Spring、Spring Boot和TestNG测试指南

书栈(BookStack.CN)

目　录

致谢

当前文档 《Spring、Spring Boot和TestNG测试指南》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-08-09。

README

Spring、Spring Boot和TestNG测试指南

Spring、Spring Boot都提供了非常便利的测试工具，但遗憾的是官方文档的大多数例子都是基于JUnit的。本人比较喜欢用TestNG做单元、集成测试，所以开启了本项目收集了在Spring、Spring Boot项目中利用TestNG测试的例子。

章节列表

来源(书栈小编注)

> *https://github.com/chanjarster/spring-test-examples*

Chapter 0: 基本概念

Chapter 0: 基本概念

在了解学习本项目提供的例子之前，先了解一下什么是单元测试（Unit Testing，简称UT)和集成测试（Integration Testing，简称IT）。

如果你之前没有深究过这两个概念，那么你可能会得出如下错误的答案：

错误答案1：

> 单元测试就是对一个方法进行测试的测试

听上去很像那么回事儿，对吧？单元测试，就是测一个逻辑单元，所以就测一个方法就是单元测试，听上去很有道理是不是？但是，那么测试两个方法（这两个方法互相关联）的话叫什么呢？

错误答案2：

> 集成测试是把几个方法或者几个类放在一起测试

既然前面单元测试只测一个方法，那么几个方法放在一起测就是集成测试，听上去挺有道理的。那么是不是只要测一个以上的方法就是集成测试呢？

错误答案3：

> 集成测试就是和其他系统联合调试做的测试

听上去有点像SOA或者现在流行的微服务是吧。做这种测试的时候必须得各个开发团队紧密配合，一个不小心就会测试失败，然后就是各种返工，总之难度和火箭发射有的一拼。

那么正确答案是什么？其实这两个概念的解释比较冗长这里就不细讲了，只需记住UT和IT具备以下特征：

1. UT和IT必须是自动化的。
2. UT只专注于整个系统里的某一小部分，粒度没有规定，一般都比较小可以到方法级别。比如某个字符串串接方法。
3. UT不需要连接外部系统，在内存里跑跑就行了。
4. IT需要连接外部系统，比如连接数据库做CRUD测试。
5. 测试环境和生产环境是隔离的。
6. 能做UT的就不要做IT。

参考链接：

- Martin Fowler - Unit Test
- Wikipedia - Unit Testing
- Wikipedia - Integration Testing

Chapter 1: 基本用法

引言

Chapter 1: 基本用法 - 引言

本项目所有的项目均采用Maven的标准目录结构:

- `src/main/java`，程序java文件目录
- `src/main/resource`，程序资源文件目录
- `src/test/java`，测试代码目录
- `src/test/resources`，测试资源文件目录

并且所有Maven项目都可以使用 `mvn clean test` 方式跑单元测试，特别需要注意，只有文件名是 `*Test.java` 才会被执行，一定要注意这一点哦。

参考文档

- Maven Standard Directory Layout

认识TestNG

Chapter 1: 基本用法 - 认识TestNG

先认识一下TestNG，这里有一个FooServiceImpl，里面有两个方法，一个是给计数器+1，一个是获取当前计数器的值：

```
1.  @Component
2.  public class FooServiceImpl implements FooService {
3.
4.    private int count = 0;
5.
6.    @Override
7.    public void plusCount() {
8.      this.count++;
9.    }
10.
11.   @Override
12.   public int getCount() {
13.     return count;
14.   }
15.
16. }
```

然后我们针对它有一个FooServiceImplTest作为UT：

```
1.  public class FooServiceImplTest {
2.
3.    @Test
4.    public void testPlusCount() {
5.      FooService foo = new FooServiceImpl();
6.      assertEquals(foo.getCount(), 0);
7.
8.      foo.plusCount();
9.      assertEquals(foo.getCount(), 1);
10.   }
11.
12. }
```

注意看代码里的 `assertEquals(...)` ，我们利用它来判断 `Foo.getCount` 方法是否按照预期执行。所以，所谓的测试其实就是给定输入、执行一些方法，assert结果是否符合预期的过程。

参考文档

- TestNG documentation

使用Spring Testing工具

Chapter 1: 基本用法 - 使用Spring Testing工具

既然我们现在开发的是一个Spring项目，那么肯定会用到Spring Framework的各种特性，这些特性实在是太好用了，它能够大大提高我们的开发效率。那么自然而然，你会想在测试代码里也能够利用Spring Framework提供的特性，来提高测试代码的开发效率。这部分我们会讲如何使用Spring提供的测试工具来做测试。

例子1

源代码见FooServiceImplTest:

```
1.  @ContextConfiguration(classes = FooServiceImpl.class)
2.  public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
4.    @Autowired
5.    private FooService foo;
6.
7.    @Test
8.    public void testPlusCount() throws Exception {
9.      assertEquals(foo.getCount(), 0);
10.
11.     foo.plusCount();
12.     assertEquals(foo.getCount(), 1);
13.   }
14.
15. }
```

在上面的源代码里我们要注意三点:

1. 测试类继承了AbstractTestNGSpringContextTests，如果不这么做测试类是无法启动Spring容器的
2. 使用了[@ContextConfiguration][javadoc-ContextConfiguration]来加载被测试的Bean: `FooServiceImpl`
3. `FooServiceImpl`是`@Component`

以上三点缺一不可。

例子2

在这个例子里，我们将 `@Configuration` 作为nested static class放在测试类里，根据@ContextConfiguration的文档，它会在默认情况下查找测试类的nested static @Configuration class，用它来导入Bean。

源代码见FooServiceImplTest:

```
1.  @ContextConfiguration
2.  public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
4.    @Autowired
5.    private FooService foo;
6.
7.    @Test
8.    public void testPlusCount() throws Exception {
9.      assertEquals(foo.getCount(), 0);
10.
11.     foo.plusCount();
12.     assertEquals(foo.getCount(), 1);
13.   }
14.
15.   @Configuration
16.   @Import(FooServiceImpl.class)
17.   static class Config {
18.   }
19.
20. }
```

例子3

在这个例子里，我们将 `@Configuration` 放到外部，并让@ContextConfiguration去加载。

源代码见Config:

```
1.  @Configuration
2.  @Import(FooServiceImpl.class)
3.  public class Config {
4.  }
```

FooServiceImplTest:

```
1.  @ContextConfiguration(classes = Config.class)
```

```
public class FooServiceImplTest extends AbstractTestNGSpringContextTests {

  @Autowired
  private FooService foo;

  @Test
  public void testPlusCount() throws Exception {
    assertEquals(foo.getCount(), 0);

    foo.plusCount();
    assertEquals(foo.getCount(), 1);
  }

}
```

需要注意的是，如果 `@Configuration` 是专供某个测试类使用的话，把它放到外部并不是一个好主意，因为它有可能会被 `@ComponentScan` 扫描到，从而产生一些奇怪的问题。

参考文档

- Spring Framework Testing
- Context configuration with annotated classes

使用Spring Boot Testing工具

Chapter 1: 基本用法 - 使用Spring Boot Testing工具

前面一个部分讲解了如何使用Spring Testing工具来测试Spring项目，现在我们讲解如何使用Spring Boot Testing工具来测试Spring Boot项目。

> 在*Spring Boot*项目里既可以使用*Spring Boot Testing*工具，也可以使用*Spring Testing*工具。在*Spring*项目里，一般使用*Spring Testing*工具，虽然理论上也可以使用*Spring Boot Testing*，不过因为*Spring Boot Testing*工具会引入*Spring Boot*的一些特性比如*AutoConfiguration*，这可能会给你的测试带来一些奇怪的问题，所以一般不推荐这样做。

例子1: 直接加载Bean

使用Spring Boot Testing工具只需要将 `@ContextConfiguration` 改成 `@SpringBootTest` 即可，源代码见FooServiceImpltest:

```
1.  @SpringBootTest(classes = FooServiceImpl.class)
2.  public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
4.    @Autowired
5.    private FooService foo;
6.
7.    @Test
8.    public void testPlusCount() throws Exception {
9.      assertEquals(foo.getCount(), 0);
10.
11.     foo.plusCount();
12.     assertEquals(foo.getCount(), 1);
13.   }
14.
15. }
```

例子2: 使用内嵌@Configuration加载Bean

源代码见FooServiceImpltest:

```
1.  @SpringBootTest
2.  public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
```

```
  @Autowired
  private FooService foo;

  @Test
  public void testPlusCount() throws Exception {
    assertEquals(foo.getCount(), 0);

    foo.plusCount();
    assertEquals(foo.getCount(), 1);
  }

  @Configuration
  @Import(FooServiceImpl.class)
  static class Config {
  }

}
```

例子3：使用外部@Configuration加载Bean

Config:

```
@Configuration
@Import(FooServiceImpl.class)
public class Config {
}
```

FooServiceImpltest:

```
@SpringBootTest(classes = Config.class)
public class FooServiceImplTest extends AbstractTestNGSpringContextTests {

  @Autowired
  private FooService foo;

  @Test
  public void testPlusCount() throws Exception {
    assertEquals(foo.getCount(), 0);

    foo.plusCount();
    assertEquals(foo.getCount(), 1);
```

```
13.   }
14.
15. }
```

这个例子和例子2差不多，只不过将@Configuration放到了外部。

例子4: 使用@SpringBootConfiguration

前面的例子 `@SpringBootTest` 的用法和 `@ContextConfiguration` 差不多。不过根据 `@SpringBootTest` 的文档:

