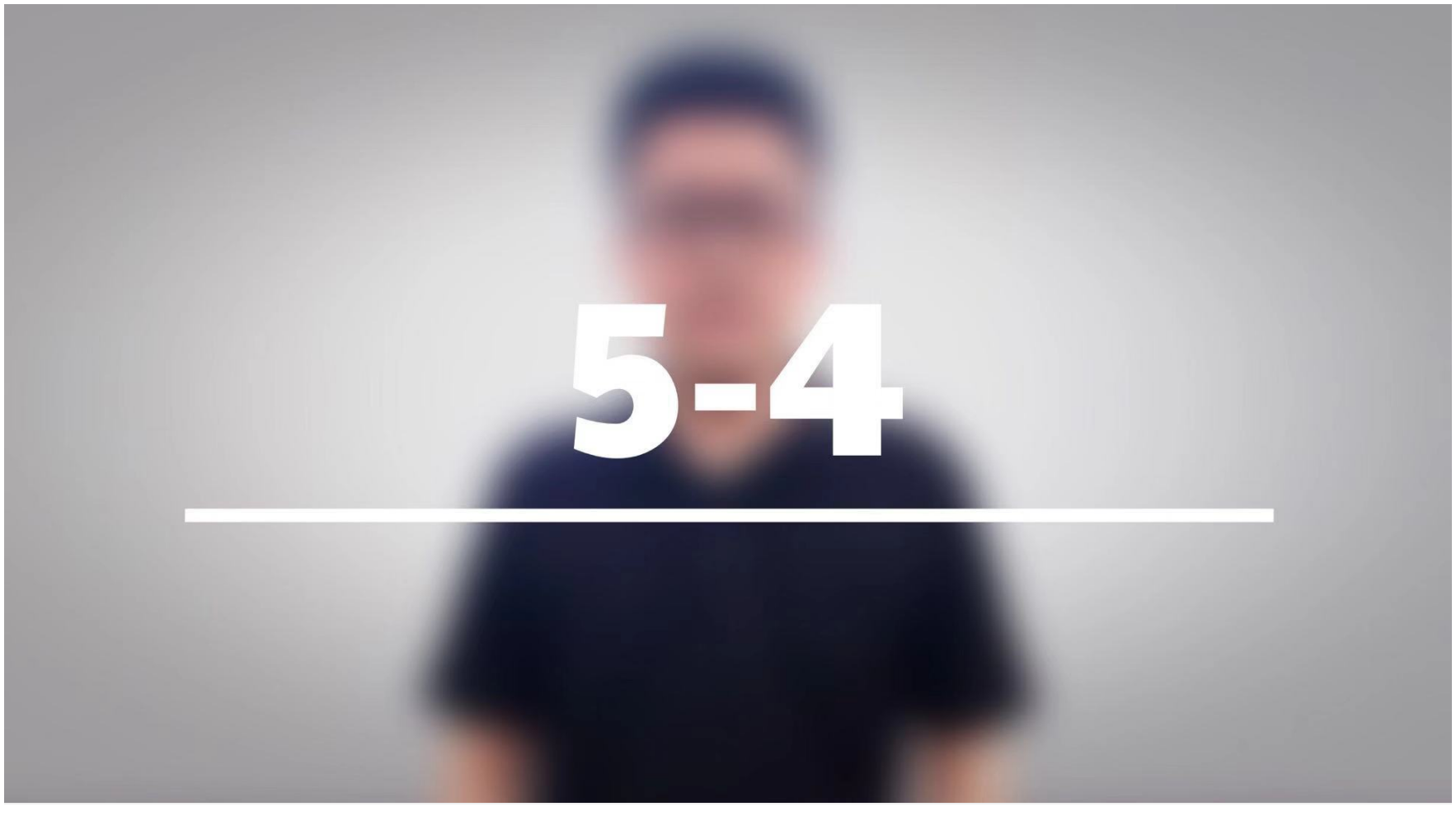

第 5-4 课： Spring Boot 对测试的支持

在微服务架构下，整个系统被切割为 N 个独立的微服务相互配合来使用，那么对于系统可用性会有更高的要求。从大到小可以分为三个层级，开发人员编码需要做的单元测试、微服务和微服务之间的接口联调测试、微服务和微服务之间的集成测试，通过三层的严格测试才能有效保证系统的稳定性。

