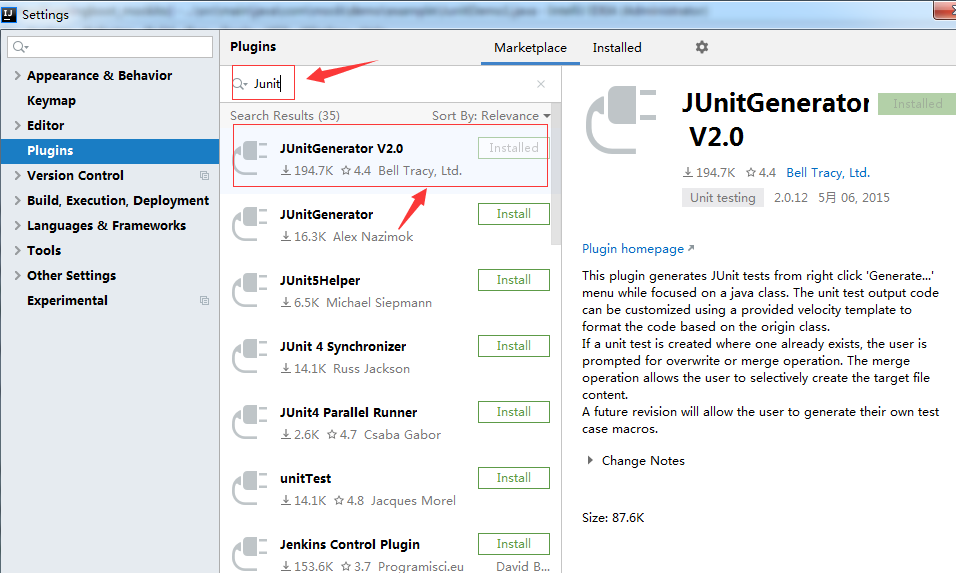
# 1 idea安装Junit插件

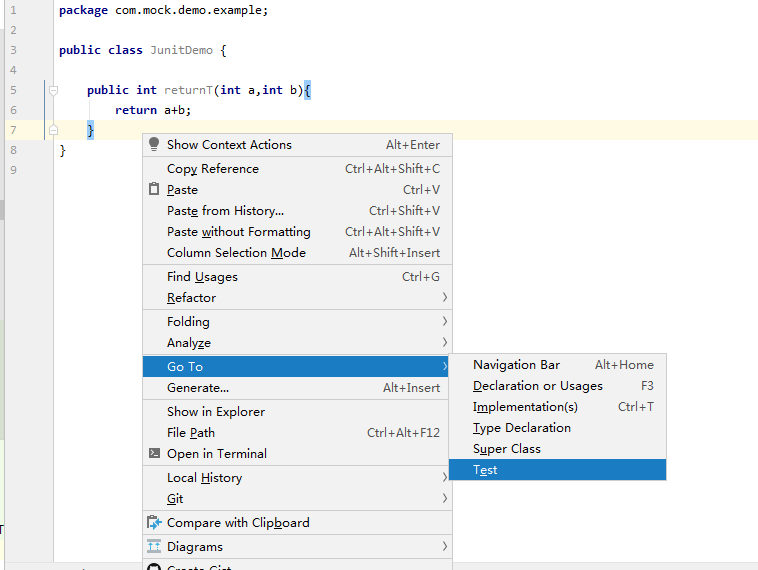
new—>setting—>Plugins

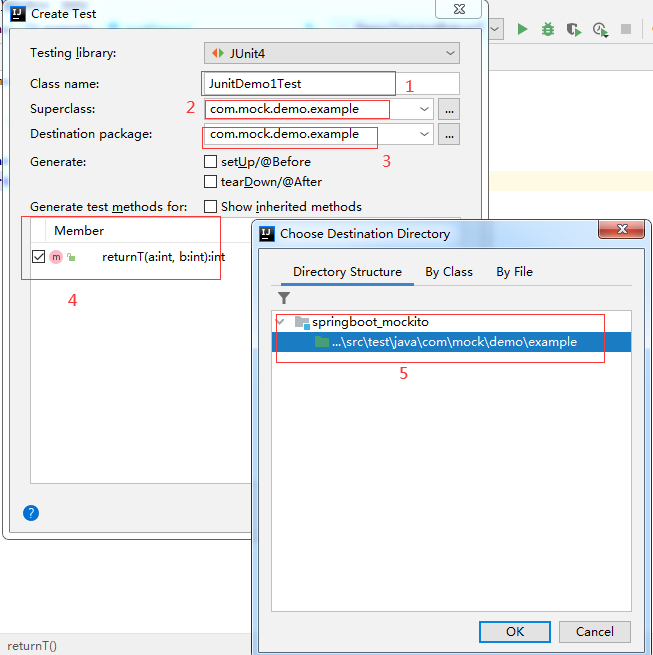


# 2 Junit

## Junit单元测试创建

右键单击需要创建单元测试的类名，选择【Go To】，然后选择【Test】即可。





|  |  |  |
| --- | --- | --- |
| 编号 | 名称 | 详情 |
| 1 | 生成的测试文件名 | 一般以被测文件名为基础， “被测文件名+Test” |
| 2 | 生成测试文件的位置 | 一般我们把测试代码都放在 “test” 文件夹里，此处选择 “/src/test/java/com/mock/demo/example” |
| 3 | 被测文件 | 这里需指定这个test case是测试哪个文件的 |
| 4 | 测试方法 | 勾选进行单元测试的方法 |

## **junit 基础注解**

常用注解：

@Test ：该注释表示，用其附着的公共无效方法（即用public修饰的void类型的方法 ）可以作为一个测试用例；

@Before ：该注释表示，用其附着的方法必须在类中的每个测试之前执行，以便执行测试某些必要的先决条件；

@BeforeClass ：该注释表示，用其附着的静态方法必须执行一次并在类的所有测试之前，发生这种情况时一般是测试计算共享配置方法，如连接到数据库；

@After ：该注释表示，用其附着的方法在执行每项测试后执行，如执行每一个测试后重置某些变量，删除临时变量等；

@AfterClass ：该注释表示，当需要执行所有的测试在JUnit测试用例类后执行，AfterClass注解可以使用以清理建立方法，如断开数据库连接，注意：附有此批注（类似于BeforeClass）的方法必须定义为静态；

@Ignore ：该注释表示，当想暂时禁用特定的测试执行可以使用忽略注释，每个被注解为@Ignore的方法将不被执行。

## **junit demo案例**

①@Before注解