1. 它会尝试加载 `@SpringBootTest(classes=...)` 的定义的Annotated classes。Annotated classes的定义在ContextConfiguration中有说明。
2. 如果没有设定 `@SpringBootTest(classes=...)` ，那么会去找当前测试类的nested `@Configuration` class
3. 如果上一步找到，则会尝试查找 `@SpringBootConfiguration` ，查找的路径有: 1)看当前测试类是否 `@SpringBootConfiguration` ，2)在当前测试类所在的package里找。

所以我们可以利用这个特性来进一步简化测试代码。

Config:

```
1. @SpringBootConfiguration
2. @Import(FooServiceImpl.class)
3. public class Config {
4. }
```

FooServiceImpltest:

```
1. @SpringBootTest
2. public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
4.   @Autowired
5.   private FooService foo;
6.
7.   @Test
8.   public void testPlusCount() throws Exception {
9.     assertEquals(foo.getCount(), 0);
10.
11.    foo.plusCount();
12.    assertEquals(foo.getCount(), 1);
13.  }
```

```
14.
15. }
```

例子5：使用@ComponentScan扫描Bean

前面的例子我们都使用 `@Import` 来加载Bean，虽然这中方法很精确，但是在大型项目中很麻烦。

在常规的Spring Boot项目中，一般都是依靠自动扫描机制来加载Bean的，所以我们希望我们的测试代码也能够利用自动扫描机制来加载Bean。

Config:

```
1. @SpringBootConfiguration
2. @ComponentScan(basePackages = "me.chanjar.basic.service")
3. public class Config {
4. }
```

FooServiceImpltest:

```
1. @SpringBootTest
2. public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
4.   @Autowired
5.   private FooService foo;
6.
7.   @Test
8.   public void testPlusCount() throws Exception {
9.     assertEquals(foo.getCount(), 0);
10.
11.    foo.plusCount();
12.    assertEquals(foo.getCount(), 1);
13.  }
14.
15. }
```

例子6：使用@SpringBootApplication

也可以在测试代码上使用 `@SpringBootApplication` ，它有这么几个好处：

1. 自身 `SpringBootConfiguration`
2. 提供了 `@ComponentScan` 配置，以及默认的excludeFilter，有了这些filter Spring在初

始化ApplicationContext的时候会排除掉某些Bean和@Configuration

3. 启用了`EnableAutoConfiguration`，这个特性能够利用Spring Boot来自动化配置所需要的外部资源，比如数据库、JMS什么的，这在集成测试的时候非常有用。

Config:

```
1. @SpringBootApplication(scanBasePackages = "me.chanjar.basic.service")
2. public class Config {
3. }
```

FooServiceImpltest:

```
1. @SpringBootTest
2. public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
3.
4.   @Autowired
5.   private FooService foo;
6.
7.   @Test
8.   public void testPlusCount() throws Exception {
9.     assertEquals(foo.getCount(), 0);
10.
11.    foo.plusCount();
12.    assertEquals(foo.getCount(), 1);
13.  }
14.
15. }
```

避免@SpringBootConfiguration冲突

当`@SpringBootTest`没有定义`(classes=...`，且没有找到nested `@Configuration` class的情况下，会尝试查询`@SpringBootConfiguration`，如果找到多个的话则会抛出异常:

```
1. Caused by: java.lang.IllegalStateException: Found multiple
   @SpringBootConfiguration annotated classes [Generic bean: class [...]; scope=;
   abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0;
   autowireCandidate=true; primary=false; factoryBeanName=null;
   factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in
   file [/Users/qianjia/workspace-os/spring-test-examples/basic/target/test-
   classes/me/chanjar/basic/springboot/ex7/FooServiceImplTest1.class], Generic
   bean: class [me.chanjar.basic.springboot.ex7.FooServiceImplTest2]; scope=;
```

```
    abstract=false; lazyInit=false; autowireMode=0; dependencyCheck=0;
    autowireCandidate=true; primary=false; factoryBeanName=null;
    factoryMethodName=null; initMethodName=null; destroyMethodName=null; defined in
    file [...]]
```

比如以下代码就会造成这个问题：

```
@SpringBootApplication(scanBasePackages = "me.chanjar.basic.service")
public class Config1 {
}

@SpringBootApplication(scanBasePackages = "me.chanjar.basic.service")
public class Config2 {
}

@SpringBootTest
public class FooServiceImplTest extends AbstractTestNGSpringContextTests {
  // ...
}
```

解决这个问题的方法有就是避免自动查询 `@SpringBootConfiguration` ：

1. 定义 `@SpringBootTest(classes=...)`
2. 提供nested `@Configuration` class

最佳实践

除了单元测试（不需要初始化ApplicationContext的测试）外，尽量将测试配置和生产配置保持一致。比如如果生产配置里启用了AutoConfiguration，那么测试配置也应该启用。因为只有这样才能够在测试环境下发现生产环境的问题，也避免出现一些因为配置不同导致的奇怪问题。

在测试代码之间尽量做到配置共用，这么做的优点有3个：

1. 能够有效利用Spring TestContext Framework的缓存机制，ApplicationContext只会创建一次，后面的测试会直接用已创建的那个，加快测试代码运行速度。
2. 当项目中的Bean很多的时候，这么做能够降低测试代码复杂度，想想如果每个测试代码都有一套自己的@Configuration或其变体，那得多吓人。

参考文档

- Spring Framework Testing

- Spring Boot Testing
- Spring TestContext Framework

Chapter 2: Annotations

引言

Chapter 2: Annotations

Spring & Spring Boot Testing工具提供了一些方便测试的Annotation，本章节会对其中的一些做一些讲解。

@TestPropertySource

Chapter 2: Annotations - @TestPropertySource

@TestPropertySource可以用来覆盖掉来自于系统环境变量、Java系统属性、@PropertySource的属性。

同时 `@TestPropertySource(properties=...)` 优先级高于 `@TestPropertySource(locations=...)` 。

利用它我们可以很方便的在测试代码里微调、模拟配置（比如修改操作系统目录分隔符、数据源等）。

例子1: 使用Spring Testing工具

我们先使用@PropertySource将一个外部properties文件加载进来，PropertySourceConfig:

```
1. @Configuration
2. @PropertySource("classpath:me/chanjar/annotation/testps/ex1/property-
   source.properties")
3. public class PropertySourceConfig {
4. }
```

```
1. file: property-source.properties
2. foo=abc
```

然后我们用@TestPropertySource覆盖了这个property:

```
1. @TestPropertySource(properties = { "foo=xyz" ...
```

最后我们测试了是否覆盖成功（结果是成功的）:

```
1. @Test
2. public void testOverridePropertySource() {
3.   assertEquals(environment.getProperty("foo"), "xyz");
4. }
```

同时我们还对@TestPropertySource做了一些其他的测试，具体情况你可以自己观察。为了方便你观察@TestPropertySource对系统环境变量和Java系统属性的覆盖效果，我们在一开始打印出了它们的值。

源代码TestPropertyTest:

```
@ContextConfiguration(classes = PropertySourceConfig.class)
@TestPropertySource(
    properties = { "foo=xyz", "bar=uvw", "PATH=aaa", "java.runtime.name=bbb" },
    locations = "classpath:me/chanjar/annotation/testps/ex1/test-property-source.properties"
)
public class TestPropertyTest extends AbstractTestNGSpringContextTests implements EnvironmentAware {

  private Environment environment;

  @Override
  public void setEnvironment(Environment environment) {
    this.environment = environment;
    Map<String, Object> systemEnvironment = ((ConfigurableEnvironment) environment).getSystemEnvironment();
    System.out.println("=== System Environment ===");
    System.out.println(getMapString(systemEnvironment));
    System.out.println();

    System.out.println("=== Java System Properties ===");
    Map<String, Object> systemProperties = ((ConfigurableEnvironment) environment).getSystemProperties();
    System.out.println(getMapString(systemProperties));
  }

  @Test
  public void testOverridePropertySource() {
    assertEquals(environment.getProperty("foo"), "xyz");
  }

  @Test
  public void testOverrideSystemEnvironment() {
    assertEquals(environment.getProperty("PATH"), "aaa");
  }

  @Test
  public void testOverrideJavaSystemProperties() {
    assertEquals(environment.getProperty("java.runtime.name"), "bbb");
  }
```

```
37.
38.   @Test
39.   public void testInlineTestPropertyOverrideResourceLocationTestProperty() {
40.     assertEquals(environment.getProperty("bar"), "uvw");
41.   }
42.
43.   private String getMapString(Map<String, Object> map) {
44.     return String.join("\n",
45.         map.keySet().stream().map(k -> k + "=" + map.get(k)).collect(toList())
46.     );
47.   }
48. }
```

例子2: 使用Spring Boot Testing工具

@TestPropertySource也可以和@SpringBootTest一起使用。

源代码见TestPropertyTest:

```
1. @SpringBootTest(classes = PropertySourceConfig.class)
2. @TestPropertySource(
3.     properties = { "foo=xyz", "bar=uvw", "PATH=aaa", "java.runtime.name=bbb" },
4.     locations = "classpath:me/chanjar/annotation/testps/ex1/test-property-
   source.properties"
5. )
6. public class TestPropertyTest extends AbstractTestNGSpringContextTests
   implements EnvironmentAware {
7.   // ...
8. }
```

参考文档

- Spring Framework Testing
- Spring Boot Testing
- Context configuration with test property sources