作为一名开发人员，严格做好代码的单元测试才是保证软件质量的第一步。Spring Boot 做为一个优秀的开源框架合集对测试的支持非常友好，Spring Boot 提供了专门支持测试的组件 Spring Boot Test，其集成了业内流行的 7 种强大的测试框架：

- JUnit，一个 Java 语言的单元测试框架；
- Spring Test，为 Spring Boot 应用提供集成测试和工具支持；
- AssertJ，支持流式断言的 Java 测试框架；
- Hamcrest，一个匹配器库；
- Mockito，一个 Java Mock 框架；
- JSONassert，一个针对 JSON 的断言库；
- JsonPath，JSON XPath 库。

这 7 种测试框架完整的支持了软件开发中各种场景，我们只需要在项目中集成 Spring Boot Test 即可拥有这 7 种测试框架的各种功能，并且 Spring 针对 Spring Boot 项目使用场景进行了封装和优化，以方便在 Spring Boot 项目中去使用，接下来介绍 Spring Boot Test 的使用。

快速入手

我们创建一个 spring-boot-test 项目来演示 Spring Boot Test 的使用，只需要在项目中添加 spring-boot-starter-test 依赖即可：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

测试方法

首先来演示一个最简单的测试，只是测试一个方法的执行：

```
public class HelloTest {
    @Test
    public void hello() {
        System.out.println("hello world");
    }
}
```

在 Idea 中点击 helle() 方法名，选择 Run hello() 即可运行，执行完毕控制台打印信息如下：

```
hello world
```

证明方法执行成功。

测试服务

大多数情况下都是需要测试项目中某一个服务的准确性，这个时候往往需要 Spring Boot 启动后的上下文环境，对于这种情况只需要添加两个注解即可支持。我们创建一个 HelloService 服务来演示。

```
public interface HelloService {
    public void sayHello();
}
```

创建一个它的实现类:

```
@Service
public class HelloServieImpl implements HelloService {
    @Override
    public void sayHello() {
        System.out.println("hello service");
    }
}
```

在这个实现类中 sayHello() 方法输出了字符串："hello service"。

为了可以在测试中获取到启动后的上下文环境（Beans），Spring Boot Test 提供了两个注解来支持，测试时只需在测试类的上面添加 @RunWith(SpringRunner.class) 和 @SpringBootTest 注解即可。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloServiceTest {
    @Resource
    HelloService helloService;

    @Test
    public void  sayHelloTest(){
        helloService.sayHello();
    }
}
```

同时在测试类中注入 HelloService，sayHelloTest 测试方法中调用 HelloService 的 sayHello() 方法，执行测试方法后，就会发现在控制台打印出了 Spring Boot 的启动信息，说明在执行测试方法之前，Spring Boot 对容器进行了初始化，输出完启动信息后会打印出以下信息:

```
hello service
```

证明测试服务成功，但是这种测试会稍显麻烦，因为控制台打印了太多的东西，需要我们来仔细分辨，这里有更优雅的解决方案，可以利用 OutputCapture 来判断 System 是否输出了我们想要的内容，添加 OutputCapture 改造如下。

```
import static org.assertj.core.api.Assertions.assertThat;
import org.springframework.boot.test.rule.OutputCapture;