public class JunitDemoTest {

List<String> list = new ArrayList<String>();

@Before

public void init(){

list.add("java 语言");

list.add("Python 语言");

}

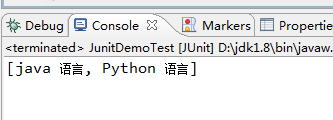
@Test

public void test3(){

System.out.println(list);

}

}



②@After

public class JunitDemoTest {

List<String> list = new ArrayList<String>();

@Before

public void init(){

list.add("java 语言");

list.add("Python 语言");

}

@Test

public void test(){

System.out.println("test"+list);

}

@After

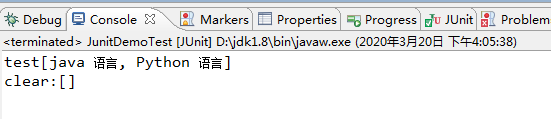
public void clear(){

list.clear();

System.out.println("clear:"+list);

}

}



③@Ignore

public class JunitDemoTest {

List<String> list = new ArrayList<String>();

@Before

public void init(){

list.add("java 语言");

list.add("Python 语言");

}

@Test

public void test(){

System.out.println("test"+list);

}

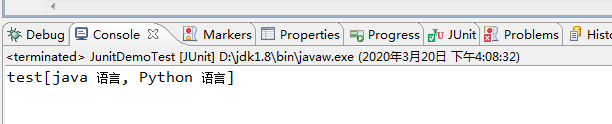
@Ignore

public void Ignore(){//方法未执行

System.out.println("Ignore"+list);