@ActiveProfile

Chapter 2: Annotations - @ActiveProfiles

@ActiveProfiles可以用来在测试的时候启用某些Profile的Bean。本章节的测试代码使用了下面的这个配置:

```
@Configuration
public class Config {

  @Bean
  @Profile("dev")
  public Foo fooDev() {
    return new Foo("dev");
  }

  @Bean
  @Profile("product")
  public Foo fooProduct() {
    return new Foo("product");
  }

  @Bean
  @Profile("default")
  public Foo fooDefault() {
    return new Foo("default");
  }

  @Bean
  public Bar bar() {
    return new Bar("no profile");
  }

}
```

例子1: 不使用ActiveProfiles

在没有@ActiveProfiles的时候，profile=default和没有设定profile的Bean会被加载到。

源代码ActiveProfileTest:

```
@ContextConfiguration(classes = Config.class)
```

```
2.  public class ActiveProfileTest extends AbstractTestNGSpringContextTests {
3.
4.    @Autowired
5.    private Foo foo;
6.
7.    @Autowired
8.    private Bar bar;
9.
10.   @Test
11.   public void test() {
12.     assertEquals(foo.getName(), "default");
13.     assertEquals(bar.getName(), "no profile");
14.   }
15.
16. }
```

例子2：使用ActiveProfiles

当使用了@ActiveProfiles的时候，profile匹配的和没有设定profile的Bean会被加载到。

源代码ActiveProfileTest：

```
1.  @ContextConfiguration(classes = Config.class)
2.  [@ActiveProfiles][doc-active-profiles]("product")
3.  public class ActiveProfileTest extends AbstractTestNGSpringContextTests {
4.
5.    @Autowired
6.    private Foo foo;
7.
8.    @Autowired
9.    private Bar bar;
10.
11.   @Test
12.   public void test() {
13.     assertEquals(foo.getName(), "product");
14.     assertEquals(bar.getName(), "no profile");
15.   }
16.
17. }
```

总结

- 在没有@ActiveProfiles的时候，profile=default和没有设定profile的Bean会被加载到。
- 当使用了@ActiveProfiles的时候，profile匹配的和没有设定profile的Bean会被加载到。

@ActiveProfiles同样也可以和@SpringBootTest配合使用，这里就不举例说明了。

参考文档

- Spring Framework Testing
- Spring Boot Testing

@JsonTest

Chapter 2: Annotations - @JsonTest

@JsonTest是Spring Boot提供的方便测试JSON序列化反序列化的测试工具，在Spring Boot的文档中有一些介绍。

需要注意的是@JsonTest需要Jackson的 `ObjectMapper` ，事实上如果你的Spring Boot项目添加了 `spring-web` 的Maven依赖，JacksonAutoConfiguration就会自动为你配置一个：

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

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

这里没有提供关于日期时间的例子，关于这个比较复杂，可以看我的另一篇文章：Spring Boot Jackson对于日期时间类型处理的例子。

例子1：简单例子

源代码见SimpleJsonTest：

```
@SpringBootTest(classes = SimpleJsonTest.class)
@JsonTest
public class SimpleJsonTest extends AbstractTestNGSpringContextTests {

  @Autowired
  private JacksonTester<Foo> json;

  @Test
  public void testSerialize() throws Exception {
    Foo details = new Foo("Honda", 12);
    // 使用通包下的json文件测试结果是否正确
    assertThat(this.json.write(details)).isEqualToJson("expected.json");
    // 或者使用基于JSON path的校验
    assertThat(this.json.write(details)).hasJsonPathStringValue("@.name");

```

```
    assertThat(this.json.write(details)).extractingJsonPathStringValue("@.name").isEq
16.     assertThat(this.json.write(details)).hasJsonPathNumberValue("@.age");
17. 
    assertThat(this.json.write(details)).extractingJsonPathNumberValue("@.age").isEqu
18.   }
19. 
20.   @Test
21.   public void testDeserialize() throws Exception {
22.     String content = "{\"name\":\"Ford\",\"age\":13}";
23.     Foo actual = this.json.parseObject(content);
24.     assertThat(actual).isEqualTo(new Foo("Ford", 13));
25.     assertThat(actual.getName()).isEqualTo("Ford");
26.     assertThat(actual.getAge()).isEqualTo(13);
27. 
28.   }
29. 
30. }
```

例子2： 测试@JsonComponent

@JsonTest可以用来测试@JsonComponent。

这个例子里使用了自定义的 `@JsonComponent` FooJsonComponent:

```
1.  @JsonComponent
2.  public class FooJsonComponent {
3.  
4.    public static class Serializer extends JsonSerializer<Foo> {
5.      @Override
6.      public void serialize(Foo value, JsonGenerator gen, SerializerProvider
    serializers)
7.          throws IOException, JsonProcessingException {
8.        // ...
9.      }
10. 
11.   }
12. 
13.   public static class Deserializer extends JsonDeserializer<Foo> {
14. 
15.     @Override
16.     public Foo deserialize(JsonParser p, DeserializationContext ctxt) throws
```

```
    IOException, JsonProcessingException {
17.       // ...
18.     }
19.
20.   }
21.
22. }
```

测试代码JsonComponentJsonTest:

```
1.  @SpringBootTest(classes = { JsonComponentJacksonTest.class,
    FooJsonComponent.class })
2.  @JsonTest
3.  public class JsonComponentJacksonTest extends AbstractTestNGSpringContextTests
    {
4.
5.    @Autowired
6.    private JacksonTester<Foo> json;
7.
8.    @Test
9.    public void testSerialize() throws Exception {
10.     Foo details = new Foo("Honda", 12);
11.
    assertThat(this.json.write(details).getJson()).isEqualTo("\"name=Honda,age=12\"")
12.   }
13.
14.   @Test
15.   public void testDeserialize() throws Exception {
16.     String content = "\"name=Ford,age=13\"";
17.     Foo actual = this.json.parseObject(content);
18.     assertThat(actual).isEqualTo(new Foo("Ford", 13));
19.     assertThat(actual.getName()).isEqualTo("Ford");
20.     assertThat(actual.getAge()).isEqualTo(13);
21.
22.   }
23.
24. }
```

例子3: 使用@ContextConfiguration

事实上@JsonTest也可以配合 `@ContextConfiguration` 一起使用。

源代码见ThinJsonTest:

```
1.  @JsonTest
2.  @ContextConfiguration(classes = JsonTest.class)
3.  public class ThinJsonTest extends AbstractTestNGSpringContextTests {
4.
5.    @Autowired
6.    private JacksonTester<Foo> json;
7.
8.    @Test
9.    public void testSerialize() throws Exception {
10.     // ...
11.   }
12.
13.   @Test
14.   public void testDeserialize() throws Exception {
15.     // ...
16.   }
17.
18. }
```

参考文档

- Spring Framework Testing
- Spring Boot Testing
- @JsonTest
- JsonComponent
- JacksonAutoConfiguration
- JacksonTester
- GsonTester
- BasicJsonTester

@OverrideAutoConfiguration

Chapter 2: Annotations - @OverrideAutoConfiguration

在Chapter 1: 基本用法 - 使用Spring Boot Testing工具里提到:

> 除了单元测试（不需要初始化*ApplicationContext*的测试）外，尽量将测试配置和生产配置保持一致。比如如果生产配置里启用了*AutoConfiguration*，那么测试配置也应该启用。因为只有这样才能够在测试环境下发现生产环境的问题，也避免出现一些因为配置不同导致的奇怪问题。

那么当我们想在测试代码里关闭Auto Configuration如何处理？

1. 方法1：提供另一套测试配置
2. 方法2：使用`@OverrideAutoConfiguration`

方法1虽然能够很好的解决问题，但是比较麻烦。而方法2则能够不改变原有配置、不提供新的配置的情况下，就能够关闭Auto Configuration。

在本章节的例子里，我们自己做了一个Auto Configuration类，AutoConfigurationEnableLogger:

```
1.  @Configuration
2.  public class AutoConfigurationEnableLogger {
3.
4.    private static final Logger LOGGER =
    LoggerFactory.getLogger(AutoConfigurationEnableLogger.class);
5.
6.    public AutoConfigurationEnableLogger() {
7.      LOGGER.info("Auto Configuration Enabled");
8.    }
9.
10. }
```

并且在`META-INF/spring.factories`里注册了它:

```
1.  org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
2.  me.chanjar.annotation.overrideac.AutoConfigurationEnableLogger
```

这样一来，只要Spring Boot启动了Auto Configuration就会打印出日志:

```
1.  2017-08-24 16:44:52.789  INFO 13212 --- [           main]
    m.c.a.o.AutoConfigurationEnableLogger    : Auto Configuration Enabled
```

例子1: 未关闭Auto Configuration

源代码见BootTest:

```
1. @SpringBootTest
2. @SpringBootApplication
3. public class BootTest extends AbstractTestNGSpringContextTests {
4.
5.   @Test
6.   public void testName() throws Exception {
7.
8.   }
9. }
```

查看输出的日志，会发现Auto Configuration已经启用。

例子2: 关闭Auto Configuration

然后我们用@OverrideAutoConfiguration关闭了Auto Configuration。

源代码见BootTest:

```
1. @SpringBootTest
2. @OverrideAutoConfiguration(enabled = false)
3. @SpringBootApplication
4. public class BootTest extends AbstractTestNGSpringContextTests {
5.
6.   @Test
7.   public void testName() throws Exception {
8.
9.   }
10. }
```

再查看输出的日志，就会发现Auto Configuration已经关闭。

参考文档

- Spring Framework Testing
- Spring Boot Testing
- [Context configuration with test property sources][doc-test-property-

```
    source]
```

