTP 请求使用 POST 方法，可以写成 **get()** 对应 GET 方法。
- **param()** 方法设置请求中的各个参数。

# 模拟处理请求

- 接下来，我们就可以使用 MockMvc 对象的 **perform()** 方法处理这个请求，并断言一些条件：

```
1 mockMvc.perform(request)
2           .andExpect(view().name("login.html"))
3           .andExpect(model().attributeDoesNotExist("error"));
```

- 有很多可用的断言方法，最常用的包括：
  - `view().name("xxx.html")`: 断言视图（网页或模板文件）的名称；
  - `model().attribute("error", true)`: 断言传向网页的参数为特定的值；
  - `model().attributeDoesNotExist()`: 断言某个参数不存在；
  - `status().is(200)`: 断言 HTTP 状态码为某个特定的值；
  - `redirectedUrl("http://localhost:8080/login")`: 断言重定向的网址。
  - `content().string()` 等：断言响应内容。
- 使用这些方法，我们就**不需要**再调用 `assertXxx()` 方法了。

# Spring Security 的对应

- 在使用 Spring Security 的项目中，构筑请求时还需设置一些和已登录用户相关的信息：

```
1 UserDetails alice = User.withDefaultPasswordEncoder( )
2     .username( "Alice" )
3     .password( "123456" )
4     .roles( "USER" )
5     .build( );
6
7 RequestBuilder request = MockMvcRequestBuilders
8     .get( "/hello" )
9     .with(csrf( ))
10    .with(user(alice));
```



# 解除外部依赖

- 在我们测试控制器的过程中，Controller 类调用了一些 Service 类的方法，这些方法又调用了一些其他类的方法。单元测试的原则是只测试 **1** 个类，而我们实际进行的其实是一种**集成测试**。
- 为了进行单元测试，我们可以使用一个**桩对象**[スタブ]模拟 Service 类的行为。我们可以定义 Service 类的方法在接收到某些参数时直接返回一些固定的返回值，而不调用真实的 Service 类中的代码。
- 要定义桩对象，首先在测试类中声明一个要替换的类的对象，并在其声明前添加 **@MockBean** 注解：

```
@MockBean  
private AccountService accountService;
```

# 桩对象动作的定义

- 使用 **when()** 方法定义桩对象在其各个方法被调用时的行为：

```
1 @BeforeEach
2 public void prepareData() {
3     when(accountService.createAccount(any(), any())).thenReturn(true);
4     when(accountService.createAccount(eq("Alice"), any())).thenReturn(false);
5     when(accountService.validateAccount(any(), any())).thenReturn(false);
6     when(accountService.validateAccount("Alice", "ABC12345")).thenReturn(true);
7 }
```

- 例如，上面的代码表示 Controller 想通过 Service 类创建任意一个账号时都会返回 true，除非用户名是 Alice；当 Controller 想验证任何一个账号时都会返回 false，除非用户名、密码分别为 Alice、ABC12345。
- 定义好桩对象后直接进行测试，会发现原本编写的 Service 代码不会被调用，只会直接返回桩对象定义的结果。

Q & A

*Question and answer*



## Coffee ☕ Break

## 测试方法的起名风格

单元测试方法的起名方式有很多种。共同点在于，方法名以大写字母开头，或以 **test + 大写字母开头**；方法名的各个部分都是大写驼峰形，用下划线“\_”分隔。

一种较为常见、可读性较高的起名方式如下：测试方法名 = 被测试方法名\_测试情况\_预期结果。比如：

```
testGetHelloPage_DoesNotLogin_Redirected( )
```

该方法测试了 `getHelloPage()` 这一方法在用户没有登录时的表现，预期的结果是请求被重定向到另一个网址。

# 总结

## Sum Up

1. 开发测试的概念：
  - ① 单元、集成、系统测试；
  - ② 白盒、灰盒、黑盒测试；
  - ③ 覆盖率。
  
2. 在 Java / Spring 中进行测试的方法：
  - ① 使用 JUnit 进行单元测试：assertXxx() 方法。
  
  - ② Controller 的测试方法：
    - a) MockMvc、
    - b) MockBean。



**THANK YOU!**