}

}



## **注意**

①测试方法上必须使用@Test进行修饰

②测试方法必须使用public void 进行修饰，不能带任何的参数

③新建一个源代码目录来存放我们的测试代码，即将测试代码和项目业务代码分开

④测试类所在的包名应该和被测试类所在的包名保持一致

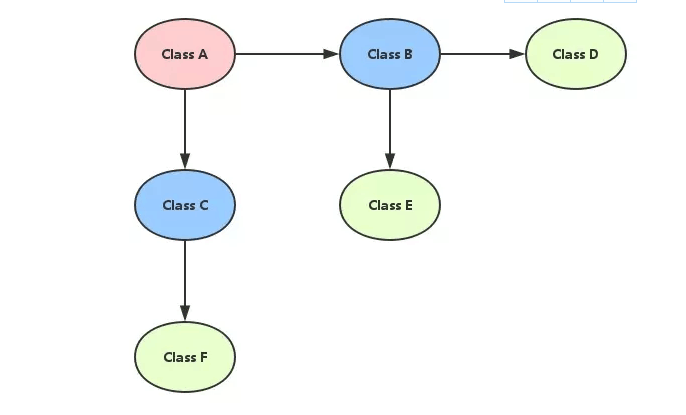
⑤测试单元中的每个方法必须可以独立测试，测试方法间不能有任何的依赖

⑥测试类使用Test作为类名的后缀（不是必须）

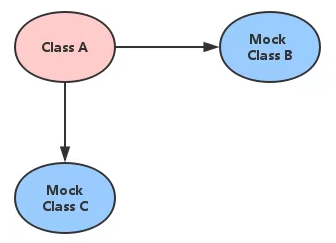
⑦测试方法使用test作为方法名的前缀（不是必须）

# 3 mock介绍

Mock测试就是在测试过程中,创建一个假的对象,避免你为了测试一个方法,却要自行构建整个Bean的依赖链。



如上图:类A需要调用类B和类C,而类B和类C又需要调用其他类如D、E、F等,假设类D是一个外部服务,那就会很难测,因为你的返回结果会直接的受外部服务影响,导致你的单元测试可能今天会过、但明天就过不了了。



当我们引入Mock测试时,就可以创建一个假的对象,替换掉真实的BeanB和C，这样在调用B、C的方法时,实际上就会去调用这个假的Moc对象的方法,而我们就可以自己设定这个Mock对象的参数和期望结果，让我们可以专注在测试当前的类A,而不会受到其他的外部服务影响,这样测试效率就能提高很多。

即Mock 测试就是在测试过程中,对于某些不容易构造或者不容易获取的对象,用一个虚拟的对象（Mock 对象）来创建（覆盖方法返回）以便测试的测试方法

# 4 mockito

Mockito 是一种 Java Mock 框架，他主要就是用来做 Mock 测试的，它可以模拟任何 Spring 管理的 Bean、模拟方法的返回值、模拟抛出异常等等，同时也会记录调用这些模拟方法的参数、调用顺序，从而可以校验出这个 Mock 对象是否有被正确的顺序调用，以及按照期望的参数被调用。

## **4.1 mockito引入**

1. springboot依赖

spring-boot-starter-test，类库默认包含junit与mockito包。

2.添加mockito包

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>2.15.0</version>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-inline</artifactId>

<version>2.15.0</version>

</dependency>

## 4.2 mockito案例demo

### 4.2.1 MockitoJUnitRunner类

@RunWith(MockitoJUnitRunner.class)

public class TestEmployeeService {

@InjectMocks

EmployeeServcie manager;

@Mock

EmployeeDao dao;

}

@RunWith(MockitoJUnitRunner.class)在单元测试的顶部使用，MockitoJUnitRunner类自动初始化所有@Mock和@InjectMocks批注的对象。

### 4.2.2 MockitoAnnotations.initMocks()方法

public class TestEmployeeService {

@InjectMocks

EmployeeServcie manager;

