Mockito单元测试

| **修订版本号** | **修订人** | **修订日期** | **修订描述** |
| --- | --- | --- | --- |
| V0.1 |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[1. 为什么要进行单元测试 3](#_Toc11536)

[2. 常用Junit基本介绍 3](#_Toc17328)

[2.1 Junit基本注解 3](#_Toc16726)

[2.2 junit用例 4](#_Toc13824)

[2.3 注意 6](#_Toc14497)

[3. Mockito介绍 7](#_Toc21142)

[3.1 什么是mock 7](#_Toc5150)

[3.2 mockito简介 8](#_Toc641)

[4. Maven依赖 8](#_Toc15059)

[4.1 springboot依赖 8](#_Toc29212)

[4.2 引入mockito包 9](#_Toc11124)

[5. Mockito测试方法 9](#_Toc8708)

[5.1 常用的Mockito方法 9](#_Toc22008)

[5.2 测试类 10](#_Toc13536)

[5.3 Mockito的使用 12](#_Toc8915)

[5.3.1 MockitoJUnitRunner类 12](#_Toc2091)

[5.3.2 MockitoAnnotations.initMocks()方法 12](#_Toc7420)

[5.3.3 @Mock与@InjectMocks 13](#_Toc22700)

[5.3.4 @Mock与@Spy 13](#_Toc7465)

[5.3.5 使用Mockito进行JUnit测试 15](#_Toc30519)

[5.3.6 注意 16](#_Toc2026)

[6. Mockito的限制 17](#_Toc30957)

[7. 事务处理 17](#_Toc32129)

[8. 单元测试覆盖率 20](#_Toc26141)

[8.1 EclEmma 20](#_Toc4529)

[8.1.1 安装EclEmma插件 20](#_Toc17767)

[8.1.2 覆盖率统计 21](#_Toc4831)

[8.2 Jacoco 24](#_Toc3292)

[8.2.1 spring Boot集成jacoco 24](#_Toc25385)

[8.2.2 Jacoco测试覆盖率 25](#_Toc4392)

# 为什么要进行单元测试

单元测试（unit testing），是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如C语言中单元指一个函数，Java里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

# 常用Junit基本介绍

## Junit基本注解

@RunWith：指定使用的单元测试执行类。

@Before：初始化方法。

@After：释放资源。

@Test：测试方法，可以测试期望异常和超时时间。

@Ignore：测试方法被忽略，不在执行。

@BeforeClass：在所有测试方法执行前执行一次，一般在其中写上整体初始化的代码,且必须为static void。

@AfterClass：在所有测试方法后执行一次，一般在其中写上销毁和释放资源的代码,且必须为static void。

|  |  |
| --- | --- |
| **@BeforeClass and @AfterClass** | **@Before and @After** |
| 在一个类中只可以出现一次 | 在一个类中可以出现多次，即可以在多个方法的声明前加上这两个Annotaion标签，执行顺序不确定 |
| 方法名不做限制 | 方法名不做限制 |
| 在类中只运行一次 | 在每个测试方法之前或者之后都会运行一次 |
| @BeforeClass父类中标识了该Annotation的方法将会先于当前类中标识了该Annotation的方法执行。 @AfterClass 父类中标识了该Annotation的方法将会在当前类中标识了该Annotation的方法之后执行 | @Before父类中标识了该Annotation的方法将会先于当前类中标识了该Annotation的方法执行。 @After父类中标识了该Annotation的方法将会在当前类中标识了该Annotation的方法之后执行 |
| 必须声明为public static | 必须声明为public 并且非static |
| 所有标识为@AfterClass的方法都一定会被执行，即使在标识为@BeforeClass的方法抛出异常的的情况下也一样会。 | 所有标识为@After 的方法都一定会被执行，即使在标识为 @Before 或者 @Test 的方法抛出异常的的情况下也一样会。 |

## junit用例

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");

