

## 1-2、Mockito（模拟数据，拦截，断言）辅助MockMvc进行单元测试

### （0）参考

- <https://www.javadoc.io/doc/org.mockito/mockito-core/3.0.0/org/mockito/Mockito.html>
- <https://github.com/mockito/mockito>
- <https://www.baeldung.com/mockito-series>
- <https://www.javadoc.io/doc/org.mockito>
- <https://www.baeldung.com/mockito-annotations>



The screenshot shows a web browser displaying the Mockito artifacts page on javadoc.io. The page title is 'org.mockito'. Below the title, it states 'group org.mockito has published 38 artifact(s) with total 2122 version(s)'. A table lists the artifacts, the number of versions published, and the latest version.

Artifact	Num# versions published	Latest version
junit-jupiter	1	2.20.0
mockito-all	43	2.0.2-beta
mockito-android	117	5.2.0
mockito-core	323	5.2.0

←

→

↻

🔒 javadoc.io/doc/org.mockito/mockito-core/latest/index.html

hao123

🔄 淘宝

🔄 天猫

JD 京东

🐾 百度一下

🔴 (88条消息) 使用As...

🔴 (96条消息) 在kotli...

🌈 50张简约头像 | 总...

🐾 Lo

🏠

org.mockito

mockito-core ▾

5.2.0 ▾

📄

[Skip navigation links](#)

Overview

Package

Class

Use

[Tree](#)

[Deprecated](#)

[Index](#)

[Help](#)

•

SEARCH:

Click to see examples.

Mockito 5.2.0 API.


All Packages

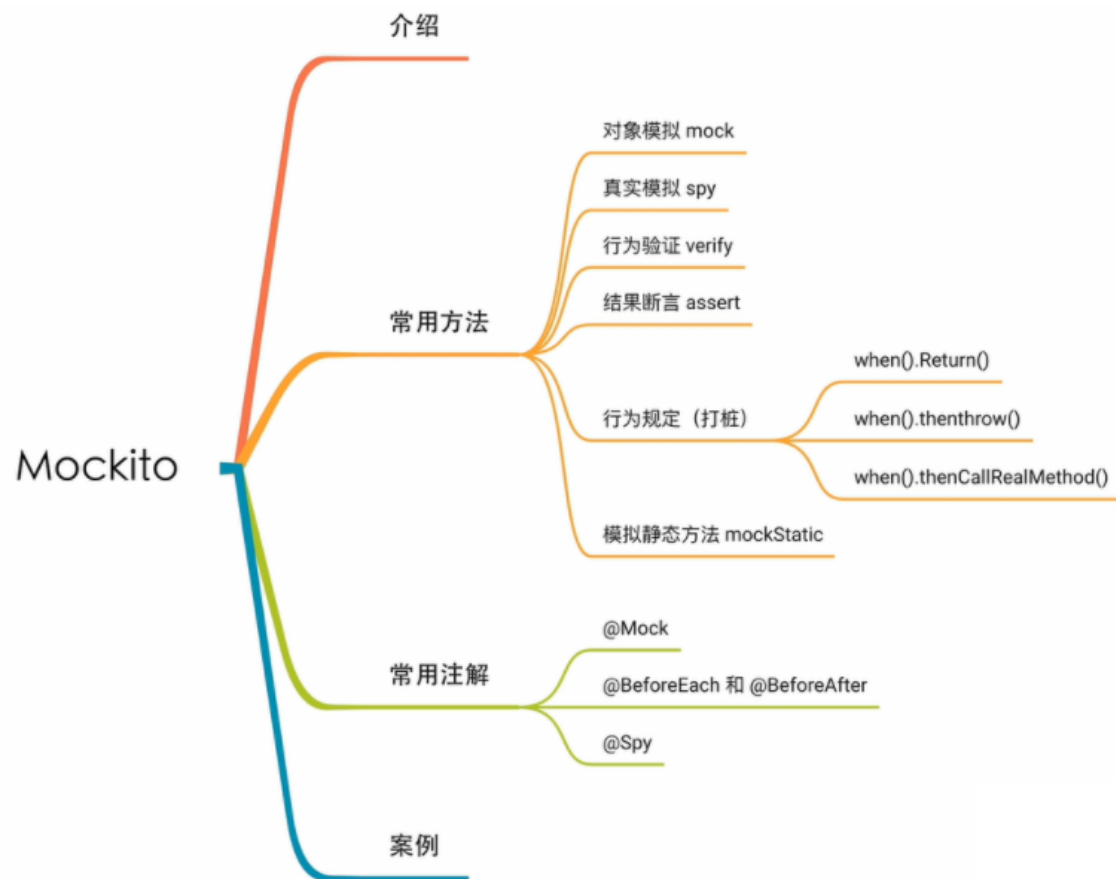
Main package

Other Packages

Package	
<b>org.mockito</b>	Mockito is a mock library for java - see <b>Mockito</b> class for usage.
<b>org.mockito.codegen</b>	
<b>org.mockito.configuration</b>	Mockito configuration utilities.
<b>org.mockito.hamcrest</b>	

<a href="#">MockedConstruction.Context</a>	The context for a construction mock.
<a href="#">MockedConstruction.MockInitializer&lt;T&gt;</a>	Functional interface that consumes a newly created mock and the mock con
<a href="#">MockedStatic&lt;T&gt;</a>	Represents an active mock of a type's static methods.
<a href="#">MockedStatic.Verification</a>	Functional interface for a verification operation on a static mock.
<a href="#">MockingDetails</a>	Provides mocking information.
<a href="#">MockitoFramework</a>	Mockito framework settings and lifecycle listeners, for advanced users or for
<a href="#">MockitoSession</a>	<code>MockitoSession</code> is an optional, highly recommended feature that drives wri
<a href="#">MockSettings</a>	Allows mock creation with additional mock settings.
<a href="#">ScopedMock</a>	Represents a mock with a thread-local explicit scope.