@TestConfiguration

Chapter 2: Annotations - @TestConfiguration

@TestConfiguration是Spring Boot Test提供的一种工具，用它我们可以在一般的@Configuration之外补充测试专门用的Bean或者自定义的配置。

@TestConfiguration实际上是一种@TestComponent，@TestComponent是另一种@Component，在语义上用来指定某个Bean是专门用于测试的。

需要特别注意，你应该使用一切办法避免在生产代码中自动扫描到@TestComponent。
如果你使用 `@SpringBootApplication` 启动测试或者生产代码，@TestComponent会自动被排除掉，如果不是则需要像 `@SpringBootApplication` 一样添加 `TypeExcludeFilter` ：

```
1.  //...
2.  @ComponentScan(excludeFilters = {
3.    @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
4.    // ...})
5.  public @interface SpringBootApplication
```

例子1：作为内部类

@TestConfiguration和@Configuration不同，它不会阻止@SpringBootTest去查找机制（在Chapter 1: 基本用法 - 使用Spring Boot Testing工具 - 例子4提到过），正如@TestConfiguration的javadoc所说，它只是对既有配置的一个补充。

所以我们在测试代码上添加@SpringBootConfiguration，
用 `@SpringBootTest(classes=...)` 或者在同package里添加@SpringBootConfiguration类都是可以的。

而且@TestConfiguration作为内部类的时候它是会被@SpringBootTest扫描掉的，这点和@Configuration一样。

测试代码TestConfigurationTest：

```
1.  @SpringBootTest
2.  @SpringBootConfiguration
3.  public class TestConfigurationTest extends AbstractTestNGSpringContextTests {
4.
5.    @Autowired
6.    private Foo foo;
7.
```

```
  @Test
  public void testPlusCount() throws Exception {
    assertEquals(foo.getName(), "from test config");
  }

  @TestConfiguration
  public class TestConfig {

    @Bean
    public Foo foo() {
      return new Foo("from test config");
    }

  }
}
```

例子2：对@Configuration的补充和覆盖

@TestConfiguration能够：

1. 补充额外的Bean
2. 覆盖已存在的Bean

要特别注意第二点，@TestConfiguration能够直接覆盖已存在的Bean，这一点正常的@Configuration是做不到的。

我们先提供了一个正常的@Configuration（Config）：

```
@Configuration
public class Config {

  @Bean
  public Foo foo() {
    return new Foo("from config");
  }
}
```

又提供了一个@TestConfiguration，在里面覆盖了 `foo` Bean，并且提供了 `foo2` Bean（TestConfig）：

```
@TestConfiguration
public class TestConfig {
```

```
3.
4.    // 这里不需要@Primary之类的机制，直接就能够覆盖
5.    @Bean
6.    public Foo foo() {
7.      return new Foo("from test config");
8.    }
9.
10.   @Bean
11.   public Foo foo2() {
12.     return new Foo("from test config2");
13.   }
14. }
```

测试代码TestConfigurationTest:

```
1.  @SpringBootTest(classes = { Config.class, TestConfig.class })
2.  public class TestConfigurationTest extends AbstractTestNGSpringContextTests {
3.
4.    @Qualifier("foo")
5.    @Autowired
6.    private Foo foo;
7.
8.    @Qualifier("foo2")
9.    @Autowired
10.   private Foo foo2;
11.
12.   @Test
13.   public void testPlusCount() throws Exception {
14.     assertEquals(foo.getName(), "from test config");
15.     assertEquals(foo2.getName(), "from test config2");
16.
17.   }
18.
19. }
```

再查看输出的日志，就会发现Auto Configuration已经关闭。

例子3：避免@TestConfiguration被扫描到

在上面的这个例子里的TestConfig是会被@ComponentScan扫描到的，如果要避免被扫描到，在本文开头已经提到过了。

先来看一下没有做任何过滤的情形，我们先提供了一个@SpringBootConfiguration（IncludeConfig）：

```
@SpringBootConfiguration
@ComponentScan
public interface IncludeConfig {
}
```

然后有个测试代码引用了它（TestConfigIncludedTest）：

```
@SpringBootTest(classes = IncludeConfig.class)
public class TestConfigIncludedTest extends AbstractTestNGSpringContextTests {

  @Autowired(required = false)
  private TestConfig testConfig;

  @Test
  public void testPlusCount() throws Exception {
    assertNotNull(testConfig);

  }

}
```

从这段代码可以看到 `TestConfig` 被加载了。

现在我们使用TypeExcludeFilter来过滤@TestConfiguration（ExcludeConfig1）：

```
@SpringBootConfiguration
@ComponentScan(excludeFilters = {
    @ComponentScan.Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class)
})
public interface ExcludeConfig1 {
}
```

再来看看结果（TestConfigExclude_1_Test）：

```
@SpringBootTest(classes = ExcludeConfig1.class)
public class TestConfigExclude_1_Test extends AbstractTestNGSpringContextTests
{

```

```
4.    @Autowired(required = false)
5.    private TestConfig testConfig;
6.
7.    @Test
8.    public void test() throws Exception {
9.      assertNull(testConfig);
10.
11.   }
12.
13. }
```

还可以用@SpringBootApplication来排除 `TestConfig` （ExcludeConfig2）：

```
1. @SpringBootApplication
2. public interface ExcludeConfig2 {
3. }
```

看看结果（TestConfigExclude_2_Test）：

```
1. @SpringBootTest(classes = ExcludeConfig2.class)
2. public class TestConfigExclude_2_Test extends AbstractTestNGSpringContextTests
   {
3.
4.    @Autowired(required = false)
5.    private TestConfig testConfig;
6.
7.    @Test
8.    public void testPlusCount() throws Exception {
9.      assertNull(testConfig);
10.
11.   }
12.
13. }
```

参考文档

- Spring Framework Testing
- Spring Boot Testing
- Detecting test configuration
- Excluding test configuration

Chapter 3: 使用Mockito

Chapter 3: 使用Mockito

Mock测试技术能够避免你为了测试一个方法，却需要自行构建整个依赖关系的工作，并且能够让你专注于当前被测试对象的逻辑，而不是其依赖的其他对象的逻辑。

举例来说，比如你需要测试 `Foo.methodA` ，而这个方法依赖了 `Bar.methodB` ，又传递依赖到了 `Zoo.methodC` ，于是它们的依赖关系就是 `Foo->Bar->Zoo` ，所以在测试代码里你必须自行new Bar和Zoo。

有人会说：”我直接用Spring的DI机制不就行了吗？”的确，你可以用Spring的DI机制，不过解决不了测试代码耦合度过高的问题：

因为Foo方法内部调用了Bar和Zoo的方法，所以你对其做单元测试的时候，必须完全了解Bar和Zoo方法的内部逻辑，并且谨慎的传参和assert结果，一旦Bar和Zoo的代码修改了，你的Foo测试代码很可能就会运行失败。

所以这个时候我们需要一种机制，能过让我们在测试Foo的时候不依赖于Bar和Zoo的具体实现，即不关心其内部逻辑，只关注Foo内部的逻辑，从而将Foo的每个逻辑分支都测试到。

所以业界就产生了Mock技术，它可以让我们做一个假的Bar（不需要Zoo，因为只有真的Bar才需要Zoo），然后控制这个假的Bar的行为（让它返回什么就返回什么），以此来测试Foo的每个逻辑分支。

你肯定会问，这样的测试有意义吗？在真实环境里Foo用的是真的Bar而不是假的Bar，你用假的Bar测试成功能代表真实环境不出问题？

其实假Bar代表的是一个行为正确的Bar，用它来测试就能验证”在Bar行为正确的情况下Foo的行为是否正确”，而真Bar的行为是否正确会由它自己的测试代码来验证。

Mock技术的另一个好处是能够让你尽量避免集成测试，比如我们可以Mock一个Repository（数据库操作类），让我们尽量多写单元测试，提高测试代码执行效率。

`spring-boot-starter-test` 依赖了Mockito，所以我们会在本章里使用Mockito来讲解。

被测试类

先介绍一下接下来要被我们测试的类Foo、Bar俩兄弟。

```
1.  public interface Foo {
2.
3.    boolean checkCodeDuplicate(String code);
4.
5.  }
```

```
6.
7.  public interface Bar {
8.
9.    Set<String> getAllCodes();
10.
11. }
12.
13. @Component
14. public class FooImpl implements Foo {
15.
16.   private Bar bar;
17.
18.   @Override
19.   public boolean checkCodeDuplicate(String code) {
20.     return bar.getAllCodes().contains(code);
21.   }
22.
23.   @Autowired
24.   public void setBar(Bar bar) {
25.     this.bar = bar;
26.   }
27.
28. }
```

例子1: 不使用Mock技术

源代码NoMockTest:

```
1.  public class NoMockTest {
2.
3.    @Test
4.    public void testCheckCodeDuplicate1() throws Exception {
5.
6.      FooImpl foo = new FooImpl();
7.      foo.setBar(new Bar() {
8.        @Override
9.        public Set<String> getAllCodes() {
10.         return Collections.singleton("123");
11.       }
12.     });
13.     assertEquals(foo.checkCodeDuplicate("123"), true);
```

```
14.
15.   }
16.
17.   @Test
18.   public void testCheckCodeDuplicate2() throws Exception {
19.
20.     FooImpl foo = new FooImpl();
21.     foo.setBar(new FakeBar(Collections.singleton("123")));
22.     assertEquals(foo.checkCodeDuplicate("123"), true);
23.
24.   }
25.
26.   public class FakeBar implements Bar {
27.
28.     private final Set<String> codes;
29.
30.     public FakeBar(Set<String> codes) {
31.       this.codes = codes;
32.     }
33.
34.     @Override
35.     public Set<String> getAllCodes() {
36.       return codes;
37.     }
38.
39.   }
40.
41. }
```

