

**JAVA SpringBoot单元测试指导书**

|  |  |
| --- | --- |
| 项目名称 | 单元测试 |
| 文档编号 |  |
| 版 本 号 | V0.0.01 |
| 作 者 | 张鹏飞 |

版权所有

大唐移动通信设备有限公司

本资料及其包含的所有内容为大唐移动通信设备有限公司(大唐移动)所有,受中国法律及适用之国际公约中有关著作权法律的保护。未经大唐移动书面授权，任何人不得以任何形式复制、传播、散布、改动或以其它方式使用本资料的部分或全部内容，违者将被依法追究责任。

**文档更新记录**

| 日期 | 更新人 | 版本 | 修改记录 |
| --- | --- | --- | --- |
|  |  |  |  |
| 2018-03-28 | 张鹏飞 | V0.0.01 | 初稿 |
| 2018-05-15 | 张鹏飞 | V0.0.01 | 1、增加dao层使用H2数据库，去掉对数据库的依赖 增加2.6.2节  2、@server @component不再使用SprigBootTest注解的测试方法 增加2.5.2节 |
|  |  |  |  |
|  |  |  |  |

## 1 Spring Boot框架下项目单元测试

### 1.1概述

[单元测试](https://www.baidu.com/s?wd=%25E5%258D%2595%25E5%2585%2583%25E6%25B5%258B%25E8%25AF%2595&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLmWP-uH0YrH6zPHTdPyRv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHTvrHn3PHnLrjnsrj6snWbsn0)是对软件基本组成单元（软件设计的[最小单位](https://www.baidu.com/s?wd=%25E6%259C%2580%25E5%25B0%258F%25E5%258D%2595%25E4%25BD%258D&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLmWP-uH0YrH6zPHTdPyRv0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHTvrHn3PHnLrjnsrj6snWbsn0)）进行正确性检验的测试工作，如函数、过程(function,procedure)或一个类的方法(method)。

SpringBoot提供了许多的工具和注解来帮助程序开发者进行测试，这使得单元测试变得更加方便。SpringBoot对于测试的支持包括两个模块：spring-boot-test包含核心项目，以及spring-boot-test-autoconfigure用以支持对测试的自动配置。使用spring-boot-starter-test这个启动器可以导入Spring Boot测试模块以及JUnit，AssertJ，Hamcrest和其他一些有用的库。

### 1.2 SpringBoot单元测试的基本概念

正如上文所示，SpringBoot 提供了一些列的注解和库来帮助开发人员完成单元测试，主要包括以下几方面：

#### 1.2.1@SpringBootTest注解

SpringBoot提供了@SpringBootTest注解，在测试中使用该注解可以创建一个ApplicationContext（Spring中负责对象的创建和组装）。

Mock 用来装配一个WebApplicationContext同时提供一个mock servlet的环境。

RANDOM\_PORT DEFINED\_PORT指定一个端口号。

NONE 通过SpringApplication 装配一个ApplicationContext但不提供任何的servlet环境。

#### 1.2.2 mock 所需的bean

在单元测试中经常需要模拟某些组件（Bean），比如需要某个不可用的远程环境或者在真实环境下很难触发的故障或异常，这时mock是必不可少的。

Spring Boot提供了一个@MockBean注解，

### 1.3 mave工程中配置Spring Boot单元测试

pom.xml文件中增加

<!--单元测试-->

<dependency>

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

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

<scope>test</scope>

</dependency>

## 2 Mockito

### 2.1 Mock测试

Mock 测试就是在测试过程中，对于某些不容易构造的（如 HttpServletRequest 必须在Servlet 容器中才能构造出来）或者不容易获取比较复杂的对象（如 JDBC 中的ResultSet 对象），用一个虚拟的对象（Mock 对象）来创建以便测试的测试方法。

### 2.2 Mockito组件

Mockito 是流行的 Java 单元测试 Mock 框架，免费[开源](https://github.com/mockito)。

大多数 Java Mock 库如 EasyMock 或 JMock 都是 expect-run-verify （期望-运行-验证）方式，而 Mockito 则使用更简单，更直观的方法：在执行后的互动中提问。使用 Mockito，你可以[验证任何你想要的](http://monkeyisland.pl/2008/02/24/can-i-test-what-i-want-please)。而那些使用 expect-run-verify 方式的库，你常常被迫查看无关的交互。[非 expect-run-verify 方式](http://monkeyisland.pl/2008/02/01/deathwish/) 也意味着，Mockito 无需准备昂贵的前期启动。他们的目标是透明的，让开发人员专注于测试选定的行为。