Class Summary	
Class	Description
<a href="#">AdditionalAnswers</a>	Additional answers provides factory methods for answers.
<a href="#">AdditionalMatchers</a>	See <code>ArgumentMatchers</code> for general info about matchers.
<a href="#">ArgumentCaptor&lt;T&gt;</a>	Use it to capture argument values for further assertions.
<a href="#">ArgumentMatchers</a>	Allow flexible verification or stubbing.
<a href="#">BDDMockito</a>	Behavior Driven Development style of writing tests uses <code>//given</code> <code>//when</code> <code>//then</code> comments as fundamental parts
<a href="#">Mockito</a>	
<a href="#">MockitoAnnotations</a>	<code>MockitoAnnotations.openMocks(this)</code> ; initializes fields annotated with Mockito annotations.
<a href="#">MockMakers</a>	Constants for built-in implementations of <code>MockMaker</code> .



## (1) 启用模拟注解

- 第一步首先就是要启用这些模拟注解，不然当我们尝试实际使用带有@Mock、@Spy、@MockBean注解的实例对象时，我们可能会遇到NullPointerException

复制代码

```
1
2 // 启用Mockito的相关注解可用，从junit4的
  @RunWith(MockitoJUnitRunner.class)变成junit5的
  @ExtendWith(MockitoExtension.class)。
3 @RunWith(MockitoJUnitRunner.class)
```

```

4 public class MockitoAnnotationTest {
5 }
6
7 //或者，我们可以通过直接调用Mockito的MockitoAnnotations.openMocks (以编程方式激活启用Mockito注释。搭配junit4的@Before
8 @Before public void init() {
9     MockitoAnnotations.openMocks(this);
10 }
11
12 //junit4到junit5的注解变化
13 @BeforeClass 变为 @BeforeAll
14 public void beforeAll()
15 @AfterClass 变为 @AfterAll
16 public void afterAll()
17 @Before 变为 @BeforeEach
18 public void beforeEach()
19 @After 变为 @AfterEach
20 public void afterEach()
21 @Test从junit4的org.junit.Test变为junit5的
    org.junit.jupiter.api.Test
22 public void test(){
23 }

```

java >

## (2) @Mock注解

- 先看下不使用@Mock来创建模拟实例，而是手动调用方法的方式来创建模拟实例，MockitoAnnotations.initMocks(this)来启用模拟实例

```

1 public class MockDemo {
2

```

复制代码

```

3  @Before
4  public void before() {
5      MockitoAnnotations.initMocks(this);
6  }
7
8  @Test
9  public void notUseMockAnnotation() {
10     //①mock方法: 来模拟一个对象
11     List mockList = Mockito.mock(ArrayList.class);
12     DemoService demoService = Mockito.mock(DemoService.class);
13
14     //②mock方法: 来模拟一个对象
15     private DemoService demoService;
16     @Before
17     public void before() {
18         demoService = mock(DemoService.class);
19     }
20 }
21 }

```

java >

- 使用@Mock来创建模拟实例，MockitoAnnotations.initMocks(this)来启用模拟实例

复制代码

```

1  public class MockDemo {
2      @Mock
3      private DemoService demoService;
4
5      @Before
6      public void before() {
7          MockitoAnnotations.initMocks(this);
8      }

```

```
9 }
```

java >

- 使用@Mock来创建模拟实例，@RunWith(MockitoJUnitRunner.class)来启用模拟实例

复制代码

```
1 @RunWith(MockitoJUnitRunner.class)
2 public class MockDemo {
3     @Mock
4     private DemoService demoService;
5 }
```

java >

### (3) @Spy注解

和@Mock注解的区别：主要区别在于调用的实例是否是真实的。

@Mock修饰的实例是模拟的实例，所以调用的方法也不会调用真实的方法，也是模拟出来的方法，包括方法参数和返回值，有返回值的，默认是返回null，除非模拟了返回值，而如果想要正常获取到返回值，需要模拟的方法参数在程序中调用时是一致的才能获取到模拟的返回值，如果方法没有参数，则直接返回模拟的返回值。

@Spy修饰的实例是真实的实例，所以调用的方法都是真实的方法，有返回值的返回真实的值，除非模拟了返回值，而如果想要正常获取到模拟返回值，需要方法的参数是预期的参数，如果方法没有参数，则直接返回模拟的返回值。

相同点：@Mock的实例和@Spy的实例都用不到spring上下文环境，即不受spring容器管理。

- 简单案例

复制代码

```
1 //使用@Spy的方式
```

```

2 @Spy
3 List<String> spiedList = new ArrayList<String>();
4
5 @Test
6 public void whenNotUseSpyAnnotation_thenCorrect() {
7     //真实实例（不使用@Spy，通过调用spy方法的方式）
8     List<String> spyList = Mockito.spy(new ArrayList<String>());
9
10    //真实实例中调用真实方法
11    spyList.add("one");
12    spyList.add("two");
13
14    //用来验证真实实例的真实方法（add方法）是否被调用了，下面这两行能通过说明上面那两行确实被调用了
15    Mockito.verify(spyList).add("one");
16    Mockito.verify(spyList).add("two");
17
18    //断言：真实的返回值
19    assertEquals(2, spyList.size());
20
21    //调用spyList.size()方法时是返回的真实值2，但是这里模拟了返回值是100，所以返回100
22    Mockito.doReturn(100).when(spyList).size();
23    //断言：模拟的返回值
24    assertEquals(100, spyList.size());
25 }

```

java >

#### (4) @Captor注释



用于创建一个ArgumentCaptor实例，并且声明了哪种方法的参数类型是要拦截的，通过方法capture()进行拦截参数值，通过getValue/getAllValues来获取参数值，用来进一步验证。

一般是调用完方法后，才会使用这个进行拦截。

一般是Mockito的verify验证方法和ArgumentCaptor的capture()方法搭配使用。

- 不使用@Captor注解的方式来拦截方法参数，而是使用ArgumentCaptor.forClass(XXX.class)方式来创建ArgumentCaptor实例