@Mock

EmployeeDao dao;

@Before

public void init() {

MockitoAnnotations.initMocks(this);

}

}

如果不使用@RunWith(MockitoJUnitRunner.class)注解方法时，则可以使用MockitoAnnotations.initMocks()方法, 而MockitoAnnotations.initMocks(this)需要在单元测试类的@Before方法中使用,在初始化junit测试时,对模拟对象初始化。

### 4.2.3 @Mock与@InjectMocks

@Mock

EmployeeDao dao;

@InjectMocks

EmployeeServcie manager;

@Mock注解为它创建标注有类模拟实现.

**@InjectMocks：创建一个实例，简单的说是这个Mock可以调用真实代码的方法，其余用@Mock（或@Spy）注解创建的mock将被注入到用该实例中。**

如：EmployeeServcie类添加@InjectMocks注解，mockito将为EmployeeServcie该类创建模拟对象并将其模拟依赖项EmployeeDao注入到该对象中。

### 4.2.4 @Mock与@Spy

1.Mock声明的对象，对函数的调用均执行mock，对象为虚假函数，不执行真正部分,运行时不进入具体的类中。

2.Spy声明的对象，对函数的调用均执行真正部分。



原因:

使用@Mock生成的类，所有方法都不是真实的方法，而且返回值都是null。

使用@Spy生成的类，所有方法都是真实方法，返回值都是和真实方法一样的。

### 4.2.5使用Mockito进行JUnit测试

测试service代码:

public class TestEmployeeService {

@InjectMocks

EmployeeServcie manager;

@Mock

EmployeeDao dao;

@Before

public void init() {

MockitoAnnotations.initMocks(this);

}

@Test

public void testGetEmployeeList(){

List<EmployeeVO> list = new ArrayList<EmployeeVO>();

EmployeeVO empOne = new EmployeeVO(1, "John", "John", "John@email.com");

EmployeeVO empTwo = new EmployeeVO(2, "Alex", "Alex", "Alex@yahoo.com");

EmployeeVO empThree = new EmployeeVO(3, "Steve", "Steve", "Steve@gmail.com");

list.add(empOne);

list.add(empTwo);

list.add(empThree);

Mockito.when(dao.getEmployeeList()).thenReturn(list);

//test

List<EmployeeVO> empList = manager.getEmployeeList();

// 比较测试结果

assertEquals(3, empList.size());

//进行测试验证

Mockito.verify(dao, times(1)).getEmployeeList();

}

@Test

public void testGetEmployeeById()

{

Mockito.when(dao.getEmployeeById(1)).thenReturn(new EmployeeVO(1,"admin1","admin2","user@email.com"));

EmployeeVO emp = manager.getEmployeeById(1);

// 比较测试结果

assertEquals("admin1", emp.getFirstName());

assertEquals("admin2", emp.getLastName());

assertEquals("user@email.com", emp.getEmail());

}

@Test

public void testAddEmployee()

{

EmployeeVO emp = new EmployeeVO(1,"admin1","admin2","user@email.com");

manager.addEmployee(emp);

Mockito.verify(dao, times(1)).addEmployee(emp);

}

}

①: Mockito.when(dao.getEmployeeList()).thenReturn(list)

dao.getEmployeeList()：调用的方法，list:设定结果集，当调用方法返回指定结果集则执行成功，反正执行失败。

②assertEquals(3, empList.size())

结果集比较，第一个表示期望值，第二表示实际值，当期望值与实际值不相等时，执行报错。

③Mockito.verify(dao, times(1)).addEmployee(emp)

验证dao层addEmployee(emp)方法被调用次数，当调用次数与times(n)中的n不符时，执行不通过。

# 5 单元测试之事务处理

单元测试涉及到脏数据的处理。

springboot中编写单元测试，在头部添加@Transactional注解，默认是事物回滚的，这样测试的脏数据不影响数据库。

运行下列代码：

@RunWith(SpringRunner.class)

