基于junit5单元测试

## 一：环境准备

构建maven项目，添加pom.xml配置文件，将需要测试的项目包引入到工程中。如果是基于springBoot开发测试，则需要引入该框架相关的包，并在测试类中加上@SpringBootTest注解，详情见下面第二点。

## 二、基于SpringBoot进行测试

1、引入SpringBoot框架，添加到Maven配置文件中

<parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.1.8.RELEASE</version>  
</parent>

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

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

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
</build>

2、增加Application执行类

@SpringBootApplication  
public class Application {  
 public static void main(String[] args) {  
 SpringApplication.run(Application.class, args);  
 }  
}

3、在测试类名上增加@SpringBootTest注解，则标识为测试类

4、引入测试工具包

<dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter</artifactId>  
 <version>RELEASE</version>  
 <scope>test</scope>  
</dependency>

5、使用junit5测试框架进行测试（具体测试用例编写见后面部分）

## 三、普通单元测试

1、参考二.4中引入的测试包

2、需要对方法增加@Test注解来标识为测试用例，如：

@Test  
public void testTemplate(){  
 //TODO  
}

3、@BeforeEach标识方法在每个用例执行前执行一次（可以初始化每个用例）

4、@BeforeAll标识方法在所有用例执行前执行一次（初始化整个测试）

5、@AfterEach标识方法在每个用例执行后执行一次

6、@AfterAll标识方法在所有用例执行后执行一次

7、@DisplayName(“caseName”)执行用例后，用于输出台显示用例名称caseName

8、正常断言：assert boolean;

9、异常断言，当执行用例时，方法会抛异常，针对此类进行断言，如下面用例，当try catch用例执行成功时则用例执行失败，fail（“用例执行失败”），当抛异常时，则进行assert断言：

//fail包

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

……

@DisplayName("ExchangeName为空Fanout类型")  
@Test  
void testSendFanoutMessge1() {  
 DggProducerParams producerParams = DggProducerParams.*builder*()  
 .queueName(MyRabbitmqConstant.*exchangeNameFanOut*.name())  
 .routingKey(MyRabbitmqConstant.*routingKey*.name())  
 .data("ExchangeName为空Fanout类型")  
 .build();  
 try {  
 producer.sendFanoutMessage(producerParams);  
 *fail*("ExchangeName为空Fanout类型用例执行失败");  
 } catch (Exception e) {  
 assert e.getMessage().contains("交换器名称不能为空");  
 }  
}

## 四、参数化测试

1、最简单的参数源：Value Source，使用@ValueSource()+@ParameterizedTest注解

@DisplayName("精确验证手机号,正确的手机号格式")  
@ParameterizedTest  
@ValueSource(strings = {"134", "135", "136", "137", "138", "139", "147", "148", "150", "151", "152", "157","158", "159", "165", "172", "178", "182", "183", "184", "187", "188", "198", "130", "131", "132", "145", "146", "155","156", "166", "170", "171", "175", "176","185", "186", "133", "149", "153", "173", "174", "177", "180", "181","189", "191", "199"})  
void testDggRegexUtilIsMobileExactleTrue(String str) {  
 assert DggRegexUtil.isMobileExact(str + "00000000");  
}

上面实例是验证正常电话号码格式是否正确，按照最普通的用例，需要添加上面N多个case，若使用参数化，则只需要将需要执行的用例作为参数传入给case中，所有参数分别作为一条用例自动被执行。