复制代码

```
1  @Test
2  public void whenNotUseCaptorAnnotation_thenCorrect() {
3      //模拟一个List<String>集合实例
4      List mockList = Mockito.mock(List.class);
5
6      //调用集合实例的add方法添加数据（方法参数是String类型）
7      mockList.add("one");
8
9      //准备一个ArgumentCaptor实例，用来拦截方法参数类型是String类型的数据
10     ArgumentCaptor<String> arg =
        ArgumentCaptor.forClass(String.class);
11
12     //verify: 验证集合实例的add方法是否被调用了，验证默认调用次数1，而且每
        次调用方法时的参数值，都进行验证。每次调用add方法时的参数值是通过
        arg.capture()方法拦截保存到arg内部的变量中。
13     Mockito.verify(mockList).add(arg.capture());
14
15     //通过arg的getValue()方法来获取拦截到的参数值，也就是调用add时的参数值
16     assertEquals("one", arg.getValue());
17 }
```

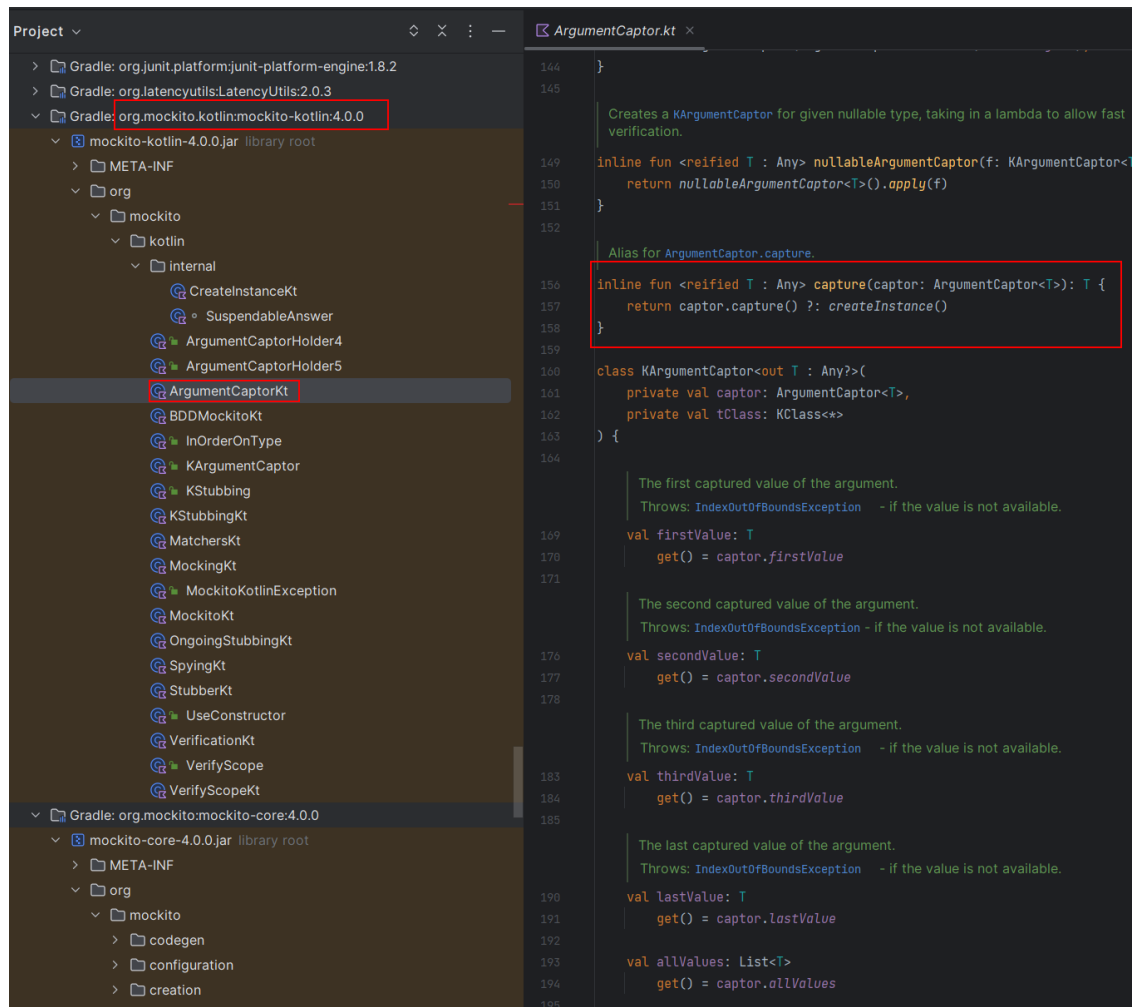
```
18
19
20 @Test
21 public void test(){
22     //模拟一个List<String>集合实例
23     List mockList = Mockito.mock(List.class);
24     //调用集合实例的add方法添加数据（方法参数是String类型）
25     mockList.add("one");
26     mockList.add("two");
27
28     //准备一个ArgumentCaptor实例，用来拦截方法参数类型是String类型的数据
29     ArgumentCaptor<String> argCaptor =
        ArgumentCaptor.forClass(String.class);
30
31     //verify: 验证集合实例的add方法是否被调用了，并且验证了调用次数，每次调用
        方法时的参数值，都进行了验证。每次调用add方法时的参数值是通过
        argCaptor.capture()拦截放到这里的。
32
        Mockito.verify(mockList, Mockito.times(2)).add(argCaptor.capture()
        );
33
34     //断言：如果使用argCaptor.getValue() 获取拦截到的参数值，如果方法进行了
        多次调用，它将返回最后一个方法的参数值。argCaptor.getAllValues()用来获取拦
        截到的所有的参数值，返回的是一个集合，也就是每次调用add时的参数值。
35     Assertions.assertEquals("one",
        argCaptor.getAllValues().get(0));
36     Assertions.assertEquals("two",
        argCaptor.getAllValues().get(1));
37 }
```

- 使用ArgumentCaptor拦截参数时，使用ArgumentCaptor的capture(argCaptor)这个方法替换掉capture()方法返回值为空的问题