@SpringBootTest

@Transactional

public class TitleServiceTest {

@Autowired

private TitleService titleService;

@Test

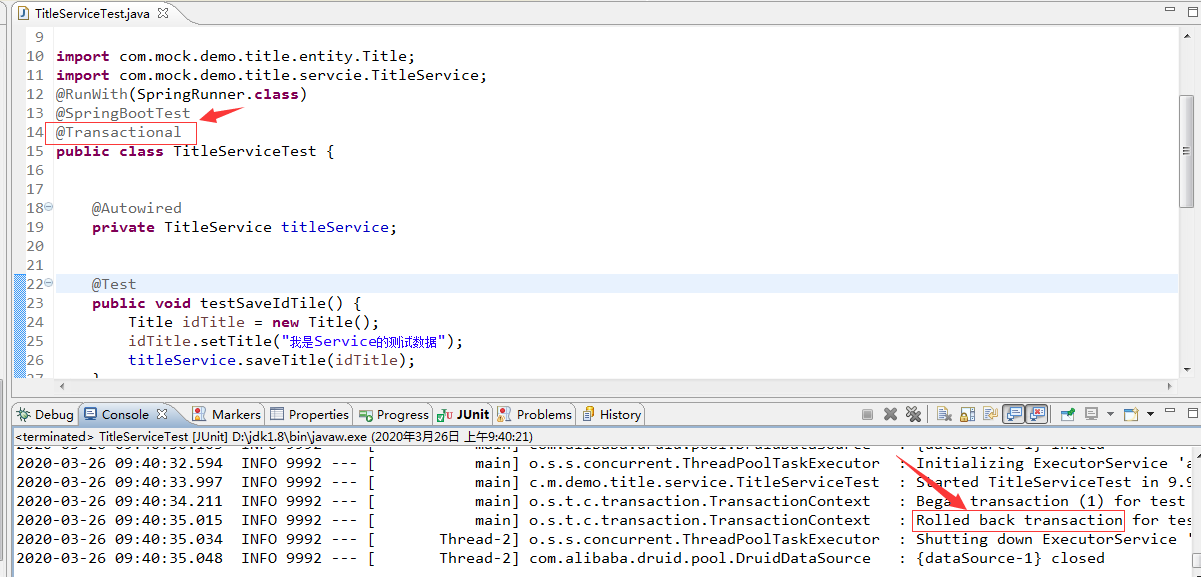
public void testSaveIdTile() {

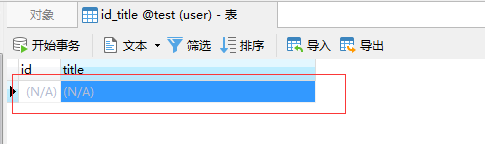
Title idTitle = new Title();

idTitle.setTitle("我是Service的测试数据");

titleService.saveTitle(idTitle);

}





测试结果数据未添加到数据库

在方法添加@Rollback(true)注解，默认是true，事务会回滚，false时事务不会回滚，数据会写到数据库中。

测试方法:

在测试方法上添加注解：

@RunWith(SpringRunner.class)

@SpringBootTest

@Transactional

public class TitleServiceTest {

@Autowired

private TitleService titleService;

@Test

@Rollback(false)