Mockito 拥有的非常少的 API，所有开始使用 Mockito，几乎没有时间成本。因为只有一种创造 mock 的方式。只要记住，在执行前 stub，而后在交互中验证。你很快就会发现这样 TDD java 代码是多么自然。

### 2.3 SpringBoot中使用Mockito

在Spring Boot应用程序单元测试的过程中，我们有时候需要在应用程序上下文中Mock一个组件。Spring Boot提供一个注解@MockBean，该注解可以被用来定义一个Mockito mock到我们的ApplicationContext中。

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.beans.factory.annotation.\*;

**import** org.springframework.boot.test.context.\*;

**import** org.springframework.boot.test.mock.mockito.\*;

**import** org.springframework.test.context.junit4.\*;

**import** **static** org.assertj.core.api.Assertions.\*;

**import** **static** org.mockito.BDDMockito.\*;

*@RunWith(SpringRunner.class)*

*@SpringBootTest*

**public** **class** MyTests {

*@MockBean*

**private** RemoteService remoteService;

*@Autowired*

**private** Reverser reverser;

*@Test*

**public** **void** exampleTest() {

*// RemoteService has been injected into the reverser bean*

given(**this**.remoteService.someCall()).willReturn("mock");

String reverse = reverser.reverseSomeCall();

assertThat(reverse).isEqualTo("kcom");

}

}

### 2.4 对Spring Boot Controller层进行测试

如果想要测试应用程序 Spring MVC的 控制器是否按照预期工作，可以使用@WebMvcTest注解来自动的配置SpringMVC的基础设施环境。@WebMVcTest注解会自动配置MockMvc，MockMvc 提供一个强大的方式来快速的测试 MVC 控制器，而无需启动HTTP server，当然我们也可以在没有@WebMvcTest注解的情况下自动配置一个 MockMvc，比如通过 @SpringBootTest注解，添加@AutoConfigureMockMvc来实现。

**import** org.junit.\*;

**import** org.junit.runner.\*;

**import** org.springframework.beans.factory.annotation.\*;

**import** org.springframework.boot.test.autoconfigure.web.servlet.\*;

**import** org.springframework.boot.test.mock.mockito.\*;

**import** **static** org.assertj.core.api.Assertions.\*;

**import** **static** org.mockito.BDDMockito.\*;

**import** **static** org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;

**import** **static** org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;

*@RunWith(SpringRunner.class)*

*@WebMvcTest(UserVehicleController.class)*

**public** **class** MyControllerTests {

*@Autowired*

**private** MockMvc mvc;

*@MockBean*

**private** UserVehicleService userVehicleService;

*@Test*

**public** **void** testExample() **throws** Exception {

given(**this**.userVehicleService.getVehicleDetails("sboot"))

.willReturn(**new** VehicleDetails("Honda", "Civic"));

**this**.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT\_PLAIN))

.andExpect(status().isOk()).andExpect(content().string("Honda Civic"));

}

}

### 2.5 对 Spring Boot Server层进行测试

##### 2.5.1 使用Spring容器管理

Spring Boot Server层的单元测试主要需要mock一个 dao，如下：

@RunWith(SpringRunner.**class**)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.***RANDOM\_PORT***)

**public** **class** TestServiceImpl {

@Autowired

UserService userSerice;

List<User> userList = **new** ArrayList<User>();

@MockBean

UserMapper userDao;

@Test

**public** **void** testGetAll(){

User user = **new** User();

user.setName("huaAn");

user.setId(0);

user.setAccount("9527");

BDDMockito.*given*(**this**.userDao.get(0)).willReturn(user);

org.junit.Assert.*assertTrue*(userSerice.getUser(0).equals(user));

}

}

##### 2.5.2 不依赖Spring容器管理

为了使得单元测试（unit tests）有更少的依赖，同时提高编写和调试单元测试的效率，建议server层的单元测试代码测试时不依赖 Spring容器，将2.5.1的方法用于 集成测试（integration tests 。我们的实现思路为，首先new一个需要测试的类对象，然后将该类所需要的所有外部依赖mock并set到被测试类中，从而完成对测试环境的构建。如下是性能服务中测量任务模块的测试代码：

(1) mock 外部依赖并将依赖注入被测试类

（2）使用mock和构造的对象数据进行测试并断言



### 2.6 对 Spring Boot Dao层进行测试

##### 2.6.1 @DataJpaTest

可以用来测试JPA，默认它会使用一个嵌入式内存数据库。扫描@Entity类和JAP repository。Data JPA 测试还会默认注入一个TestEntityManager，它是标准EntityManager的替代品。