这个测试代码里用到了两种方法来做假的Bar:

1. 匿名内部类
2. 做了一个 `FakeBar`

这两种方式都不是很优雅，看下面使用Mockito的例子。

例子2: 使用Mockito

源代码[MockitoTest][src-MockitoTest]:

```
1. public class MockitoTest {
2.
```

```
3.    @Mock
4.    private Bar bar;
5.
6.    @InjectMocks
7.    private FooImpl foo;
8.
9.    @BeforeMethod(alwaysRun = true)
10.   public void initMock() {
11.     MockitoAnnotations.initMocks(this);
12.   }
13.
14.   @Test
15.   public void testCheckCodeDuplicate() throws Exception {
16.
17.     when(bar.getAllCodes()).thenReturn(Collections.singleton("123"));
18.     assertEquals(foo.checkCodeDuplicate("123"), true);
19.
20.   }
21.
22. }
```

1. 我们先给了一个Bar的Mock实现：`@Mock private Bar bar;`
2. 然后又规定了`getAllCodes`方法的返回值：`when(bar.getAllCodes()).thenReturn(Collections.singleton("123"))`。这样就把一个假的Bar定义好了。
3. 最后利用Mockito把Bar注入到Foo里面，`@InjectMocks private FooImpl foo;`、`MockitoAnnotations.initMocks(this);`

例子3: 配合Spring Test

源代码Spring_1_Test:

```
1.  @ContextConfiguration(classes = FooImpl.class)
2.  @TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
3.  public class Spring_1_Test extends AbstractTestNGSpringContextTests {
4.
5.    @MockBean
6.    private Bar bar;
7.
8.    @Autowired
9.    private Foo foo;
```

```
10.
11.    @Test
12.    public void testCheckCodeDuplicate() throws Exception {
13.
14.      when(bar.getAllCodes()).thenReturn(Collections.singleton("123"));
15.      assertEquals(foo.checkCodeDuplicate("123"), true);
16.
17.    }
18.
19.  }
```

要注意，如果要启用Spring和Mockito，必须添加这么一行：`@TestExecutionListeners(listeners = MockitoTestExecutionListener.class)`。

例子4：配合Spring Test（多层依赖）

当Bean存在这种依赖关系当时候：`LooImpl -> FooImpl -> Bar`，我们应该怎么测试呢？

按照Mock测试的原则，这个时候我们应该mock一个`Foo`对象，把这个注入到`LooImpl`对象里，就像例子3里的一样。

不过如果你不想mock `Foo`而是想mock `Bar`的时候，其实做法和前面也差不多，Spring会自动将mock Bar注入到`FooImpl`中，然后将`FooImpl`注入到`LooImpl`中。

源代码Spring_2_Test：

```
1.  @ContextConfiguration(classes = { FooImpl.class, LooImpl.class })
2.  @TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
3.  public class Spring_2_Test extends AbstractTestNGSpringContextTests {
4.
5.    @MockBean
6.    private Bar bar;
7.
8.    @Autowired
9.    private Loo loo;
10.
11.   @Test
12.   public void testCheckCodeDuplicate() throws Exception {
13.
14.     when(bar.getAllCodes()).thenReturn(Collections.singleton("123"));
15.     assertEquals(loo.checkCodeDuplicate("123"), true);
16.
```

```
17.    }
18.
19. }
```

也就是说，得益于Spring Test Framework，我们能够很方便地对依赖关系中任意层级的任意Bean做mock。

例子5: 配合Spring Boot Test

源代码Boot_1_Test:

```
1.  @SpringBoot_1_Test(classes = { FooImpl.class })
2.  @TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
3.  public class Boot_1_Test extends AbstractTestNGSpringContextTests {
4.
5.    @MockBean
6.    private Bar bar;
7.
8.    @Autowired
9.    private Foo foo;
10.
11.   @Test
12.   public void testCheckCodeDuplicate() throws Exception {
13.
14.     when(bar.getAllCodes()).thenReturn(Collections.singleton("123"));
15.     assertEquals(foo.checkCodeDuplicate("123"), true);
16.
17.   }
18.
19. }
```

参考文档

- Spring Framework Testing
- Spring Boot Testing
- Mockito

Chapter 4: 测试关系型数据库

Chapter 4: 测试关系型数据库

Spring Test Framework提供了对JDBC的支持，能够让我们很方便对关系型数据库做集成测试。

同时Spring Boot提供了和Flyway的集成支持，能够方便的管理开发过程中产生的SQL文件，配合Spring已经提供的工具能够更方便地在测试之前初始化数据库以及测试之后清空数据库。

本章节为了方便起见，本章节使用了H2作为测试数据库。

注意：在真实的开发环境中，集成测试用数据库应该和最终的生产数据库保持一致，这是因为不同数据库的对于SQL不是完全相互兼容的，如果不注意这一点，很有可能出现集成测试通过，但是上了生产环境却报错的问题。

因为是集成测试，所以我们使用了 `maven-failsafe-plugin` 来跑，它和 `maven-surefire-plugin` 的差别在于， `maven-failsafe-plugin` 只会搜索 `*IT.java` 来跑测试，而 `maven-surefire-plugin` 只会搜索 `*Test.java` 来跑测试。

如果想要在maven打包的时候跳过集成测试，只需要 `mvn clean install -DskipITs` 。

被测试类

先介绍一下被测试的类。

Foo.java:

```
1.  public class Foo {
2.
3.    private String name;
4.
5.    public String getName() {
6.      return name;
7.    }
8.
9.    public void setName(String name) {
10.     this.name = name;
11.   }
12. }
```

FooRepositoryImpl.java:

```
1.  @Repository
```

```
2.  public class FooRepositoryImpl implements FooRepository {
3.
4.    private JdbcTemplate jdbcTemplate;
5.
6.    @Override
7.    public void save(Foo foo) {
8.      jdbcTemplate.update("INSERT INTO FOO(name) VALUES (?)", foo.getName());
9.    }
10.
11.   @Override
12.   public void delete(String name) {
13.     jdbcTemplate.update("DELETE FROM FOO WHERE NAME = ?", name);
14.   }
15.
16.   @Autowired
17.   public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
18.     this.jdbcTemplate = jdbcTemplate;
19.   }
20.
21. }
```

例子1: 不使用Spring Testing提供的工具

Spring_1_IT_Configuration.java:

```
1.  @Configuration
2.  @ComponentScan(basePackageClasses = FooRepository.class)
3.  public class Spring_1_IT_Configuration {
4.
5.    @Bean(destroyMethod = "shutdown")
6.    public DataSource dataSource() {
7.
8.      return new EmbeddedDatabaseBuilder()
9.          .generateUniqueName(true)
10.         .setType(EmbeddedDatabaseType.H2)
11.         .setScriptEncoding("UTF-8")
12.         .ignoreFailedDrops(true)
13.         .addScript("classpath:me/chanjar/domain/foo-ddl.sql")
14.         .build();
15.   }
16.
```

```
17.   @Bean
18.   public JdbcTemplate jdbcTemplate() {
19.
20.     return new JdbcTemplate(dataSource());
21.
22.   }
23. }
```

在 `Spring_1_IT_Configuration` 中，我们定义了一个H2的DataSource Bean，并且构建了JdbcTemplate Bean。

注意看 `addScript("classpath:me/chanjar/domain/foo-ddl.sql")` 这句代码，我们让 `EmbeddedDatabase` 执行foo-ddl.sql脚本来建表：

```
1. CREATE TABLE FOO (
2.   name VARCHAR2(100)
3. );
```

Spring_1_IT.java:

```
1. @ContextConfiguration(classes = Spring_1_IT_Configuration.class)
2. public class Spring_1_IT extends AbstractTestNGSpringContextTests {
3.
4.   @Autowired
5.   private FooRepository fooRepository;
6.
7.   @Autowired
8.   private JdbcTemplate jdbcTemplate;
9.
10.  @Test
11.  public void testSave() {
12.
13.    Foo foo = new Foo();
14.    foo.setName("Bob");
15.    fooRepository.save(foo);
16.
17.    assertEquals(
18.        jdbcTemplate.queryForObject("SELECT count(*) FROM FOO", Integer.class),
19.        Integer.valueOf(1)
20.    );
21.
22.  }
```

```
23.
24.   @Test(dependsOnMethods = "testSave")
25.   public void testDelete() {
26.
27.     assertEquals(
28.         jdbcTemplate.queryForObject("SELECT count(*) FROM FOO", Integer.class),
29.         Integer.valueOf(1)
30.     );
31.
32.     Foo foo = new Foo();
33.     foo.setName("Bob");
34.     fooRepository.save(foo);
35.
36.     fooRepository.delete(foo.getName());
37.     assertEquals(
38.         jdbcTemplate.queryForObject("SELECT count(*) FROM FOO", Integer.class),
39.         Integer.valueOf(0)
40.     );
41.   }
42.
43. }
```

在这段测试代码里可以看到，我们分别测试了 `FooRepository` 的 `save` 和 `delete` 方法，并且利用 `JdbcTemplate` 来验证数据库中的结果。