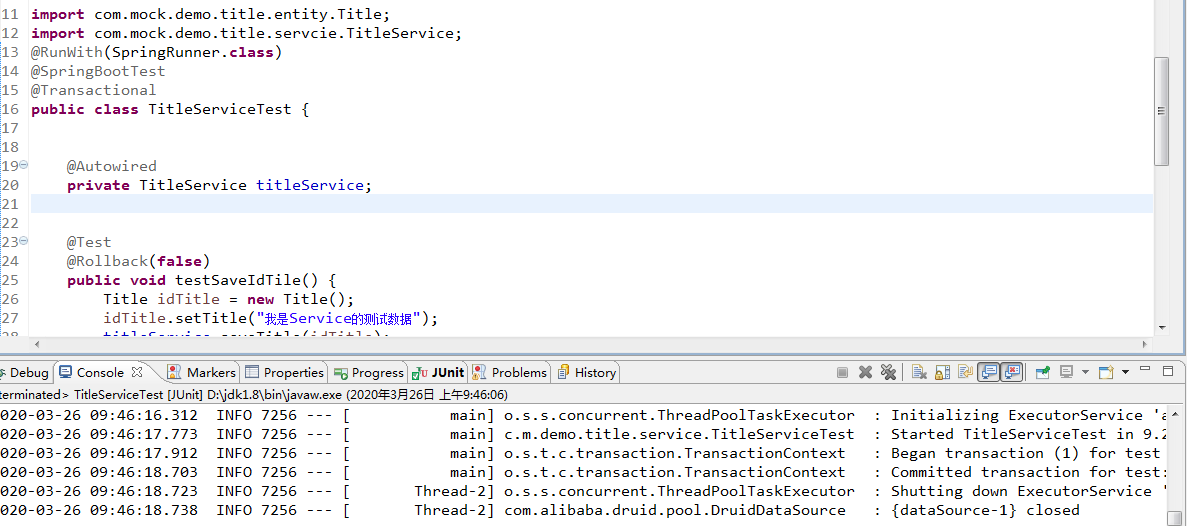
public void testSaveIdTile() {

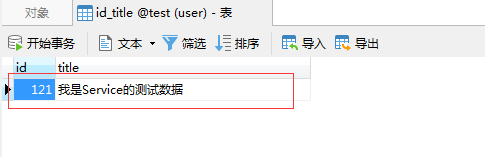
Title idTitle = new Title();

idTitle.setTitle("我是Service的测试数据");

titleService.saveTitle(idTitle);