**package** com.springboot.jpa;

**import** **static** org.hamcrest.CoreMatchers.is;

**import** **static** org.junit.Assert.*assertThat*;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import**org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;

**import**org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;

**import**org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTes;

**import**org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** com.springboot.jpa.entity.User;

**import** com.springboot.jpa.repository.UserRepository;

/\*\*

\* **@author** zhangpengfei2

\*

\*/

@RunWith(SpringRunner.**class**)

@DataJpaTest

@AutoConfigureTestDatabase(replace = Replace.***NONE***)

**public** **class** UserJPA {

@Autowired

**private** UserRepository repository;

@Autowired

**private** TestEntityManager entityManager;

@Test

**public** **void** testExample() **throws** Exception {

User user = **new** User();

user.setAge(15);

user.setId(10);

user.setUsername("zhangsan");

**this**.entityManager.persist(user);

User userActual = **this**.repository.findByUsername("zhangsan");

*assertThat*(userActual.getUsername(),*is*("zhangsan"));

}

}

##### 2.6.2 h2 数据库的配置方法

（1）添加maven依赖

<!-- 内嵌数据库 -->

<dependency>

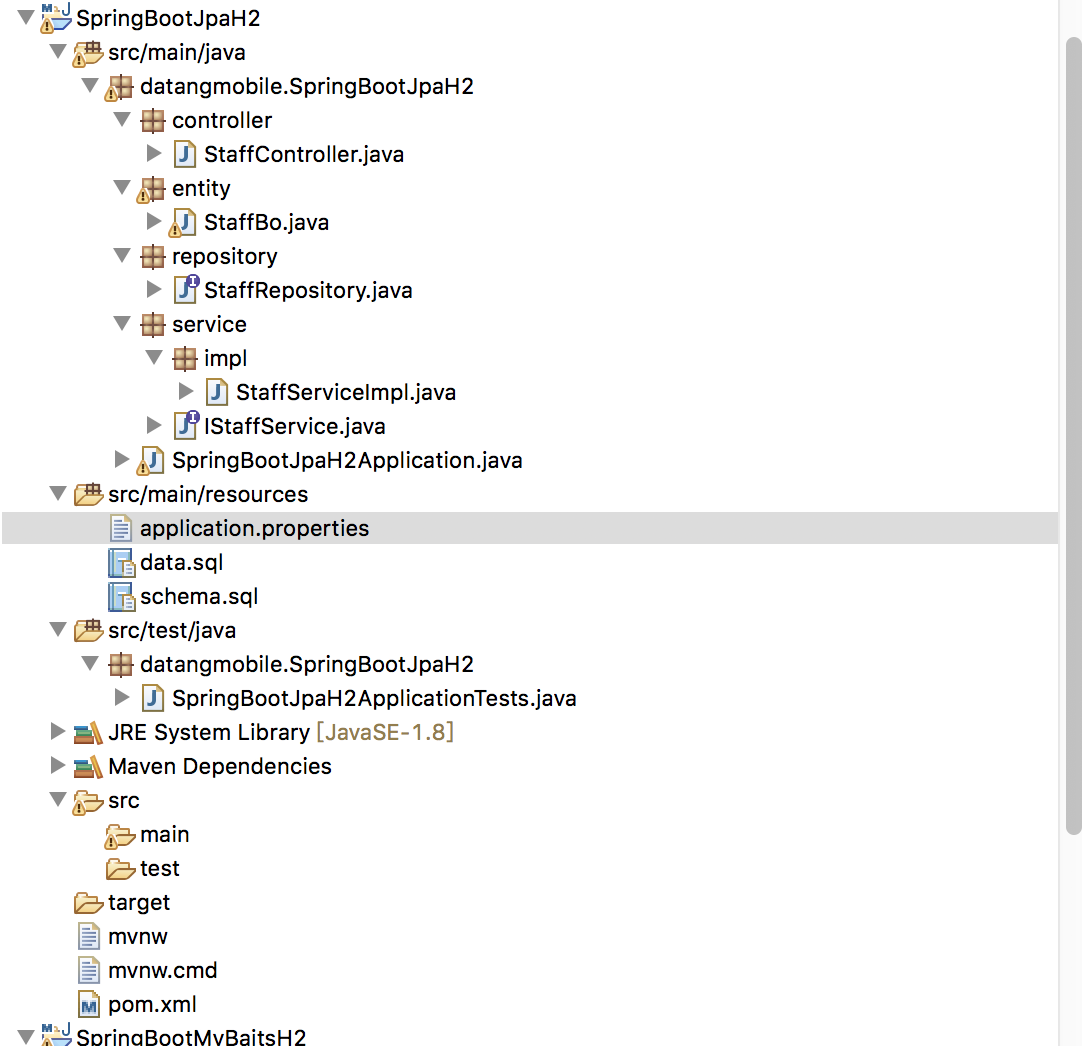
<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