复制代码

```
1 testImplementation("org.mockito.kotlin:mockito-kotlin:4.0.0")
```

java &gt;



```
1
2 val argCaptor: ArgumentCaptor<ClockInRequest> =
  ArgumentCaptor.forClass(ClockInRequest::class.java)
3
4 Mockito.verify(clockInRpcService).saveClockIn(capture(argCaptor))
5 Mockito.verify(clockInRpcService,times(2)).saveClockIn(capture(ar
  gCaptor))
6 Mockito.verify(clockInRpcService,atLeast(2)).saveClockIn(capture(
  argCaptor
7 ))
8
9 Assertions.assertEquals(request.answerBy,argCaptor.value.patientI
  d)
10
11 Assertions.assertEquals(request.answerBy,argCaptor.allValues[0].p
  atientId)
12 Assertions.assertEquals(HealthPlanType.DIET_PLAN,argCaptor.allVal
  ues[0].healthPlanType)
13
14 Assertions.assertEquals(request.answerBy,argCaptor.allValues[1].p
  atientId)
15 Assertions.assertEquals(HealthPlanType.DIET_PLAN,argCaptor.allVal
  ues[1].healthPlanType)
16
17 Assertions.assertEquals(request.answerBy,argCaptor.firstValue.pat
  ientId)
18 Assertions.assertEquals(HealthPlanType.DIET_PLAN,argCaptor.firstV
  alue.healthPlanType)
19
20 Assertions.assertEquals(request.answerBy,argCaptor.secondValue.pa
  tientId)
```

```
21 Assertions.assertEquals(HealthPlanType.DIET_PLAN, argCaptor.second
    Value.healthPlanType)
```

java >

- 使用@Captor注解的方式来实现相同的目的，创建一个 ArgumentCaptor 实例

复制代码

```
1  @Mock
2  List mockedList;
3
4  @Captor
5  ArgumentCaptor argCaptor;
6
7  @Test
8  public void whenUseCaptorAnnotation_thenTheSame() {
9
10     mockedList.add("one");
11
12     Mockito.verify(mockedList).add(argCaptor.capture());
13
14     assertEquals("one", argCaptor.getValue());
15 }
16
17 //////////////////////////////////////
18
19 @Mock
20 HashMap<String,Integer> hashMap;
21
22 @Captor
23 ArgumentCaptor keyCaptor;
24
```

```

25 @Captor
26 ArgumentCaptor valueCaptor;
27
28 @Test
29 public void whenUseCaptorAnnotation_thenTheSame() {
30
31     hashMap.put("A", 10);
32
33
34
35     Mockito.verify(hashMap).put(keyCaptor.capture(), valueCaptor.capture());
36
37     assertEquals("A", keyCaptor.getValue());
38     assertEquals(Integer(10), valueCaptor.getValue());
39 }

```

java >

## ( 5 ) @InjectMocks注解

- @Mock修饰的模拟属性，会自动注入到@InjectMocks修饰的属性中

```

1 @RunWith(xxx)
2 class Test{
3
4     @Mock
5     Map<String, String> wordMap;
6
7     @InjectMocks
8     MyDictionary dic = new MyDictionary();
9

```

复制代码

```

10  @Test
11  public void whenUseInjectMocksAnnotation_thenCorrect() {
12
13      //模拟数据
14      Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning");
15
16      //调用时, 走的模拟数据
17      val value = String = dic.getMeaning("aWord")
18
19      assertEquals("aMeaning", value);
20  }
21 }
22
23 ///////////////////////////////////////////////////
24
25 public class MyDictionary {
26     Map<String, String> wordMap;
27
28     public MyDictionary() {
29         wordMap = new HashMap<String, String>();
30     }
31     public void add(final String word, final String meaning) {
32         wordMap.put(word, meaning);
33     }
34     public String getMeaning(final String word) {
35         return wordMap.get(word);
36     }
37 }

```

## (6) 类比5, 我们希望将模拟的属性注入到间谍属性中

- 但是Mockito 不支持将模拟注入到间谍中, 以下测试会导致异常

复制代码

```
1  @Mock
2  Map<String, String> wordMap;
3
4  @Spy
5  MyDictionary spyDic = new MyDictionary();
6
7  @Test
8  public void whenUseInjectMocksAnnotation_thenCorrect() {
9      Mockito.when(wordMap.get("aWord")).thenReturn("aMeaning")
10
11      String value = spyDic.getMeaning("aWord")
12
13      assertEquals("aMeaning", value)
14  }
15
16
17  //////////////////////////////////////
18
19  public class MyDictionary {
20      Map<String, String> wordMap;
21
22      public MyDictionary() {
23          wordMap = new HashMap<String, String>();
24      }
25      public void add(final String word, final String meaning) {
26          wordMap.put(word, meaning);
27      }
```



```

28     public String getMeaning(final String word) {
29         return wordMap.get(word);
30     }
31 }

```

java >

- 如果我们想将模拟与间谍一起使用，我们可以通过构造函数手动注入模拟对象，然后Mockito.spy手动创建间谍对象

复制代码

```

1  @Mock
2  Map<String, String> wordMap;
3
4  MyDictionary spyDic;
5
6  @Before
7  public void init() {
8      MockitoAnnotations.openMocks(this);
9      spyDic = Mockito.spy(new MyDictionary(wordMap));
10 }
11
12 ///////////////////////////////////////////////////
13
14 public class MyDictionary {
15     Map<String, String> wordMap;
16
17     MyDictionary(Map<String, String> wordMap) {
18         this.wordMap = wordMap;
19     }
20     public void add(final String word, final String meaning) {
21         wordMap.put(word, meaning);