2、枚举参数源：Enum Source，使用@ EnumSource()+@ParameterizedTest注解

@ParameterizedTest  
@EnumSource(DggDateFormatType.class)  
void testFormatCurrentDate(DggDateFormatType dggDateFormatType) {  
 assert DggDateFormatUtil.formatCurrentDate(dggDateFormatType).matches(getRegxStr(dggDateFormatType.getValue()));  
}

public enum DggDateFormatType {  
 MONTH\_FORMATTER("yyyy-MM"),  
 DATE\_FORMATTER("yyyy-MM-dd"),  
 DATETIME\_FORMATTER("yyyy-MM-dd HH:mm:ss"),  
 CONTINUITY\_MONTH\_FORMATTER("yyyyMM"),  
 CONTINUITY\_DATE\_FORMATTER("yyyyMMdd"),  
 CONTINUITY\_TIMESTAMP\_FORMATTER("yyyyMMddHHmmss"),  
 CONTINUITY\_MILLISECOND\_FORMATTER("yyyyMMddHHmmssSSS"),  
 BACKSLASH\_MONTH\_FORMATTER("yyyy/MM"),  
 BACKSLASH\_DATE\_FORMATTER("yyyy/MM/dd"),  
 BACKSLASH\_DATETIME\_FORMATTER("yyyy/MM/dd HH:mm:ss");  
 private String value;  
 public String getValue() {  
 return this.value;  
 }  
 private DggDateFormatType(String formatString) {  
 this.value = formatString;  
 }  
}

上面实例中方法formatCurrentDate：获取格式化后的日期字符串。而用例需要验证不同格式化后格式是否正确，公共库中有枚举库，则使用枚举数据源将所有格式都进行参数化测试。

PS：额外还可以制定约束条件或正则匹配来过滤需要执行的参数

3、java方法数据源：Method Source，使用@ MethodSource()+@ParameterizedTest注解

@DisplayName("getPropertyValueSize测试JSON获取属性值大小")  
@ParameterizedTest(name = "[case:{index}] id: {0} name: {1}")  
@MethodSource("GetPropertyValueSizeGenerator")  
public void testGetPropertyValueSize(int id, String name) {  
 assert id==DggJsonPathUtil.getPropertyValueSize(obj,name);  
}

//自定义额外方法，并添加需要测试的参数

static Stream<Arguments> GetPropertyValueSizeGenerator(){  
 return Stream.of(Arguments.of(-1, "num"), Arguments.of(6, "ab"));  
}

上面实例中，传入的参数格式如key value类型，这样可以对不同输入不同的断言结果分别进行用例测试

4、Argument Source、CSV Source、CSV File Source （略，本次单元测试没有涉及此类参数化）

## 五、多线程测试

额外引入多线程测试工具包

<dependency>  
 <groupId>fakepath</groupId>  
 <artifactId>groboutils-core</artifactId>  
 <version>5</version>  
 <scope>test</scope>  
</dependency>

如下所示，可以实现本地多线程操作某个方法并执行用例。

PS：线程外申明一个对象set，由于存放在方法区，所以是线程共享的，顾每个线程都能将当前线程生成的Uid指向set，而对象set值是存放在java栈中，每个线程独立的，为了安全，顾增加synchronizedSet。

@SuppressWarnings("unchecked")  
@DisplayName("多线程测试microGetUidData方法生成500万Uid ,验证Uid是否唯一")  
@Test  
void testMicroGetUidData() {  
 //构造Set存储多线程中生成的Uid   
 Set<Long> set = Collections.synchronizedSet(new HashSet<>());  
 TestRunnable runner = new TestRunnable() {  
 @Override  
 public void runTest() {

//每个线程需要执行的测试用例  
 for (int i = 1; i <= 100; i++) {  
 List<Long> listUidData = (List<Long>) uidGenerateTemplate.microGetUidData(501);  
 set.addAll(listUidData);  
 }  
 }  
 };  
 //线程数量  
 int runnerCount = 100;  
 TestRunnable[] trs = new TestRunnable[runnerCount];  
 for (int i = 0; i < runnerCount; i++) {  
 trs[i] = runner;  
 }  
 MultiThreadedTestRunner mttr = new MultiThreadedTestRunner(trs);  
 try {  
 mttr.runTestRunnables();  
 } catch (Throwable e) {  
 e.printStackTrace();  
 }  
 assert set.size() == 501\*100\*runnerCount;  
}