}

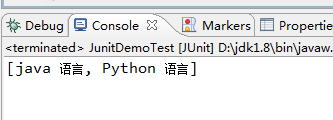
@Test

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

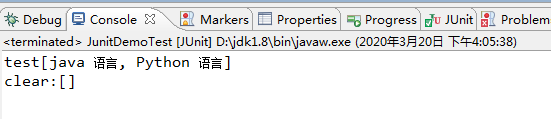
public void clear(){

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);

}

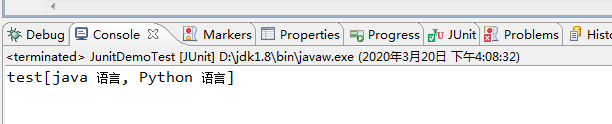
@Ignore

public void Ignore(){//方法未执行

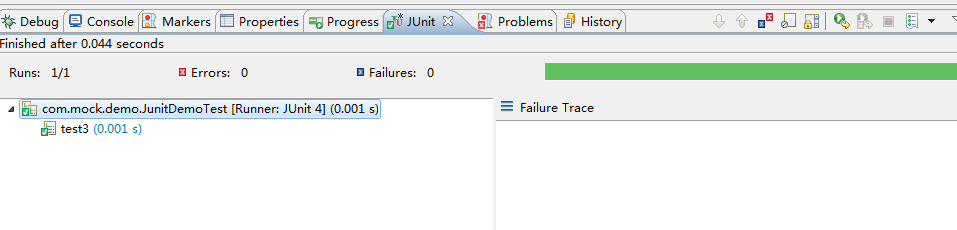
System.out.println("Ignore"+list);

}

}



Junit测试结果:

Failure 是测试失败，Error 是程序出错,当线条为绿色时，测试执行成功，当线条为红色时，执行失败。

## 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

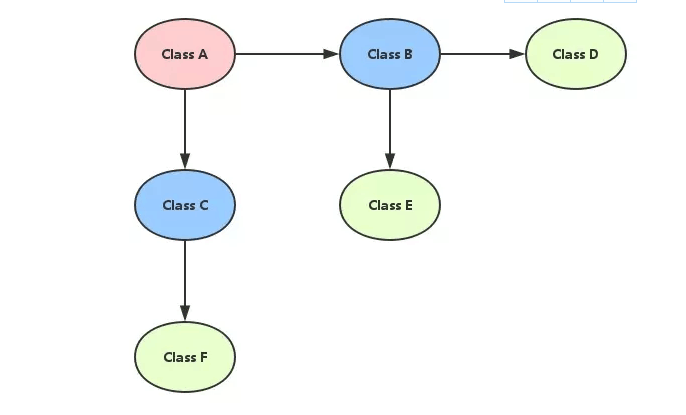
⑥测试类使用Test作为类名的后缀（不是必须）

⑦测试方法使用test作为方法名的前缀（不是必须）

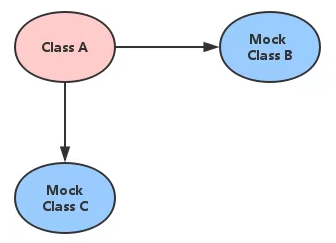
# Mockito介绍

## 什么是mock

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。

## mockito简介

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。

# Maven依赖

## springboot依赖

若使用springboot时，引入spring-boot-starter-test依赖性包

Pom.xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

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

## 引入mockito包

如果不使用spring boot，则包括添加一下依赖项：

pom.xml

<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>

# Mockito测试方法

## 常用的Mockito方法

|  |  |
| --- | --- |
| **方法名** | **描述** |
| Mockito.mock(classToMock) | 模拟对象 |
| Mockito.verify(mock) | 验证行为是否发生 |
| Mockito.when(methodCall).thenReturn(value1).thenReturn(value2) | 触发时第一次返回value1，第n次都返回value2 |
| Mockito.doThrow(toBeThrown).when(mock).[method] | 模拟抛出异常 |
| Mockito.mock(classToMock,defaultAnswer) | 使用默认Answer模拟对象 |
| Mockito.when(methodCall).thenReturn(value) | 参数匹配 |
| Mockito.doReturn(toBeReturned).when(mock).[method] | 参数匹配(直接执行不判断) |
| Mockito.when(methodCall).thenAnswer(answer)) | 预期回调接口生成期望值 |
| Mockito.doAnswer(answer).when(methodCall).[method] | 预期回调接口生成期望值(直接执行不判断) |
| Mockito.spy(Object) | 用spy监控真实对象,设置真实对象行为 |
| Mockito.doNothing().when(mock).[method] | 不做任何返回 |
| Mockito.doCallRealMethod().when(mock).[method] | 等价于Mockito.when(mock.[method]).thenCallRealMethod(); 调用真实的方法 |
| reset(mock) | 重置mock |

## 测试类

此实例主要对主要对两个类EmployeeService和EmployeeDao进行单元测试。

实体类：

@Data

public class EmployeeVO {

private int employeeId ;

private String firstName

private String lastName;

private String email;

/\*\*

\* <B>构造方法</B><BR>

\* @param employeeId

\* @param firstName

\* @param lastName

\* @param email

\*/

public EmployeeVO(int employeeId, String firstName, String lastName, String email) {

super();

this.employeeId = employeeId;

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

}

}

Dao层代码：

@Repository

public class EmployeeDao {

private Map<Integer, EmployeeVO> map = new HashMap<>();

public List<EmployeeVO> getEmployeeList()

{

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

if(list.isEmpty()) {

list.addAll(map.values());

}

return list;

}

public EmployeeVO getEmployeeById(int id) {

return map.get(id);

}

public void addEmployee(EmployeeVO employee) {

employee.setEmployeeId(map.keySet().size() + 1);

map.put(employee.getEmployeeId(), employee);

}

public void updateEmployee(EmployeeVO employee) {

map.put(employee.getEmployeeId(), employee);

}

public void deleteEmployee(int id) {

map.remove(id);

}

}

Service层代码：

@Service

public class EmployeeServcie {

@Autowired

EmployeeDao dao;

public List<EmployeeVO> getEmployeeList() {

return dao.getEmployeeList();

}

public EmployeeVO getEmployeeById(int id) {

return dao.getEmployeeById(id);

}

public void addEmployee(EmployeeVO employee) {

dao.addEmployee(employee);

}

}

## Mockito的使用

### 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。

### 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。

### @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还创建模拟实现，另外将标有注释的依赖模拟注入mock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。

### @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。

@RunWith(MockitoJUnitRunner.class)

public class TestEmployeeDao {

@Mock

EmployeeDao employeeDao;

@Test

public void TestGetEmployeeList(){

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

employeeDao.addEmployee(emp);

//验证addEmployee()方法是否执行过

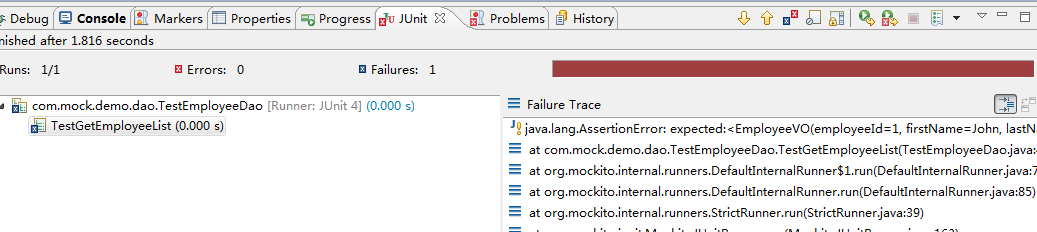
verify(employeeDao, times(1)).addEmployee(emp);

//测试验证

assertEquals(emp, employeeDao.getEmployeeById(1));