```

```
22     }
23     public String getMeaning(final String word) {
24         return wordMap.get(word);
25     }
26 }
27
```

java >

## (7) @MockBean注解

和@Mock的区别

主要区别就是是否需要用到spring上下文环境（spring容器）

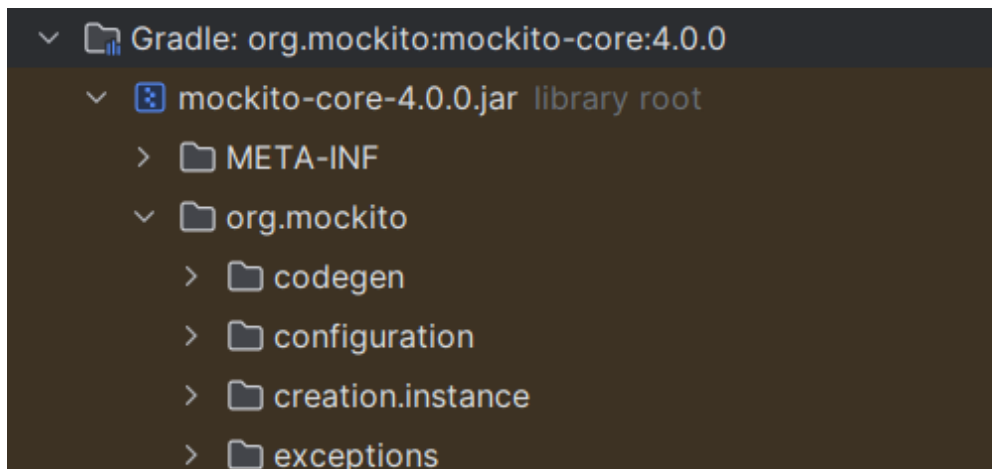
@Mock模拟的实例不需要用到spring上下文环境（不需要依赖框架的情况）即不受spring容器管理。

@MockBean模拟的实例需要放到spring上下文环境，受spring容器管理。

相同点：

调用实例的方法都不是真实的方法，有返回值的，默认返回null，除非模拟了返回值，而如果想要正常获取到返回值，需要方法的参数是预期的参数。

(7-1) @Mock是Mockito的包，如果是简单的，不依赖Spring Boot Container的测试，那么@Mock就够用了。







- > hamcrest
- > internal
- > invocation
- > junit
- > listeners
- > mock
- > plugins
- > quality
- > session
- > stubbing
- > verification
  - © AdditionalAnswers
  - © AdditionalMatchers
  - Ⓔ Answers
  - © ArgumentCaptor
  - Ⓘ ArgumentMatcher
  - © ArgumentMatchers
- > © BDDMockito
  - @ Captor
  - @ CheckReturnValue
  - @ Incubating
  - @ InjectMocks
  - Ⓘ InOrder
- Mock
- > Ⓘ MockedConstruction

## 笔记目录



- 1-2、Mockito (模拟数...
  - (0) 参考
  - (1) 启用模拟注解
  - (2) @Mock注解
  - (3) @Spy注解
  - (4) @Captor注释
  - (5) @InjectMocks...
  - (6) 类比5, 我们希...
  - (7) @MockBean注解
    - (7-1) @Mock是M...
    - (7-2) @MockBea...
  - (8) Mockito的基本API
    - (8-1) 基本API
    - (8-2) verify: 进...
    - (8-3) stubbing: ...
    - (8-4) extra flexibil...
    - (8-5-1) verify exa...

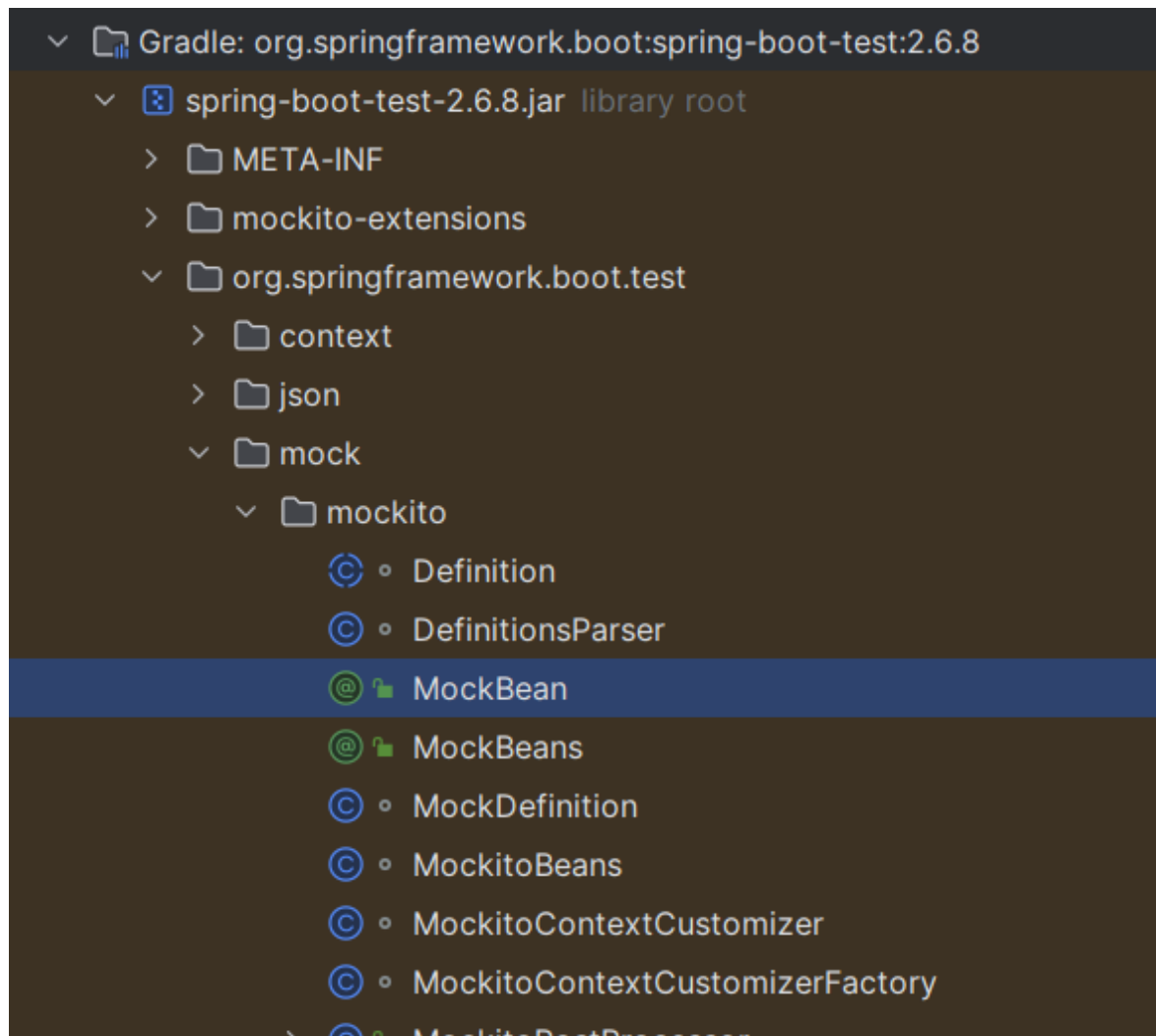
>   MockedStatic

  MockingDetails

• (8-5-2) Verificatio...