例子2: 使用Spring Testing提供的工具

在这个例子里，我们会使用JdbcTestUtils来辅助测试。

Spring_2_IT_Configuration.java:

```
1.  @Configuration
2.  @ComponentScan(basePackageClasses = FooRepository.class)
3.  public class Spring_2_IT_Configuration {
4.
5.    @Bean
6.    public DataSource dataSource() {
7.
8.      EmbeddedDatabase db = new EmbeddedDatabaseBuilder()
9.          .generateUniqueName(true)
10.         .setType(EmbeddedDatabaseType.H2)
11.         .setScriptEncoding("UTF-8")
```

```
        .ignoreFailedDrops(true)
        .addScript("classpath:me/chanjar/domain/foo-ddl.sql")
        .build();
    return db;
  }

  @Bean
  public JdbcTemplate jdbcTemplate() {

    return new JdbcTemplate(dataSource());

  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    return new DataSourceTransactionManager(dataSource());
  }

}
```

这里和例子1的区别在于，我们提供了一个 `PlatformTransactionManager` Bean，这是因为在下面的测试代码里的 `AbstractTransactionalTestNGSpringContextTests` 需要它。

Spring_2_IT.java:

```
@ContextConfiguration(classes = Spring_2_IT_Configuration.class)
public class Spring_2_IT extends AbstractTransactionalTestNGSpringContextTests
{

  @Autowired
  private FooRepository fooRepository;

  @Test
  public void testSave() {

    Foo foo = new Foo();
    foo.setName("Bob");
    fooRepository.save(foo);

    assertEquals(countRowsInTable("FOO"), 1);
    countRowsInTableWhere("FOO", "name = 'Bob'");
  }
```

```
17.
18.   @Test(dependsOnMethods = "testSave")
19.   public void testDelete() {
20.
21.     assertEquals(countRowsInTable("FOO"), 0);
22.
23.     Foo foo = new Foo();
24.     foo.setName("Bob");
25.     fooRepository.save(foo);
26.
27.     fooRepository.delete(foo.getName());
28.     assertEquals(countRowsInTable("FOO"), 0);
29.
30.   }
31.
32. }
```

在这里我们使用 `countRowsInTable("FOO")` 来验证数据库结果，这个方法是 `AbstractTransactionalTestNGSpringContextTests` 对 `JdbcTestUtils` 的代理。

而且要注意的是，每个测试方法在执行完毕后，会自动rollback，所以在 `testDelete` 的第一行里，我们 `assertEquals(countRowsInTable("FOO"), 0)` ，这一点和例子1里是不同的。

更多关于Spring Testing Framework与Transaction相关的信息，可以见Spring官方文档 Transaction management。

例子3: 使用Spring Boot

Boot_1_IT.java:

```
1. @SpringBootTest
2. @SpringBootApplication(scanBasePackageClasses = FooRepository.class)
3. public class Boot_1_IT extends AbstractTransactionalTestNGSpringContextTests {
4.
5.   @Autowired
6.   private FooRepository fooRepository;
7.
8.   @Test
9.   public void testSave() {
10.
11.     Foo foo = new Foo();
12.     foo.setName("Bob");
```

```
13.     fooRepository.save(foo);
14.
15.     assertEquals(countRowsInTable("FOO"), 1);
16.     countRowsInTableWhere("FOO", "name = 'Bob'");
17.   }
18.
19.   @Test(dependsOnMethods = "testSave")
20.   public void testDelete() {
21.
22.     assertEquals(countRowsInTable("FOO"), 0);
23.
24.     Foo foo = new Foo();
25.     foo.setName("Bob");
26.     fooRepository.save(foo);
27.
28.     fooRepository.delete(foo.getName());
29.     assertEquals(countRowsInTable("FOO"), 0);
30.
31.   }
32.
33.   @AfterTest
34.   public void cleanDb() {
35.     flyway.clean();
36.   }
37.
38. }
```

因为使用了Spring Boot来做集成测试，得益于其AutoConfiguration机制，不需要自己构建 `DataSource` 、 `JdbcTemplate` 和 `PlatformTransactionManager` 的Bean。

并且因为我们已经将 `flyway-core` 添加到了maven依赖中，Spring Boot会利用flyway来帮助我们初始化数据库，我们需要做的仅仅是将sql文件放到classpath的 `db/migration` 目录下：

`V1.0.0__foo-ddl.sql` ：

```
1.
2. CREATE TABLE FOO (
3.   name VARCHAR2(100)
4. );
```

而且在测试最后，我们利用flyway清空了数据库：

```
@AfterTest
public void cleanDb() {
  flyway.clean();
}
```

使用flyway有很多好处:

1. 每个sql文件名都规定了版本号
2. flyway按照版本号顺序执行
3. 在开发期间，只需要将sql文件放到db/migration目录下就可以了，不需要写类似`EmbeddedDatabaseBuilder.addScript()`这样的代码
4. 基于以上三点，就能够将数据库初始化SQL语句也纳入到集成测试中来，保证代码配套的SQL语句的正确性
5. 可以帮助你清空数据库，这在你使用非内存数据库的时候非常有用，因为不管测试前还是测试后，你都需要一个干净的数据库

参考文档

本章节涉及到的Spring Testing Framework JDBC、SQL相关的工具:

- Transaction management
- Executing SQL scripts

和flyway相关的:

- flyway的官方文档
- flway和spring boot的集成

Chapter 5: 测试Spring MVC

Chapter 5: 测试Spring MVC

Spring Testing Framework提供了Spring MVC Test Framework，能够很方便的来测试Controller。同时Spring Boot也提供了Auto-configured Spring MVC tests更进一步简化了测试需要的配置工作。

本章节将分别举例说明在不使用Spring Boot和使用Spring Boot下如何对Spring MVC进行测试。

例子1: Spring

测试Spring MVC的关键是使用 `MockMvc` 对象，利用它我们能够在不需启动Servlet容器的情况下测试Controller的行为。

源代码SpringMvc_1_Test.java:

```
1.  @EnableWebMvc
2.  @WebAppConfiguration
3.  @ContextConfiguration(classes = { FooController.class, FooImpl.class })
4.  public class SpringMvc_1_Test extends AbstractTestNGSpringContextTests {
5.
6.    @Autowired
7.    private WebApplicationContext wac;
8.
9.    private MockMvc mvc;
10.
11.   @BeforeMethod
12.   public void prepareMockMvc() {
13.     this.mvc = webAppContextSetup(wac).build();
14.   }
15.
16.   @Test
17.   public void testController() throws Exception {
18.
19.     this.mvc.perform(get("/foo/check-code-dup").param("code", "123"))
20.         .andDo(print())
21.         .andExpect(status().isOk())
22.         .andExpect(content().string("true"));
23.
24.   }
25.
26. }
```

在这段代码里，主要有三个步骤：

1. 将测试类标记为 `@WebAppConfiguration`
2. 通过 `webAppContextSetup(wac).build()` 构建 `MockMvc`
3. 利用 `MockMvc` 对结果进行判断

例子2: Spring + Mock

在例子1里， `FooController` 使用了一个实体 `FooImpl` 的Bean，实际上我们也可以提供一个 `Foo` 的mock bean来做测试，这样就能够更多的控制测试过程。如果你还不知道Mock那么请看 Chapter 3: 使用Mockito。

源代码SpringMvc_2_Test.java:

```
1.  @EnableWebMvc
2.  @WebAppConfiguration
3.  @ContextConfiguration(classes = { FooController.class })
4.  @TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
5.  public class SpringMvc_2_Test extends AbstractTestNGSpringContextTests {
6.
7.    @Autowired
8.    private WebApplicationContext wac;
9.
10.   @MockBean
11.   private Foo foo;
12.
13.   private MockMvc mvc;
14.
15.   @BeforeMethod
16.   public void prepareMockMvc() {
17.     this.mvc = webAppContextSetup(wac).build();
18.   }
19.
20.   @Test
21.   public void testController() throws Exception {
22.
23.     when(foo.checkCodeDuplicate(anyString())).thenReturn(true);
24.
25.     this.mvc.perform(get("/foo/check-code-dup").param("code", "123"))
26.         .andDo(print())
27.         .andExpect(status().isOk())
```

```
28.         .andExpect(content().string("true"));
29. 
30.   }
31. 
32. }
```

例子3: Spring Boot

Spring Boot提供了 `@WebMvcTest` 更进一步简化了对于Spring MVC的测试，我们提供了对应例子1的Spring Boot版本。

源代码BootMvc_1_Test.java:

```
1.  @WebMvcTest
2.  @ContextConfiguration(classes = { FooController.class, FooImpl.class })
3.  public class BootMvc_1_Test extends AbstractTestNGSpringContextTests {
4.  
5.    @Autowired
6.    private MockMvc mvc;
7.  
8.    @Test
9.    public void testController() throws Exception {
10. 
11.     this.mvc.perform(get("/foo/check-code-dup").param("code", "123"))
12.         .andDo(print())
13.         .andExpect(status().isOk())
14.         .andExpect(content().string("true"));
15. 
16.   }
17. 
18. }
```

在这里，我们不需要自己构建 `MockMvc` ，直接使用 `@Autowired` 注入就行了，是不是很方便？

例子4: Spring Boot + Mock

这个是对应例子2的Spring Boot版本，源代码BootMvc_2_Test.java:

```
1.  @WebMvcTest
2.  @ContextConfiguration(classes = { FooController.class })
3.  @TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
```

```
4.  public class BootMvc_2_Test extends AbstractTestNGSpringContextTests {
5.
6.    @Autowired
7.    private MockMvc mvc;
8.
9.    @MockBean
10.   private Foo foo;
11.
12.   @Test
13.   public void testController() throws Exception {
14.
15.     when(foo.checkCodeDuplicate(anyString())).thenReturn(true);
16.
17.     this.mvc.perform(get("/foo/check-code-dup").param("code", "123"))
18.         .andDo(print())
19.         .andExpect(status().isOk())
20.         .andExpect(content().string("true"));
21.
22.   }
23.
24. }
```