}

}



2.Spy声明的对象，对函数的调用均执行真正部分。

@RunWith(MockitoJUnitRunner.class)

public class TestEmployeeDao {

@Spy

EmployeeDao employeeDao;

@Test

public void TestGetEmployeeList(){

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

employeeDao.addEmployee(emp);

//验证addEmployee()方法是否执行过

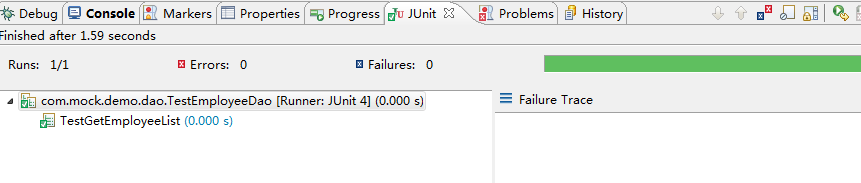
verify(employeeDao, times(1)).addEmployee(emp);

//测试验证

assertEquals(emp, employeeDao.getEmployeeById(1));

}

}



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。

### 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()

{

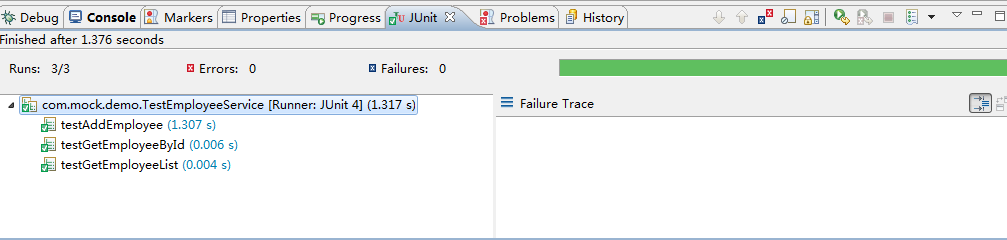
EmployeeVO emp = new EmployeeVO(1,"admin1","admin2","user@email.com");

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。

### 注意

1. 在单元测试中使用@Mock, @Spy, @InjectMocks等注解时，需要进行初始化后才能使用。
2. 若在单元测试类中使用了@RunWith(SpringJUnit4ClassRunner.class) 就不能再使用@RunWith(MockitoJUnitRunner.class),可以使用 MockitoAnnotations.initMocks(this) 来代替。
3. MockitoAnnotations.initMocks(this),其中this就是单元测试所在的类，在initMocks函数中Mockito会根据类中不同的注解（如@Mock, @Spy等）创建不同的mock对象，即初始化工作。

# Mockito的限制

Mockito的Mock对象使用方法,当使用Mockito在Mock对象时,一些限制需要遵守:

① 不能mock 静态方法

② 不能mock私有方法

③ 不能mock final 类

# 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

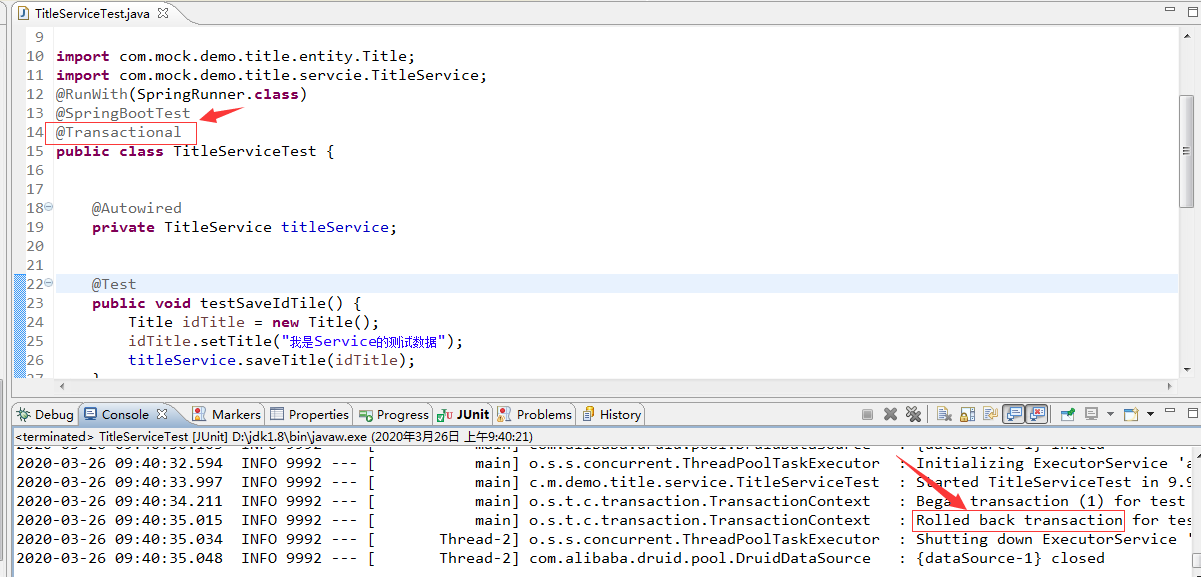
public void testSaveIdTile() {

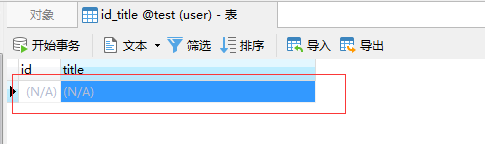
Title idTitle = new Title();

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)

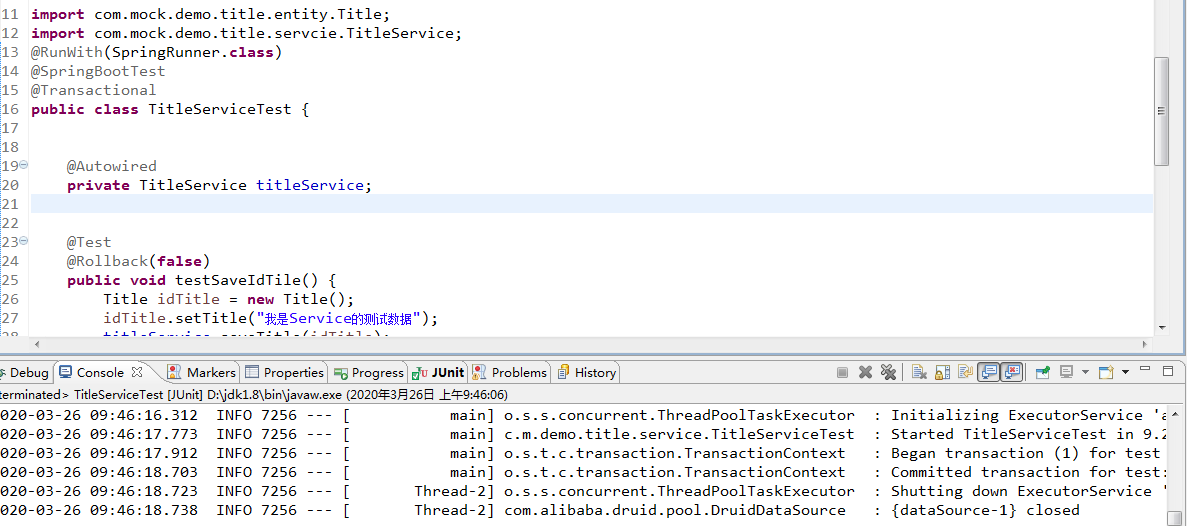
public void testSaveIdTile() {

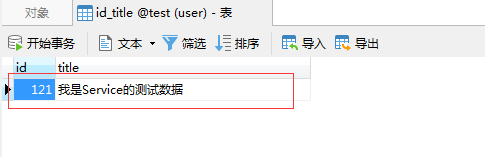
Title idTitle = new Title();

idTitle.setTitle("我是Service的测试数据");

titleService.saveTitle(idTitle);