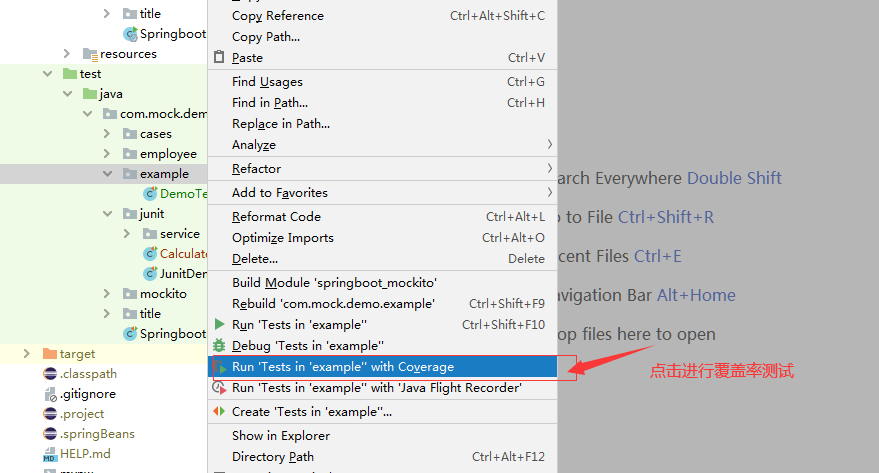
}

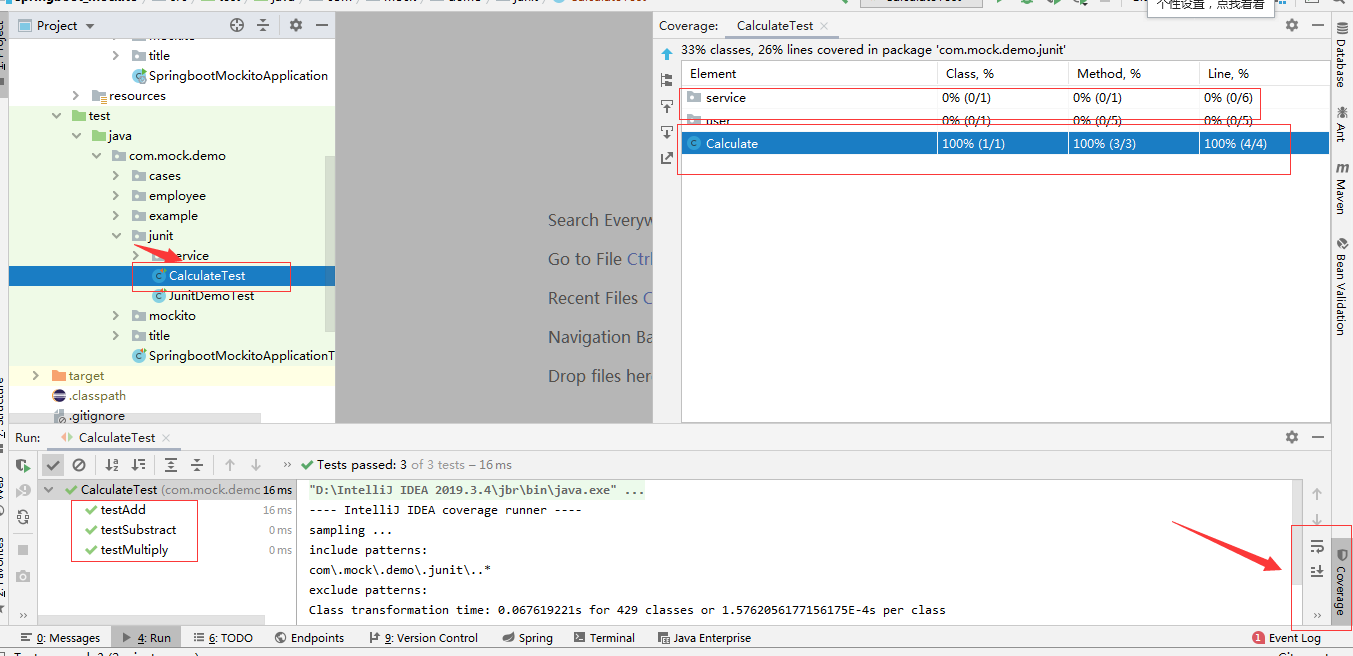




# 6 单元测试之覆盖率

## 1 idea自带覆盖率测试





## 2 jacoco

Pom.xml

<plugin>

<groupId>org.jacoco</groupId>

<artifactId>jacoco-maven-plugin</artifactId>

<version>0.7.9</version>

<configuration>

<!--指定生成 .exec文件的存放位置-->

<destFile>target/coverage-reports/jacoco-unit.exec</destFile>

<!--Jacoco 是根据 .exec文件生成最终的报告，所以需指定 .exec 的存放路径-->

<dataFile>target/coverage-reports/jacoco-unit.exec</dataFile>

</configuration>