参考文档

- Loading a WebApplicationContext
- Spring MVC Test Framework
- Spring MVC Official Sample Tests
- Spring MVC showcase - with full mvc test
- Auto-configured Spring MVC tests
- Spring Framework Testing
- Spring Boot Testing
- Spring Guides - Testing the Web Layer

Chapter 6: 测试AOP

Chapter 6: 测试AOP

Spring提供了一套AOP工具，但是当你把各种Aspect写完之后，如何确定这些Aspect都正确的应用到目标Bean上了呢？本章将举例说明如何对Spring AOP做测试。

首先先来看我们事先定义的Bean以及Aspect。

FooServiceImpl:

```
1.  @Component
2.  public class FooServiceImpl implements FooService {
3.
4.    private int count;
5.
6.    @Override
7.    public int incrementAndGet() {
8.      count++;
9.      return count;
10.   }
11.
12. }
```

FooAspect:

```
1.  @Component
2.  @Aspect
3.  public class FooAspect {
4.
5.    @Pointcut("execution(*
    me.chanjar.aop.service.FooServiceImpl.incrementAndGet())")
6.    public void pointcut() {
7.    }
8.
9.    @Around("pointcut()")
10.   public int changeIncrementAndGet(ProceedingJoinPoint pjp) {
11.     return 0;
12.   }
13.
14. }
```

可以看到 `FooAspect` 会修改 `FooServiceImpl.incrementAndGet` 方法的返回值，使其返回0。

例子1：测试FooService的行为

最简单的测试方法就是直接调用 `FooServiceImpl.incrementAndGet` ，看看它是否使用返回0。

SpringAop_1_Test:

```
1.  @ContextConfiguration(classes = { SpringAopTest.class, AopConfig.class })
2.  public class SpringAop_1_Test extends AbstractTestNGSpringContextTests {
3.
4.    @Autowired
5.    private FooService fooService;
6.
7.    @Test
8.    public void testFooService() {
9.
10.     assertNotEquals(fooService.getClass(), FooServiceImpl.class);
11.
12.     assertTrue(AopUtils.isAopProxy(fooService));
13.     assertTrue(AopUtils.isCglibProxy(fooService));
14.
15.     assertEquals(AopProxyUtils.ultimateTargetClass(fooService),
    FooServiceImpl.class);
16.
17.     assertEquals(AopTestUtils.getTargetObject(fooService).getClass(),
    FooServiceImpl.class);
18.     assertEquals(AopTestUtils.getUltimateTargetObject(fooService).getClass(),
    FooServiceImpl.class);
19.
20.     assertEquals(fooService.incrementAndGet(), 0);
21.     assertEquals(fooService.incrementAndGet(), 0);
22.
23.   }
24.
25. }
```

先看这段代码：

```
1.  assertNotEquals(fooService.getClass(), FooServiceImpl.class);
2.
3.  assertTrue(AopUtils.isAopProxy(fooService));
4.  assertTrue(AopUtils.isCglibProxy(fooService));
5.
```

```
6.  assertEquals(AopProxyUtils.ultimateTargetClass(fooService),
    FooServiceImpl.class);
7.
8.  assertEquals(AopTestUtils.getTargetObject(fooService).getClass(),
    FooServiceImpl.class);
9.  assertEquals(AopTestUtils.getUltimateTargetObject(fooService).getClass(),
    FooServiceImpl.class);
```

这些是利用Spring提供的AopUtils、AopTestUtils和AopProxyUtils来判断 `FooServiceImpl` Bean是否被代理了（Spring AOP的实现是通过动态代理来做的）。

但是证明 `FooServiceImpl` Bean被代理并不意味着 `FooAspect` 生效了（假设此时有多个 `@Aspect` ），那么我们还需要验证 `FooServiceImpl.incrementAndGet` 的行为：

```
1.  assertEquals(fooService.incrementAndGet(), 0);
2.  assertEquals(fooService.incrementAndGet(), 0);
```

例子2：测试FooAspect的行为

但是总有一些时候我们是无法通过例子1的方法来测试Bean是否被正确的advised的：

1. advised方法没有返回值
2. Aspect不会修改advised方法的返回值（比如：做日志）

那么这个时候怎么测试呢？此时我们就需要用到Mockito的Spy方法结合Spring Testing工具来测试。

SpringAop_2_Test:

```
1.  @ContextConfiguration(classes = { SpringAop_2_Test.class, AopConfig.class })
2.  @TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
3.  public class SpringAop_2_Test extends AbstractTestNGSpringContextTests {
4.
5.    @SpyBean
6.    private FooAspect fooAspect;
7.
8.    @Autowired
9.    private FooService fooService;
10.
11.   @Test
12.   public void testFooService() {
13.
```

```
    // ...
    verify(fooAspect, times(2)).changeIncrementAndGet(any());

  }

}
```

这段代码和例子1有三点区别:

1. 启用了`MockitoTestExecutionListener`，这样能够开启Mockito的支持（回顾一下Chapter 3: 使用Mockito）
2. `@SpyBean private FooAspect fooAspect`，这样能够声明一个被Mockito.spy过的Bean
3. `verify(fooAspect, times(2)).changeIncrementAndGet(any())`，使用Mockito测试`FooAspect.changeIncrementAndGet`是否被调用了两次

上面的测试代码测试的是`FooAspect`的行为，而不是`FooServiceImpl`的行为，这种测试方法更为通用。

例子3: Spring Boot的例子

上面两个例子使用的是Spring Testing工具，下面举例Spring Boot Testing工具如何测AOP（其实大同小异）:

SpringBootAopTest:

```
@SpringBootTest(classes = { SpringBootAopTest.class, AopConfig.class })
@TestExecutionListeners(listeners = MockitoTestExecutionListener.class)
public class SpringBootAopTest extends AbstractTestNGSpringContextTests {

  @SpyBean
  private FooAspect fooAspect;

  @Autowired
  private FooService fooService;

  @Test
  public void testFooService() {

    // ...
    verify(fooAspect, times(2)).changeIncrementAndGet(any());

  }
```

```
18.
19. }
```

参考文档

- Aspect Oriented Programming with Spring
- AopUtils
- AopTestUtils
- AopProxyUtils
- spring源码EnableAspectJAutoProxyTests
- spring源码AbstractAspectJAdvisorFactoryTests

Chapter 7: 测试@Configuration

Chapter 7: 测试@Configuration

在Spring引入Java Config机制之后，我们会越来越多的使用@Configuration来注册Bean，并且Spring Boot更广泛地使用了这一机制，其提供的大量Auto Configuration大大简化了配置工作。那么问题来了，如何确保@Configuration和Auto Configuration按照预期运行呢，是否正确地注册了Bean呢？本章举例测试@Configuration和Auto Configuration的方法（因为Auto Configuration也是@Configuration，所以测试方法是一样的）。

例子1：测试@Configuration

我们先写一个简单的@Configuration：

```
1.  @Configuration
2.  public class FooConfiguration {
3.
4.    @Bean
5.    public Foo foo() {
6.      return new Foo();
7.    }
8.
9.  }
```

然后看FooConfiguration是否能够正确地注册Bean：

```
1.  public class FooConfigurationTest {
2.
3.    private AnnotationConfigApplicationContext context;
4.
5.    @BeforeMethod
6.    public void init() {
7.      context = new AnnotationConfigApplicationContext();
8.    }
9.
10.   @AfterMethod(alwaysRun = true)
11.   public void reset() {
12.     context.close();
13.   }
14.
15.   @Test
16.   public void testFooCreation() {
```

```
        context.register(FooConfiguration.class);
        context.refresh();
        assertNotNull(context.getBean(Foo.class));
    }

}
```

注意上面代码中关于Context的代码：

1. 首先，我们构造一个Context
2. 然后，注册FooConfiguration
3. 然后，refresh Context
4. 最后，在测试方法结尾close Context

如果你看Spring Boot中关于@Configuration测试的源代码会发现和上面的代码有点不一样：

```
public class DataSourceAutoConfigurationTests {

    private final AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();

    @Before
    public void init() {
        EmbeddedDatabaseConnection.override = null;
        EnvironmentTestUtils.addEnvironment(this.context,
                "spring.datasource.initialize:false",
                "spring.datasource.url:jdbc:hsqldb:mem:testdb-" + new
Random().nextInt());
    }

    @After
    public void restore() {
        EmbeddedDatabaseConnection.override = null;
        this.context.close();
    }
```

这是因为Spring和Spring Boot都是用JUnit做测试的，而JUnit的特性是每次执行测试方法前，都会new一个测试类实例，而TestNG是在共享同一个测试类实例的。

例子2：测试@Conditional

Spring Framework提供了一种可以条件控制@Configuration的机制，即只在满足某条件的情况下

才会导入@Configuration，这就是@Conditional。

下面我们来对@Conditional做一些测试，首先我们自定义一个Condition FooConfiguration：

```
1.  @Configuration
2.  public class FooConfiguration {
3.
4.    @Bean
5.    @Conditional(FooCondition.class)
6.    public Foo foo() {
7.      return new Foo();
8.    }
9.
10.   public static class FooCondition implements Condition {
11.
12.     @Override
13.     public boolean matches(ConditionContext context, AnnotatedTypeMetadata
    metadata) {
14.       if (context.getEnvironment() != null) {
15.         Boolean property = context.getEnvironment().getProperty("foo.create",
    Boolean.class);
16.         return Boolean.TRUE.equals(property);
17.       }
18.       return false;
19.     }
20.
21.   }
22. }
```