}





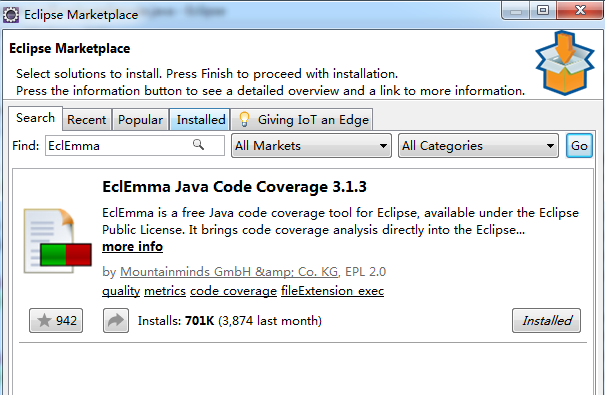
# 单元测试覆盖率

代码覆盖率（CodeCoverage）是反映测试用例对被测软件覆盖程度的重要指标,是衡量测试工作进展情况的重要指标,也是对测试工作进行量化的重要指标之一。

## EclEmma

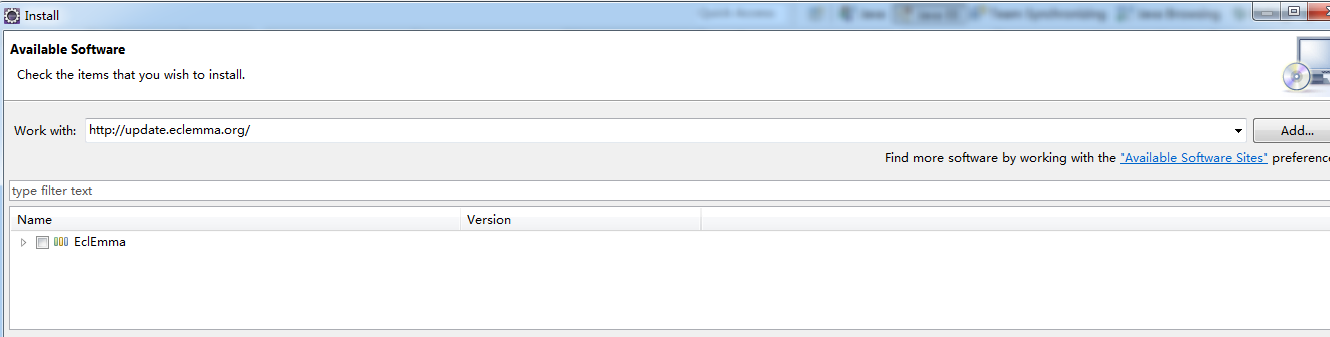
### 安装EclEmma插件

1. 从Eclipse Marketplace客户端安装



1. 从更新站点安装

**<http://update.eclemma.org/>**



1. 手动下载和安装

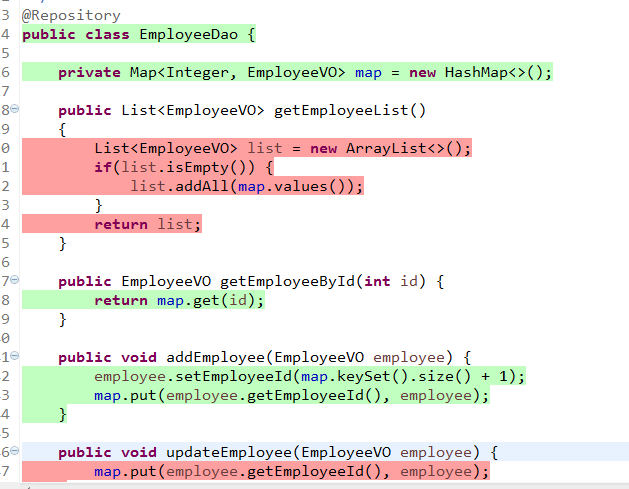