( 7-2 ) @MockBean是spring的包 ( spring-boot-test包, 算是对Mockito包的包装 );

- 如果需要依赖Spring Boot Container测试, 且需要添加或者mock一个container bean ( 模拟一个spring上下文环境 ), 那么就用@MockBean, 使用@MockBean修饰的属性会创建一个模拟bean, 注入到spring上下文, 即替换了Spring上下文中的Bean, 这样会导致Spring上下文重启。



> MockitoPostProcessor  
> MockitoTestExecutionListener  
> MockReset  
    QualifierDefinition  
    ResetMocksTestExecutionListener  
    SpringBootMockResolver  
    SpyBean  
    SpyBeans  
> SpyDefinition

- 简单案例

复制代码

```
1
2
3 @SpringBootTest(webEnvironment =
   SpringBootTest.WebEnvironment.RANDOM_PORT)
4 @AutoConfigureMockMvc
5 class UserManagerWithMockTest {
6
7     @Autowired
8     lateinit var webApplicationContext: WebApplicationContext
9
10    lateinit var mvc: MockMvc
11
12    //注入Mock的一个UserManagerRpc对象，替换Spring上下文中的
    UserManagerRpc
13    @MockBean
14    private userManagerRpc: UserManagerRpc;
15
```

```

16     @BeforeEach
17     public void before() {
18         // 获取mockmvc对象实例
19         mvc = MockMvcBuilders
20             .webApplicationContextSetup(webApplicationContext)
21             .defaultResponseCharacterEncoding<DefaultMockMvcBuilder>
22             (Charset.forName("UTF-8"))
23             .build()
24         }
25     }

```

java >

## ( 8 ) Mockito的基本API

- <https://www.javadoc.io/doc/org.mockito/mockito-core/3.0.0/org/mockito/Mockito.html>

### ( 8-1 ) 基本API

```

1 package org.mockito;
2
3
4 public class Mockito extends ArgumentMatchers {
5     static final MockitoCore MOCKITO_CORE = new MockitoCore();
6     public static final Answer<Object> RETURNS_DEFAULTS;
7     public static final Answer<Object> RETURNS_SMART_NULLS;
8     public static final Answer<Object> RETURNS MOCKS;
9     public static final Answer<Object> RETURNS_DEEP_STUBS;
10    public static final Answer<Object> CALLS_REAL_METHODS;

```

复制代码

```
11     public static final Answer<Object> RETURNS_SELF;
12
13     public Mockito() {
14     }
15
16     @CheckReturnValue
17     public static <T> T mock(Class<T> classToMock) {
18         return mock(classToMock, withSettings());
19     }
20
21     @CheckReturnValue
22     public static <T> T spy(T object) {
23         return MOCKITO_CORE.mock(object.getClass(),
24             withSettings().spiedInstance(object).defaultAnswer(CALLS_REAL_ME
25                 THODS));
26     }
27
28     @Incubating
29     @CheckReturnValue
30     public static <T> MockedStatic<T> mockStatic(Class<T>
31         classToMock) {
32         return mockStatic(classToMock, withSettings());
33     }
34
35     @Incubating
36     @CheckReturnValue
37     public static <T> MockedConstruction<T>
38         mockConstruction(Class<T> classToMock) {
39         return mockConstruction(classToMock, (index) → {
40             return withSettings();
41         }, (mock, context) → {
```

```
39     });
40 }
41
42
43 @CheckReturnValue
44 public static <T> OngoingStubbing<T> when(T methodCall) {
45     return MOCKITO_CORE.when(methodCall);
46 }
47
48 @CheckReturnValue
49 public static <T> T verify(T mock) {
50     return MOCKITO_CORE.verify(mock, times(1));
51 }
52
53 @CheckReturnValue
54 public static <T> T verify(T mock, VerificationMode mode) {
55     return MOCKITO_CORE.verify(mock, mode);
56 }
57
58 public static <T> void reset(T... mocks) {
59     MOCKITO_CORE.reset(mocks);
60 }
61
62 @CheckReturnValue
63 public static Stubber doThrow(Throwable... toBeThrown) {
64     return MOCKITO_CORE.stubber().doThrow(toBeThrown);
65 }
66
67
68 @CheckReturnValue
69 public static Stubber doCallRealMethod() {
70     return MOCKITO_CORE.stubber().doCallRealMethod();
```



```
71     }
72
73     @CheckReturnValue
74     public static Stubber doAnswer(Answer answer) {
75         return MOCKITO_CORE.stubber().doAnswer(answer);
76     }
77
78     @CheckReturnValue
79     public static Stubber doNothing() {
80         return MOCKITO_CORE.stubber().doNothing();
81     }
82
83     @CheckReturnValue
84     public static Stubber doReturn(Object toBeReturned) {
85         return MOCKITO_CORE.stubber().doReturn(toBeReturned);
86     }
87
88
89     @CheckReturnValue
90     public static VerificationMode times(int
wantedNumberOfInvocations) {
91         return
VerificationModeFactory.times(wantedNumberOfInvocations);
92     }
93
94     @CheckReturnValue
95     public static VerificationMode never() {
96         return times(0);
97     }
98
99     @CheckReturnValue
100    public static VerificationMode atLeastOnce() {
```

```
101         return VerificationModeFactory.atLeastOnce();
102     }
103
104     @CheckReturnValue
105     public static VerificationMode atLeast(int
minNumberOfInvocations) {
106         return
VerificationModeFactory.atLeast(minNumberOfInvocations);
107     }
108
109     @CheckReturnValue
110     public static VerificationMode atMostOnce() {
111         return VerificationModeFactory.atMostOnce();
112     }
113
114     @CheckReturnValue
115     public static VerificationMode atMost(int
maxNumberOfInvocations) {
116         return
VerificationModeFactory.atMost(maxNumberOfInvocations);
117     }
118
119     @CheckReturnValue
120     public static VerificationMode calls(int
wantedNumberOfInvocations) {
121         return
VerificationModeFactory.calls(wantedNumberOfInvocations);
122     }
123
124     @CheckReturnValue
125     public static VerificationMode only() {
126         return VerificationModeFactory.only();
```

```

127     }
128
129     @CheckReturnValue
130     public static VerificationWithTimeout timeout(long millis) {
131         return new Timeout(millis,
132             VerificationModeFactory.times(1));
133     }
134
135     @CheckReturnValue
136     public static VerificationAfterDelay after(long millis) {
137         return new After(millis,
138             VerificationModeFactory.times(1));
139     }
140
141     static {
142         RETURNS_DEFAULTS = Answers.RETURNS_DEFAULTS;
143         RETURNS_SMART_NULLS = Answers.RETURNS_SMART_NULLS;
144         RETURNS MOCKS = Answers.RETURNS MOCKS;
145         RETURNS DEEP STUBS = Answers.RETURNS DEEP STUBS;
146         CALLS_REAL_METHODS = Answers.CALLS_REAL_METHODS;
147         RETURNS_SELF = Answers.RETURNS_SELF;
148     }
149 }