@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloServiceTest {
    @Rule
    public OutputCapture outputCapture = new OutputCapture();
    @Resource
    HelloService helloService;
    @Test
    public void  sayHelloTest(){
        helloService.sayHello();
        assertThat(this.outputCapture.toString().contains("hel-
lo service")).isTrue();
    }
}
```

OutputCapture 是 Spring Boot 提供的一个测试类，它能捕获 System.out 和 System.err 的输出，我们可以利用这个特性来判断程序中的输出是否执行。

这样当输出内容若是 "hello service"，则测试用例执行成功；若不是，则会执行失败，再也无需关注控制台输出内容。

Web 测试

据统计现在开发的 Java 项目中 90% 以上都是 Web 项目，如何检验 Web 项目对外提供接口的准确性就变得很重要。在以往的经历中，我们常常会在浏览器中访问一些特定的地址来进行测试，但如果涉及到一些非 get 请求就会变的稍微麻烦一些，有的读者会使用 PostMan 工具或者自己写一些 HTTP Post 请求来进行测试，但终究不够优雅方便。

Spring Boot Test 中有针对 Web 测试的解决方案：MockMvc，其实现了对 HTTP 请求的模拟，能够直接使用网络的形式，转换到 Controller 的调用，这样可以使得测试速度更快、不依赖网络环境，而且提供了一套验证的工具，这样可以使得请求的验证统一而且更方便。

接下来进行演示，首先在项目中添加 Web 依赖：

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

创建一个 HelloController 对外输出一个 hello 的方法。

```
@RestController
public class HelloController {
    @RequestMapping(name="/hello")
    public String getHello() {
        return "hello web";
    }
}
```

创建 HelloWebTest 对我们上面创建的 web 接口 getHello() 方法进行测试。

```
import static org.springframework.test.web.servlet.result.MockMvc-
ResultHandlers.print;
@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloWebTest {
    private MockMvc mockMvc;
    @Before
    public void setUp() throws Exception {
        mockMvc = MockMvcBuilders.standaloneSetup(new HelloCon-
troller()).build();
    }
    @Test
    public void testHello() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post("/hello")
                .accept(MediaType.APPLICATION_JSON_UTF8)).and-
Do(print());
    }
}
```

- @Before 注意意味着在测试用例执行前需要执行的操作，这里是初始化需要建立的测试环境。
- MockMvcRequestBuilders.post 是指支持 post 请求，这里其实可以支持各种类型的请求，如 get 请求、put 请求、patch 请求、delete 请求等。
- andDo(print())、andDo()：添加 ResultHandler 结果处理器，print() 打印出请求和相应的内容。

控制台输出：

```
MockHttpServletRequest:
      HTTP Method = POST
      Request URI = /hello
       Parameters = {}
          Headers = {Accept=[application/json;charset=UTF-8]}
             Body = <no character encoding set>
    Session Attrs = {}

Handler:
             Type = com.neo.web.HelloController
           Method = public java.lang.String com.neo.web.HelloCon-
troller.getUser()

Async:
    Async started = false
     Async result = null

Resolved Exception:
             Type = null

ModelAndView:
        View name = null
             View = null
            Model = null

FlashMap:
       Attributes = null

MockHttpServletResponse:
           Status = 200
    Error message = null
          Headers = {Content-Type=[application/json;charset=UTF-8],
 Content-Length=[9]}
     Content type = application/json;charset=UTF-8
             Body = hello web
    Forwarded URL = null
   Redirected URL = null
          Cookies = []
```

通过上面输出的信息会发现，将整个请求的过程全部打印了出来，包括请求头信息、请求参数、返回信息等，根据打印的 Body 信息可以得知 HelloController 的 getHello() 方法测试成功。

但有时候我们并不想知道整个请求流程，只需要验证返回的结果是否正确即可，可以做下面的改造：

```
@Test
public void testHello() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.post("/hello")
            .accept(MediaType.APPLICATION_JSON_UTF8))
//                  .andDo(print())
             .andExpect(content().string(equalTo("hello web")));
}
```

如果接口返回值是 "hello web" 测试执行成功，否则测试用例执行失败。也支持验证结果集中是否包含了特定的字符串，这时可以使用 containsString() 方法来判断。

```
.andExpect(content().string(containsString("hello")));;
```

支持直接将结果集转换为字符串输出:

```
String mvcResult= mockMvc.perform(MockMvcRequestBuilders.get("/mes-
sages")).andReturn().getResponse().getContentAsString();
System.out.println("Result === "+mvcResult);
```

支持在请求的时候传递参数:

```
@Test
public void testHelloMore() throws Exception {
    final MultiValueMap<String, String> params = new LinkedMulti-
ValueMap<>();
    params.add("id", "6");
    params.add("hello", "world");
    mockMvc.perform(
             MockMvcRequestBuilders.post("/hello")
            .params(params)
            .contentType(MediaType.APPLICATION_JSON_UTF8)
            .accept(MediaType.APPLICATION_JSON_UTF8))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("hello")));;
}
```

返回结果如果是 JSON 可以使用下面语法来判断:

```
.andExpect(MockMvcResultMatchers.jsonPath("$.name").value("纯洁的微
笑"))
```

MockMvc 提供一组工具函数用来执行 Assert 判断，这组工具使用函数的链式调用，允许将多个测试用例拼接在一起，同时进行多个判断。

- perform 构建一个请求，并且返回 ResultActions 实例，该实例则可以获取到请求的返回内容。
- params 构建请求时候的参数，也支持 param(key,value) 的方式连续添加。
- contentType(MediaType.APPLICATION_JSON_UTF8) 代表发送端发送的数据格式。
- accept(MediaType.APPLICATION_JSON_UTF8) 代表客户端希望接受的数据类型格式。
- mockMvc.perform() 建立 Web 请求。
- andExpect(...) 可以在 perform(...) 函数调用后多次调用，表示对多个条件的判断。
- status().isOk() 判断请求状态是否返回 200。
- andReturn 该方法返回 MvcResult 对象，该对象可以获取到返回的视图名称、返回的 Response 状态、获取拦截请求的拦截器集合等。

JUnit 使用

JUnit 是针对 Java 语言的一个单元测试框架，它被认为是迄今为止所开发的最重要的第三方 Java 库。 JUnit 的优点是整个测试过程无需人的参与、无需分析和判断最终测试结果是否正确，而且可以很容易地一次性运行多个测试。 JUnit 的最新版本为 Junit 5，Spring Boot 默认集成的是 Junit 4。

以下为 Junit 常用注解：

- @Test，把一个方法标记为测试方法
- @Before，每一个测试方法执行前自动调用一次
- @After，每一个测试方法执行完自动调用一次
- @BeforeClass，所有测试方法执行前执行一次，在测试类还没有实例化就已经被加载，因此用 static 修饰

- @AfterClass，所有测试方法执行前执行一次，在测试类还没有实例化就已经被加载，因此用 static 修饰
- @Ignore，暂不执行该测试方法
- @RunWith 当一个类用 @RunWith 注释或继承一个用 @RunWith 注释的类时，JUnit 将调用它所引用的类来运行该类中的测试而不是开发者再去 JUnit 内部去构建它。我们在开发过程中使用这个特性看看。

创建测试类 JUnit4Test 类:

```
public class JUnit4Test {
    Calculation calculation = new Calculation();
    int result;     //测试结果

    //在 JUnit 4 中使用 @Test 标注为测试方法
    @Test
    //测试方法必须是 public void 的
    public void testAdd() {
        System.out.println("---testAdd开始测试---");

        //每个里面只测一次，因为 assertE-
quals 一旦测试发现错误就抛出异常，不再运行后续代码
        result = calculation.add(1, 2);
        assertEquals(3, result);

        System.out.println("---testAdd正常运行结束---");
    }

    //又一个测试方法
    //timeout 表示测试允许的执行时间毫秒数，expect-
ed 表示忽略哪些抛出的异常（不会因为该异常导致测试不通过）
    @Test(timeout = 1, expected = NullPointerException.class)
    public void testSub() {
        System.out.println("---testSub开始测试---");

        result = calculation.sub(3, 2);
        assertEquals(1, result);

        throw new NullPointerException();

        //System.out.println("---testSub正常运行结束---");
    }


    //指示该[静态方法]将在该类的[所有]测试方法执行之[前]执行
    @BeforeClass
```

```
public static void beforeAll() {
    System.out.println("||==BeforeClass==||");
    System.out.println("||==通常在这个方法中加载资源==||");
}

//指示该[静态方法]将在该类的[所有]测试方法执行之[后]执行
@AfterClass
public static void afterAll() {
    System.out.println("||==AfterClass==||");
    System.out.println("||==通常在这个方法中释放资源==||");
}

//该[成员方法]在[每个]测试方法执行之[前]执行
@Before
public void beforeEvery() {
    System.out.println("|==Before==|");
}