<executions>

<execution>

<id>jacoco-initialize</id>

<goals>

<goal>prepare-agent</goal>

</goals>

</execution>

<execution>

<id>jacoco-site</id>

<phase>test</phase>

<goals>

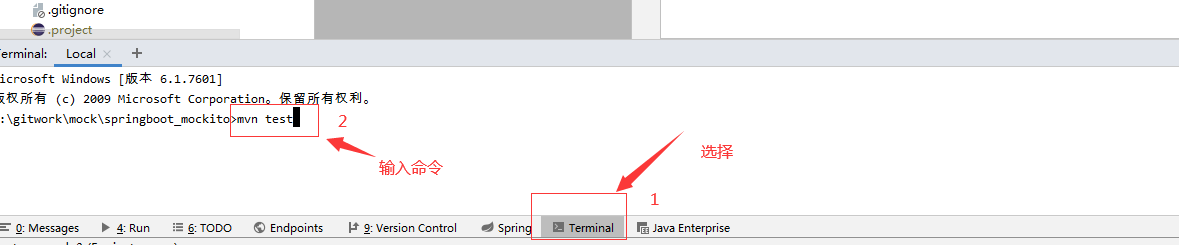
<goal>report</goal>

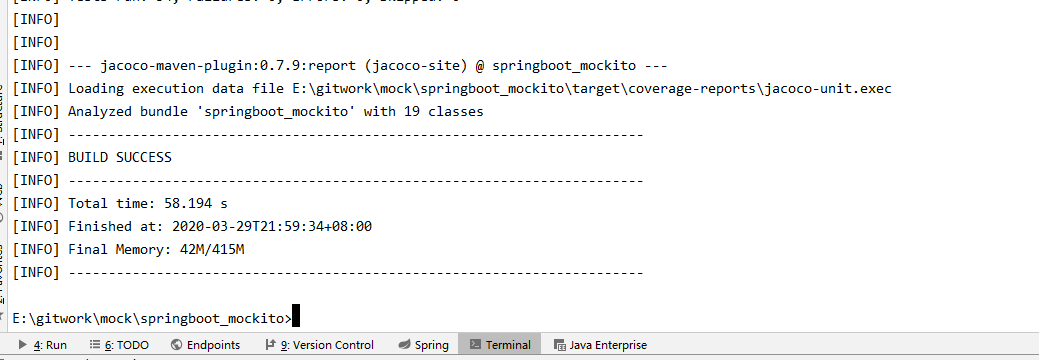
</goals>

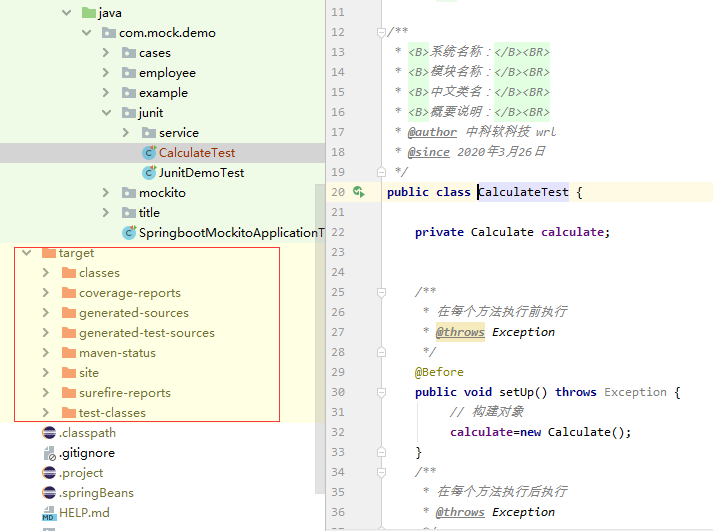
</execution>

</executions>

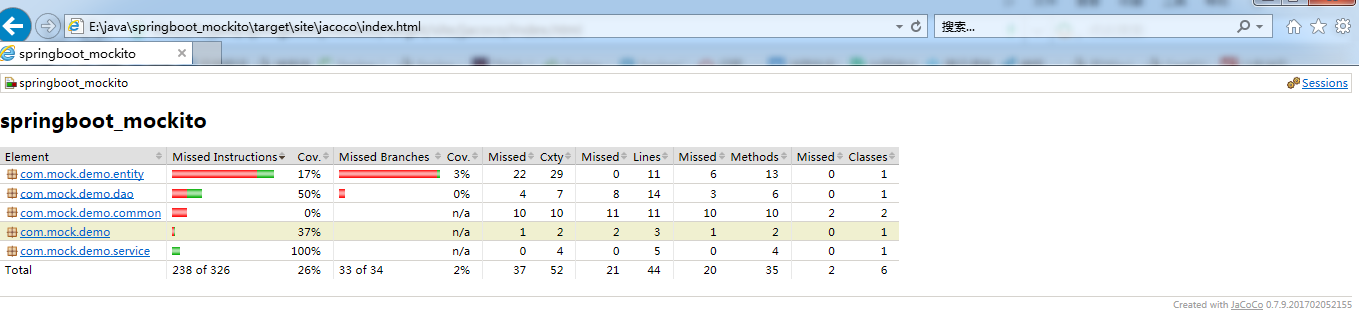
</plugin>







进入项目target目录的site中，访问index.html



进入项目target目录的site中，访问index.html