```

java >

复制代码

```

1 package org.mockito.stubbing;
2
3 import org.mockito.NotExtensible;

```

```

4
5 @NotExtensible
6 public interface OngoingStubbing<T> {
7     OngoingStubbing<T> thenReturn(T var1);
8
9     OngoingStubbing<T> thenReturn(T var1, T... var2);
10
11     OngoingStubbing<T> thenThrow(Throwable... var1);
12
13     OngoingStubbing<T> thenThrow(Class<? extends Throwable>
var1);
14
15     OngoingStubbing<T> thenThrow(Class<? extends Throwable> var1,
Class<? extends Throwable>... var2);
16
17     OngoingStubbing<T> thenCallRealMethod();
18
19     OngoingStubbing<T> thenAnswer(Answer<?> var1);
20
21     OngoingStubbing<T> then(Answer<?> var1);
22
23     <M> M getMock();
24 }

```

java >

( 8-2 ) verify: 进行验证, 在调用方法后使用

复制代码

```

1
2 //mock object: can base interface mock object
3 List mockedList = mock(List.class);
4

```

```
5 //using mock object
6 mockedList.add("one");
7 mockedList.clear();
8
9 //verification
10 verify(mockedList).add("one");
11 verify(mockedList).clear();
```

java >

( 8-3 ) stubbing: 在调用方法前使用, 先进行存根, 为模拟对象准备一份模拟数据 ( 模拟方法、请求参数、响应数据 )

复制代码

```
1 //You can base class mock object, not just interface
2 LinkedList mockedList = mock(LinkedList.class);
3
4 //stubbing
5 when(mockedList.get(0)).thenReturn("first");
6 when(mockedList.get(1)).thenThrow(new RuntimeException());
7
8 //print "first"
9 mockedList.get(0)
10 //throw runtime exception
11 mockedList.get(1)
12 //prints "null" because get(999) was not stubbed
13 mockedList.get(999)
14
15
16 //Although it is possible to verify a stubbed invocation, usually
   it's just redundant (多余的)
```

```
17 //If your business code cares what get(0) returns, then something
    else breaks, before verify() gets executed
18 verify(mockedList).get(0);
```

java >

( 8-4 ) extra flexibility: stubbing中灵活的模拟、验证参数

复制代码

```
1 //mock object
2 List mockList = Mockito.mock(List.class);
3
4 //////////////////////////////////////
5 //stubbing using built-in anyInt() argument matcher
6 when(mockedList.get(anyInt())).thenReturn("element");
7
8 //following prints "element"
9 mockedList.get(999);
10
11 //////////////////////////////////////
12
13 mockedList.get(999);
14 //you can also verify using an argument matcher
15 verify(mockedList).get(anyInt());
16
17 //////////////////////////////////////
18
19 mockList.add("123456");
20 //argument matchers can also be written as Java 8 Lambdas
21 verify(mockedList).add(argThat(someString → someString.length()
    > 5));
```

java >

( 8-5-1 ) verify exact number of invocations / at least x / never: 验证同一个方法是否调用一次、多次、从未调用 ( 模拟的方法是否正常被调用, 跟调用方法的参数值是强相关的, 参数值和预期模拟方法中的参数值一致时, 则模拟方法才会被调用 )

复制代码

```
1 //mock object
2 List mockList = Mockito.mock(List.class);
3
4 //using mock object
5 mockedList.add("once");
6
7 //following two verifications work is same , times(1) is used by
  default
8 verify(mockedList).add("once");
9 verify(mockedList, times(1)).add("once");
```

java >

复制代码

```
1 //mock object
2 List mockList = Mockito.mock(List.class);
3
4 //using mock object
5 mockedList.add("twice");
6 mockedList.add("twice");
7
8 mockedList.add("three times");
```

```
9 mockedList.add("three times");
10 mockedList.add("three times");
11
12 //exact number of invocations verification 验证一下准确的调用次数是多
    少
13 verify(mockedList, times(2)).add("twice");
14 verify(mockedList, times(3)).add("three times");
```

java >

复制代码

```
1 //verification using never(), never() is an alias to times(0) 验证没
    有调用过
2 verify(mockedList, never()).add("never happened");
```