（2）配置h2 数据库

（3）添加相应的数据库脚本 data.sql schema.sql



该实例的源码



## 3.JUnit

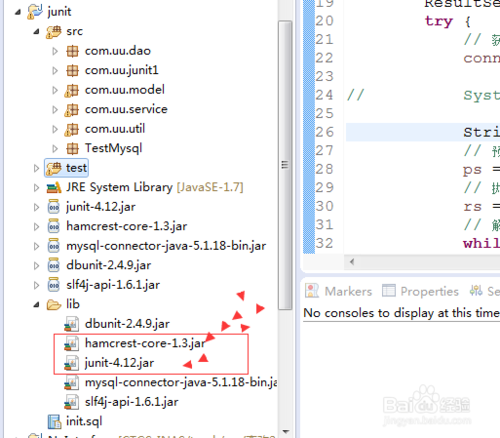
### 3.1 Junit 简介

JUnit 是Java语言事实上的标准单元测试库。JUnit的易用性无疑是它受欢迎的主要原因。在SpringBoot框架下，只需要添加spring-boot-starter-test启动器便可自动给项目添加JUnit的依赖。

### 3.2 使用JUnit单元测试的框架

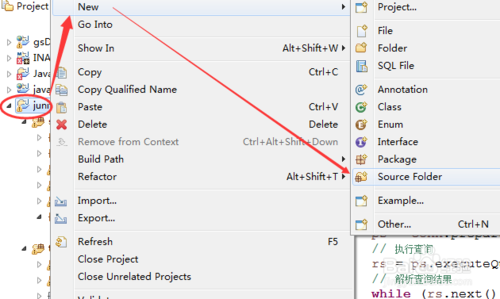
#### 3.2.1 使用JUnit编写单元测试的步骤

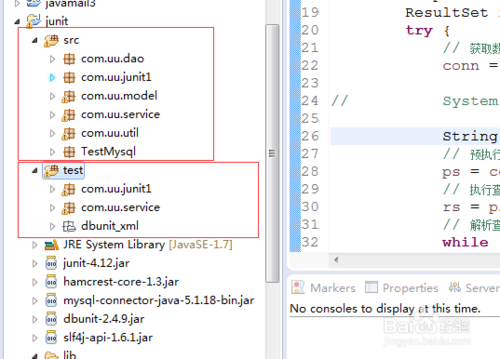
1. **导入junit测试需要的依赖包。**首先创建一个java Project；将**junit包和hamcrest-core包**导入到项目中，如果是maven项目则需要搜索对应的依赖pom内容。



1. 2

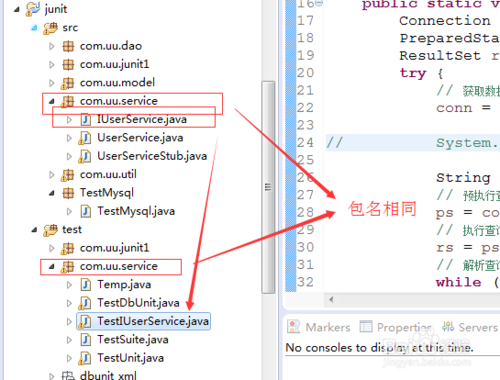
**创建资源文件夹存储资源文件**。创建一个source folder 资源文件夹 test，资源文件夹和普通的文件夹区别是，资源文件夹中的java文件可以被自动编译。





1. 3

**编写java文件对应的测试类，一般命名方式：Test+测试类或者测试类名+Test。**在java文件中编写java代码UserMananger.java，在test文件中创建一个与需要进行测试的java代码同包名称的TestUserManager.java文件。