//该[成员方法]在[每个]测试方法执行之[后]执行
@After
public void afterEvery() {
    System.out.println("|==After==|");
}
```

calculation 是自定义的计算器工具类，具体可以参考示例项目，执行测试类后，输出：

```
||==BeforeClass==||
||==通常在这个方法中加载资源==||
|==Before==|
---testAdd开始测试---
---testAdd正常运行结束---
|==After==|
|==Before==|
---testSub开始测试---
|==After==|
||==AfterClass==||
||==通常在这个方法中释放资源==||
```

对比上面的介绍可以清晰的了解每个注解的使用。

Assert 使用

Assert 翻译为中文为“断言”，使用过 JUnit 的读者都熟知这个概念，它断定某一个实际的运行值和预期想一样，否则就抛出异常 。Spring 对方法入参的检测借用了这个概念，其提供的 Assert 类拥有众多按规则对方法入参进行断言的方法，可以满足大部分方法入参检测的要求。

Spring Boot 也提供了断言式的验证，帮助我们在测试时验证方法的返回结果。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class TestAssert {
    @Autowired
    private UserService userService;
    @Test
    public void TestAssert(){
        //验证结果是否为空
        Assert.assertNotNull(userService.getUser());
        //验证结果是否相等
        Assert.assertEquals("i am neo!", userService.getUser());
        //验证条件是否成立
        Assert.assertFalse(1+1>3);
        //验证对象是否相等
        Assert.assertSame(userService,userService);
        int status=404;
        //验证结果集，提示
        Assert.assertFalse("错误，正确的返回值为200", status != 200);
        String[] expectedOutput = {"apple", "mango", "grape"};
        String[] methodOutput = {"apple", "mango", "grape1"};
        //验证数组是否相同
        Assert.assertArrayEquals(expectedOutput, methodOutput);
    }
}
```

通过上面使用的例子可以发现，使用 Assert 可以非常方便验证测试返回结果，避免写很多的 if/else 判断，让代码更加的优雅。

如何使用 assertThat

JUnit 4 学习 JMock，引入了 Hamcrest 匹配机制，使得程序员在编写单元测试的 assert 语句时，可以具有更强的可读性，而且也更加灵活。Hamcrest 是一个测试的框架，它提供了一套通用的匹配符 Matcher，灵活使用这些匹配符定义的规则，程序员可以更加精确的表达自己的测试思想，指定所想设定的测试条件。

断言便是 Junit 中最长使用的语法之一，在文章内容开始使用了 assertThat 对 System 输出的文本进行了判断，assertThat 其实是 JUnit 4 最新的语法糖，只使用 assertThat 一个断言语句，结合 Hamcrest 提供的匹配符，就可以替代之前所有断言的使用方式。

assertThat 的基本语法如下：

```
assertThat( [value], [matcher statement] );
```

- value 是接下来想要测试的变量值；
- matcher statement 是使用 Hamcrest 匹配符来表达对前面变量所期望值的声明，如果 value 值与 matcher statement 所表达的期望值相符，则测试成功，否则测试失败。

一般匹配符

```
// allOf 匹配符表明如果接下来的所有条件必须都成立测试才通过，相当于“与”（&&）
assertThat( testedNumber, allOf( greaterThan(8), lessThan(16) ) );
// anyOf 匹配符表明如果接下来的所有条件只要有一个成立则测试通过，相当于“或”（||）
assertThat( testedNumber, anyOf( greaterThan(16), lessThan(8) ) );
// anything 匹配符表明无论什么条件，永远为 true
assertThat( testedNumber, anything() );
// is 匹配符表明如果前面待测的 object 等于后面给出的 object，则测试通过
assertThat( testedString, is( "developerWorks" ) );
// not 匹配符和 is 匹配符正好相反，表明如果前面待测的 object 不等于后面给出的 object，则测试通过
assertThat( testedString, not( "developerWorks" ) );
```