java >

#### ( 8-5-2 ) Verification with timeout (Since 1.8.5)

- 在

复制代码

```
1 //passes when someMethod() is called no later than within 100 ms
2 //exits immediately when verification is satisfied (e.g. may not
    wait full 100 ms)
3 verify(mock, timeout(100)).someMethod();
4 //above is an alias to:
5 verify(mock, timeout(100).times(1)).someMethod();
6
7 //passes as soon as someMethod() has been called 2 times under
    100 ms
```



```
8 verify(mock, timeout(100).times(2)).someMethod();
9
10 //equivalent: this also passes as soon as someMethod() has been
    called 2 times under 100 ms
11 verify(mock, timeout(100).atLeast(2)).someMethod();
```

java >

( 8-6 ) stubbing void method with exception: 为无返回值的方法进行存根 ( 抛出异常 )

```
1 //stubbing
2 doThrow(new RuntimeException()).when(mockedList).clear();
3
4 //following throws RuntimeException
5 mockedList.clear();
```

复制代码

java >

( 8-7 ) Stubbing consecutive calls: 存根被连续调用时, 即如果多次调用具有相同匹配器或参数的多个存根时, 不是链接调用, 而是每个存根将覆盖前一个

```
1 //stubbing
2 Mockito.when(mock.help("param"))
3         .thenReturn("foo1");//first stubbing
4         .thenReturn("foo2");//second stubbing
5
6 //First call: prints "foo1"
7 System.out.println(mock.help("param"));
8
9 //Second call: prints "foo2"
10 System.out.println(mock.help("param"));
```

复制代码

```
11
12 //Third call: prints "foo2" as well
13 //Any consecutive call last stubbing wins (多次连续调用时最后一个存根
    获胜)
14 System.out.println(mock.help("param"));
```

java >

复制代码

```
1 //All mock.someMethod("some arg") calls always return "two"
2
3 //stubbing1
4 when(mock.someMethod("some arg"))
5     .thenReturn("one")
6
7 //stubbing2
8 when(mock.someMethod("some arg"))
9     .thenReturn("two")
```

java >

( 8-8 ) stubbing with callbacks: 存根伴随应答 ( 回调 )

复制代码

```
1 when(mock.someMethod(anyString())).thenAnswer(
2     new Answer() {
3         public Object answer(InvocationOnMock invocation) {
4             Object[] args = invocation.getArguments();
5             Object mock = invocation.getMock();
6             return "called with arguments: " +
7                 Arrays.toString(args);
8         }
9     })
```

```
8  });  
9  
10 //Following prints "called with arguments: [foo]"  
11 System.out.println(mock.someMethod("foo"));
```

java >

( 8-9 ) stub a void method: 存根一个没有返回值的方法时, 请使用doXXX

- requires a different approach from when(Object) because the compiler does not like void methods inside brackets 要求一种不同与when的方法, 因为编译器不喜欢when括号内的空方法

```
1 //stubbing  
2 doThrow(new RuntimeException()).when(mockedList).clear();  
3  
4 //following throws RuntimeException  
5 mockedList.clear();  
6  
7  
8 doReturn(Object)  
9 doThrow(Throwable...)  
10 doThrow(Class)  
11 doAnswer(Answer)  
12 doNothing()  
13 doCallRealMethod()
```

复制代码

java >

( 8-10 ) spy on real objects: 监视一个真实对象

```
1 List list = new LinkedList();  
2 //spy a object
```

复制代码

```

3 List spy = spy(list);
4
5 //stubbing
6 when(spy.size()).thenReturn(100);
7
8 //using the spy calls *real* methods
9 spy.add("one");
10 spy.add("two");
11
12 //prints "one"
13 System.out.println(spy.get(0));
14
15 //size() method was stubbed - 100 is printed
16 System.out.println(spy.size());
17
18 //verify
19 verify(spy).add("one");
20 verify(spy).add("two");

```

java >

#### ( 8-11 ) Important gotcha on spy real objects

- 重要事项：在spy一个真实对象时，因为会调用到真实对象的方法，所以如果存根时真实方法抛异常，则会中断，所以也必须使用doXXX

```

1 List list = new LinkedList();
2 //spy a object
3 List spy = spy(list);
4
5 //Impossible: real method is called so spy.get(0) throws
  IndexOutOfBoundsException (the list is yet empty)

```

复制代码

```
6 when(spy.get(0)).thenReturn("foo");
7
8 //You have to use doReturn() for stubbing
9 doReturn("foo").when(spy).get(0);
```

java >

#### ( 8-12 ) capturing argument for further assertions ( since 1.8 )

- 调用方法后，为进一步的断言，捕捉论据（拦截到调用方法时的参数）
- 详细可参见（ 4 ）

```
1 //调用方法
2 //捕获论据
3 ArgumentCaptor<Person> argument =
  ArgumentCaptor.forClass(Person.class);
4
5 verify(mock).doSomething(argument.capture());
6
7 assertEquals("John", argument.getValue().getName());
```

复制代码

java >

#### ( 8-13 ) reset mocks ( since 1.8 )

- 重置模拟对象，一般情况下是用不到重置模拟对象，因为每一个测试方法会创建一个新的模拟对象，而同一个测试方法中使用同一个模拟对象的情况，需要重置模拟对象，防止模拟数据污染。

```
1 List mock = mock(List.class);
2 when(mock.size()).thenReturn(10);
3 mock.add(1);
4
```

复制代码

```
5 reset(mock);  
6 //at this point the mock forgot any interactions & stubbing 忘记交互  
  & 存根
```

java >

#### ( 8-14 ) real partial mocks ( since 1.8 )

- 实现部分模拟，1.8之前不支持部分模拟。

```
1 //you can enable partial mock capabilities selectively on mocks:  
2 Foo mock = mock(Foo.class);  
3 //Be sure the real implementation is 'safe'.  
4 //If real implementation throws exceptions or depends on specific  
  state of the object then you're in trouble.  
5 when(mock.someMethod()).thenCallRealMethod();
```

复制代码

java >