<https://www.eclemma.org/download.html>

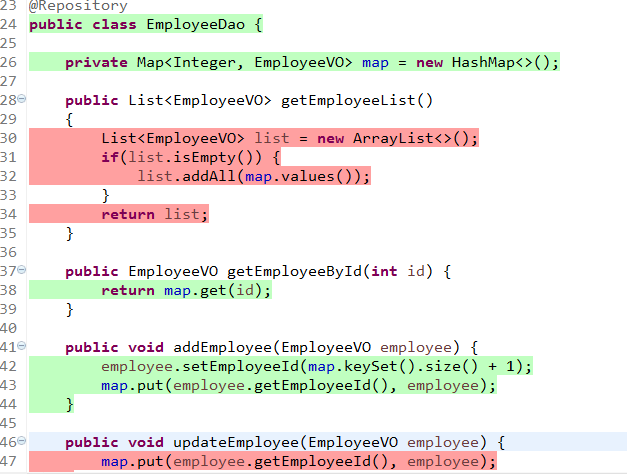
### 覆盖率统计

点击Coverage as —>Junit Test



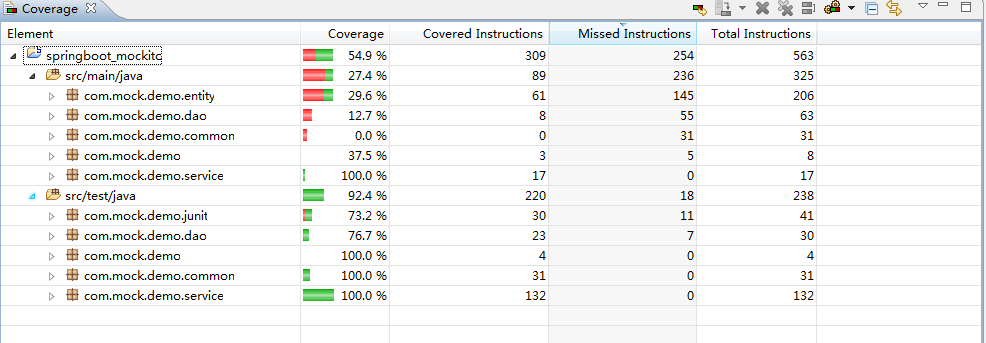
监测之后一般显示有3种颜色，其中红色表示未被覆盖，绿色表示已被覆盖，黄色表示部分被覆盖。





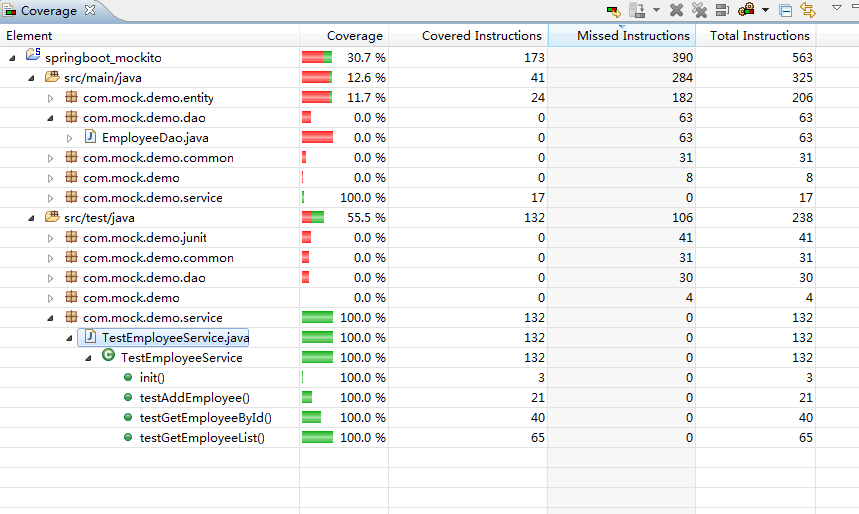
对整个项目进行覆盖率监测,点击项目名,Coverage as —>Junit Test,对整个项目进行覆盖率监测.