字符串相关匹配符

```
// containsString 匹配符表明如果测试的字符串 tested-
String 包含子字符串"developerWorks"则测试通过
assertThat( testedString, containsString( "developerWorks" ) );
// endsWith 匹配符表明如果测试的字符串 testedString 以子字符串"developer-
Works"结尾则测试通过
assertThat( testedString, endsWith( "developerWorks" ) );
// startsWith 匹配符表明如果测试的字符串 testedString 以子字符串"develop-
erWorks"开始则测试通过
assertThat( testedString, startsWith( "developerWorks" ) );
// equalTo 匹配符表明如果测试的 testedValue 等于 expectedVal-
ue 则测试通过，equalTo 可以测试数值之间的字
//符串之间和对象之间是否相等，相当于 Object 的 equals 方法
assertThat( testedValue, equalTo( expectedValue ) );
// equalToIgnoringCase 匹配符表明如果测试的字符串 tested-
String 在忽略大小写的情况下等于
//"developerWorks"则测试通过
assertThat( testedString, equalToIgnoringCase( "developerWorks" ) )
;
// equalToIgnoringWhiteSpace 匹配符表明如果测试的字符串 tested-
String 在忽略头尾的任意个空格的情况下等
//于"developerWorks"则测试通过，注意，字符串中的空格不能被忽略
assertThat( testedString, equalToIgnoringWhiteSpace( "developer-
Works" ) );
```

数值相关匹配符

```
// closeTo 匹配符表明如果所测试的浮点型数 testedDou-
ble 在 20.0±0.5 范围之内则测试通过
assertThat( testedDouble, closeTo( 20.0, 0.5 ) );
// greaterThan 匹配符表明如果所测试的数值 testedNum-
ber 大于 16.0 则测试通过
assertThat( testedNumber, greaterThan(16.0) );
// lessThan 匹配符表明如果所测试的数值 testedNumber 小于 16.0 则测试通过
assertThat( testedNumber, lessThan (16.0) );
// greaterThanOrEqualTo 匹配符表明如果所测试的数值 testedNum-
ber 大于等于 16.0 则测试通过
assertThat( testedNumber, greaterThanOrEqualTo (16.0) );
// lessThanOrEqualTo 匹配符表明如果所测试的数值 testedNum-
ber 小于等于 16.0 则测试通过
assertThat( testedNumber, lessThanOrEqualTo (16.0) );
```

collection 相关匹配符

```
// hasEntry 匹配符表明如果测试的 Map 对象 mapOb-
ject 含有一个键值为"key"对应元素值为"value"的 Entry 项则
//测试通过
assertThat( mapObject, hasEntry( "key", "value" ) );
// hasItem 匹配符表明如果测试的迭代对象 iterableObject 含有元素“ele-
ment”项则测试通过
assertThat( iterableObject, hasItem ( "element" ) );
// hasKey 匹配符表明如果测试的 Map 对象 mapObject 含有键值“key”则测试通过
assertThat( mapObject, hasKey ( "key" ) );
// hasValue 匹配符表明如果测试的 Map 对象 mapObject 含有元素值“val-
ue”则测试通过
assertThat( mapObject, hasValue ( "key" ) );
```

具体使用可参考示例项目中 CalculationTest 的使用。

Junt 使用的几条建议：

- 测试方法上必须使用 @Test 进行修饰
- 测试方法必须使用 public void 进行修饰，不能带任何的参数
- 新建一个源代码目录来存放我们的测试代码，即将测试代码和项目业务代码分开
- 测试类所在的包名应该和被测试类所在的包名保持一致
- 测试单元中的每个方法必须可以独立测试，测试方法间不能有任何的依赖
- 测试类使用 Test 作为类名的后缀（不是必须）
- 测试方法使用 Test 作为方法名的前缀（不是必须）

总结

Spring Boot 是一款自带测试组件的开源软件，Spring Boot Test 中内置了 7 种强大的测试工具，覆盖了测试中的方方面面，在实际应用中只需要导入 Spring Boot Test 既可让项目具备各种测试功能。在微服务架构下严格采用三层测试覆盖，才能有效保证项目质量。

> 点击这里下载源码 (https://github.com/ityouknow/spring-boot-lean - ing/tree/gitbook_column2.0)。

(/gitchat/column/5b86228ce15aa17d68b5b55a/topic/5c25a1fa4fcd483b0265b262)(/gitcha