该Condition判断Environment中是否有 `foo.create=true` 。

如果我们要测试这个Condition，那么就必须往Environment里添加相关property才可以，在这里我们测试了三种情况：

1. 没有配置 `foo.create=true`
2. 配置 `foo.create=true`
3. 配置 `foo.create=false`

FooConfigurationTest：

```
1.  public class FooConfigurationTest {
2.
3.    private AnnotationConfigApplicationContext context;
```

```
4.
5.    @BeforeMethod
6.    public void init() {
7.      context = new AnnotationConfigApplicationContext();
8.    }
9.
10.   @AfterMethod(alwaysRun = true)
11.   public void reset() {
12.     context.close();
13.   }
14.
15.   @Test(expectedExceptions = NoSuchBeanDefinitionException.class)
16.   public void testFooCreatePropertyNull() {
17.     context.register(FooConfiguration.class);
18.     context.refresh();
19.     context.getBean(Foo.class);
20.   }
21.
22.   @Test
23.   public void testFooCreatePropertyTrue() {
24.     context.getEnvironment().getPropertySources().addLast(
25.         new MapPropertySource("test", Collections.singletonMap("foo.create",
    "true"))
26.     );
27.     context.register(FooConfiguration.class);
28.     context.refresh();
29.     assertNotNull(context.getBean(Foo.class));
30.   }
31.
32.   @Test(expectedExceptions = NoSuchBeanDefinitionException.class)
33.   public void testFooCreatePropertyFalse() {
34.     context.getEnvironment().getPropertySources().addLast(
35.         new MapPropertySource("test", Collections.singletonMap("foo.create",
    "false"))
36.     );
37.     context.register(FooConfiguration.class);
38.     context.refresh();
39.     assertNotNull(context.getBean(Foo.class));
40.   }
41.
42. }
```

注意我们用以下方法来给Environment添加property:

```
context.getEnvironment().getPropertySources().addLast(
  new MapPropertySource("test", Collections.singletonMap("foo.create", "true"))
);
```

所以针对@Conditional和其对应的Condition的测试的根本就是给它不一样的条件，判断其行为是否正确，在这个例子里我们的Condition比较简单，只是判断是否存在某个property，如果复杂Condition的话，测试思路也是一样的。

例子3：测试@ConditionalOnProperty

Spring framework只提供了@Conditional，Spring boot对这个机制做了扩展，提供了更为丰富的@ConditionalOn*，这里我们以@ConditionalOnProperty举例说明。

先看FooConfiguration:

```
@Configuration
public class FooConfiguration {

  @Bean
  @ConditionalOnProperty(prefix = "foo", name = "create", havingValue = "true")
  public Foo foo() {
    return new Foo();
  }

}
```

FooConfigurationTest:

```
public class FooConfigurationTest {

  private AnnotationConfigApplicationContext context;

  @BeforeMethod
  public void init() {
    context = new AnnotationConfigApplicationContext();
  }

  @AfterMethod(alwaysRun = true)
  public void reset() {
```

```
    context.close();
  }

  @Test(expectedExceptions = NoSuchBeanDefinitionException.class)
  public void testFooCreatePropertyNull() {
    context.register(FooConfiguration.class);
    context.refresh();
    context.getBean(Foo.class);
  }

  @Test
  public void testFooCreatePropertyTrue() {
    EnvironmentTestUtils.addEnvironment(context, "foo.create=true");
    context.register(FooConfiguration.class);
    context.refresh();
    assertNotNull(context.getBean(Foo.class));
  }

  @Test(expectedExceptions = NoSuchBeanDefinitionException.class)
  public void testFooCreatePropertyFalse() {
    EnvironmentTestUtils.addEnvironment(context, "foo.create=false");
    context.register(FooConfiguration.class);
    context.refresh();
    assertNotNull(context.getBean(Foo.class));
  }

}
```

这段测试代码和例子2的逻辑差不多，只不过例子2里使用了我们自己写的Condition，这里使用了Spring Boot提供的@ConditionalOnProperty。

并且利用了Spring Boot提供的EnvironmentTestUtils简化了给Environment添加property的工作：

```
EnvironmentTestUtils.addEnvironment(context, "foo.create=false");
```

例子4：测试Configuration Properties

Spring Boot还提供了类型安全的Configuration Properties，下面举例如何对其进行测试。

BarConfiguration：

```
@Configuration
@EnableConfigurationProperties(BarConfiguration.BarProperties.class)
public class BarConfiguration {

  @Autowired
  private BarProperties barProperties;

  @Bean
  public Bar bar() {
    return new Bar(barProperties.getName());
  }

  @ConfigurationProperties("bar")
  public static class BarProperties {

    private String name;

    public String getName() {
      return name;
    }

    public void setName(String name) {
      this.name = name;
    }
  }

}
```

BarConfigurationTest:

```
public class BarConfigurationTest {

  private AnnotationConfigApplicationContext context;

  @BeforeMethod
  public void init() {
    context = new AnnotationConfigApplicationContext();
  }

  @AfterMethod(alwaysRun = true)
  public void reset() {
    context.close();
```

```
13.   }
14.
15.   @Test
16.   public void testBarCreation() {
17.     EnvironmentTestUtils.addEnvironment(context, "bar.name=test");
18.     context.register(BarConfiguration.class,
    PropertyPlaceholderAutoConfiguration.class);
19.     context.refresh();
20.     assertEquals(context.getBean(Bar.class).getName(), "test");
21.   }
22.
23. }
```

注意到因为我们使用了Configuration Properties机制，需要注册PropertyPlaceholderAutoConfiguration，否则在BarConfiguration里无法注入BarProperties。

参考文档

- Conditionally include @Configuration classes or @Bean methods
- Condition annotations
- Type-safe Configuration Properties
- Spring Framework Testing
- Spring Boot Testing

Chapter 8: 共享测试配置

Chapter 8: 共享测试配置

在使用Spring Boot Testing工具中提到:

> 在测试代码之间尽量做到配置共用。
> ...
> 能够有效利用*Spring TestContext Framework*的缓存机制，*ApplicationContext*只会创建一次，后面的测试会直接用已创建的那个，加快测试代码运行速度。

本章将列举几种共享测试配置的方法

@Configuration

我们可以将测试配置放在一个@Configuration里，然后在测试@SpringBootTest或ContextConfiguration中引用它。

PlainConfiguration:

```
1. @SpringBootApplication(scanBasePackages = "me.chanjar.shareconfig")
2. public class PlainConfiguration {
3. }
```

FooRepositoryIT:

```
1. @SpringBootTest(classes = PlainConfiguration.class)
2. public class FooRepositoryIT extends ...
```

@Configuration on interface

也可以把@Configuration放到一个interface上。

PlainConfiguration:

```
1. @SpringBootApplication(scanBasePackages = "me.chanjar.shareconfig")
2. public interface InterfaceConfiguration {
3. }
```

FooRepositoryIT:

```
1. @SpringBootTest(classes = InterfaceConfiguration.class)
```

```
2.  public class FooRepositoryIT extends ...
```

Annotation

也可以利用Spring的Meta-annotations及自定义机制，提供自己的Annotation用在测试配置上。

PlainConfiguration:

```
1.  @Target(ElementType.TYPE)
2.  @Retention(RetentionPolicy.RUNTIME)
3.  @SpringBootApplication(scanBasePackages = "me.chanjar.shareconfig")
4.  public @interface AnnotationConfiguration {
5.  }
```

FooRepositoryIT:

```
1.  @SpringBootTest(classes = FooRepositoryIT.class)
2.  @AnnotationConfiguration
3.  public class FooRepositoryIT extends ...
```

参考文档

- Meta-annotations
- Meta-Annotation Support for Testing
- Spring Annotation Programming Model

附录I Spring Mock Objects

附录I Spring Mock Objects

Spring提供的Mock Objects有以下这些:

package org.springframework.mock.env

1. MockEnvironment
2. MockPropertySource

package org.springframework.mock.jndi

1. SimpleNamingContext
2. ExpectedLookupTemplate
3. SimpleNamingContextBuilder

package org.springframework.mock.web

1. DelegatingServletInputStream
2. DelegatingServletOutputStream
3. HeaderValueHolder
4. MockAsyncContext
5. MockBodyContent
6. MockExpressionEvaluator
7. MockFilterChain
8. MockFilterConfig
9. MockHttpServletRequest
10. MockHttpServletResponse
11. MockHttpSession
12. MockJspWriter
13. MockMultipartFile
14. MockMultipartHttpServletRequest
15. MockPageContext
16. MockRequestDispatcher
17. MockServletConfig
18. MockServletContext

19. MockSessionCookieConfig
20. PassThroughFilterChain

附录II Spring Test Utils

附录II Spring Test Utils

Spring Boot提供的Unit Test Utils有以下这些:

1. AopTestUtils
2. AssertionErrors
3. JsonExpectationsHelper
4. JsonPathExpectationsHelper
5. MetaAnnotationUtils
6. ReflectionTestUtils
7. XmlExpectationsHelper
8. XpathExpectationsHelper
9. ModelAndViewAssert
10. JdbcTestUtils
11. BeanFactoryAnnotationUtils，用它getBeanByQualifier特别有用

Spring Boot提供的Unit Test Utils有以下这些:

1. ConfigFileApplicationContextInitializer
2. EnvironmentTestUtils
3. OutputCapture
4. TestRestTemplate