在Coverage窗口可查看测试覆盖率，如下图：



对单个文件检测,如：右键需要测试的java文件，Coverage as —>Junit Test

在Coverage窗口可查看测试覆盖率，如下图：



结果说明如下：

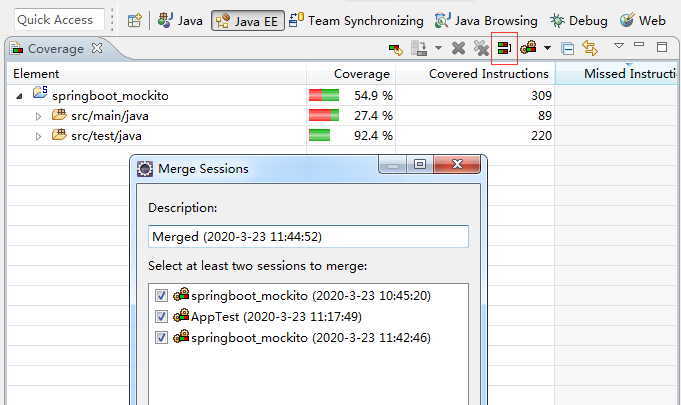
Coverage： 代码测试覆盖率

Covered Instructions：覆盖到的指令行

Missed Instructions：没覆盖的指令行

Total Instructions:总指令行数

每次执行都会有一个覆盖率统计在此显示，同时也可合并多次执行进行统计，只需点击下图中红框内的按钮



## Jacoco

Jacoco 是一个开源的覆盖率工具。Jacoco 可以嵌入到 Ant 、Maven 中，并提供了 EclEmma Eclipse 插件,也可以使用 Java Agent 技术监控 Java 程序。

### spring Boot集成jacoco

1. 首先在pom文件加入maven插件

<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>

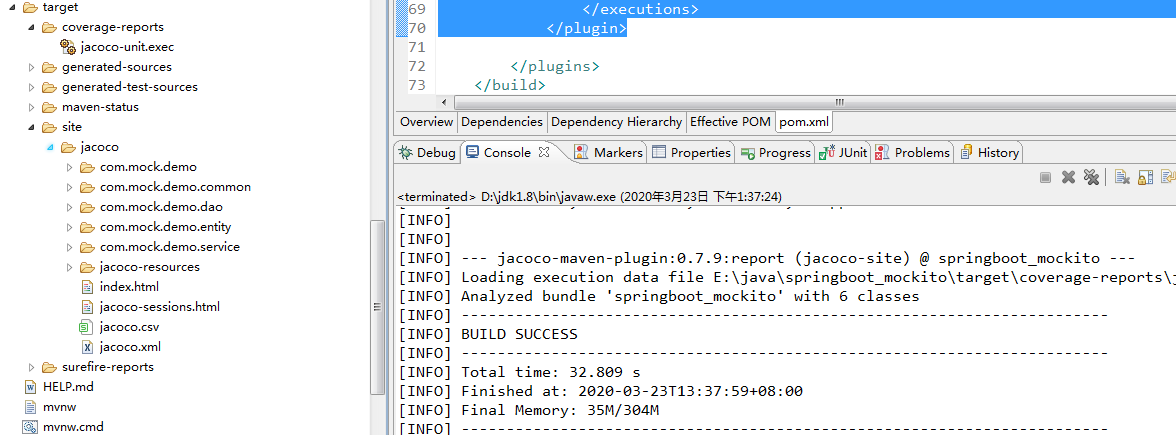
</goals>

</execution>

</executions>

</plugin>

1. 在当前项目的根目录下执行mvn test



### Jacoco测试覆盖率

进入项目target目录的site文件夹中，访问index.html。