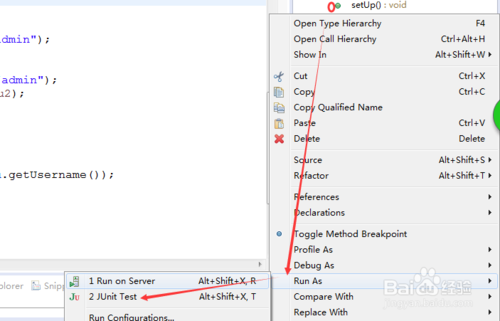
1. 4

**junit4采用的是通过注解的方式（在方法的上面加上@）.一般@Test使用最多。**

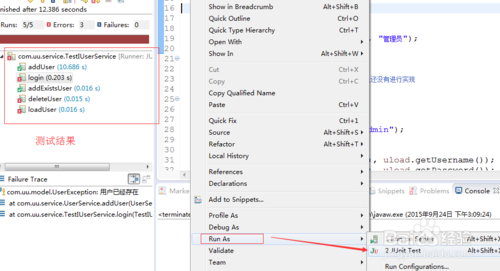


1. 5

**运行测试方法或者测试类。**直接在**带有@Test注解的方法上上点击右键 --> run as --> JUnit Test**



**查看运行结果是否通过测试。**如果是对所有的方法都执行一遍,在类体上右键--Run as--> JunitTest .**绿色的表示运行通过的方法,红x的表示运行失败的方法.**



#### 3.2.2 JUnit单元测试代码结构

1. import声明引入必需的JUnit类
2. 一个通过@Test注解标识的测试方法
3. 断言，判断结果是否到达预期

**package** com.example.test.service;

**import** java.util.ArrayList;

**import** java.util.List;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.test.context.SpringBootTest;

**import** org.springframework.test.context.junit4.SpringRunner;

**import** com.example.domain.User;

**import** com.example.service.UserService;

/\*\*

\* **@author** zhangpengfei2

\*

\*/

@RunWith(SpringRunner.**class**)

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.***RANDOM\_PORT***)

**public** **class** TestServiceImpl {

@Autowired

UserService userSerice;

List<User> userList = **new** ArrayList<User>();

@Test

**public** **void** testGetAll(){

org.junit.Assert.*assertTrue*(userSerice.getAll().equals(userList));

}

}

### 3.2 断言

断言是单元测试最基本得组成部分，因此Junit提供了不同形式的多种断言。

|  |  |  |
| --- | --- | --- |
| 方法名 | 含义 | 备注 |
| void assertEquals(String message, Object expected, Object actual) | 判断预期值和测试实际产生值是否相等 |  |
| void assertArrayEquals(char[] expecteds, char[] actuals) | 验证一个给定对象是否为空 |  |
| Void assertSame(Objectexpected, Object actual) | 验证引用的是否为同一个对象 |  |
| void assertTrue(String message, boolean condition) | 判断二元条件是否为真，message参数为可选 |  |
| Fail(String message) | 使测试立即失败，通常用于标记某个不应该到达的分支 | 例如，在某个预期发生的异常之后 |
|  |  |  |
|  |  |  |

### 3.4 异常测试

通常，对于方法中的每一个被期望的异常，都应该编写一个专门的测试，来确认，该方法在应该抛出异常的时候会确实抛出异常。

@Test(expected=ArithmeticException.**class**)

**public** **void** testDivide(){

*assertEquals*(4, 12/0);

}

### 3.5Junit注解

|  |  |  |
| --- | --- | --- |
| 注解名 | 基本功能 | 备注 |
| @Test | 将一个普通方法修饰为一个测试方法 |  |
| @Test(exception=XXX.class) | 进行异常测试 |  |
| @Test(time=毫秒) | 测试超时 |  |
| @BeforeClass | 它会在所有的测试方法前被执行，static修饰 |  |
| @AfterClass | 它会在所有的测试方法后被执行，static修饰 |  |
| @Before | 它会在每一个测试方法前被执行一次 |  |
| @After | 它会在每一个测试方法后被执行一次 |  |
| @Ignore | 省略 |  |
| @RunWith | 修改运行器org。junit。runner。Runner |  |

## 4.单元测试覆盖率

Eclemma是目前较为流行的Java单元测试覆盖率工具，该工具可以在esclipse工作空间展示单元测试覆盖率，同时可以以html等多种流行的文件格式导出覆盖率报告。

### 4.1 Eclemma插件包安装



第1步

采用下载包方式进行安装，先将zip解压，目录结构如下：

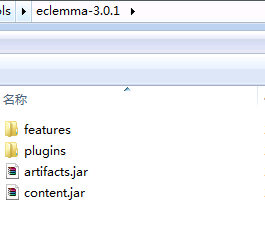


图 41

* 第2步

将该目录直接复制到eclipse安装目录的dropins目录下，如下图所示：

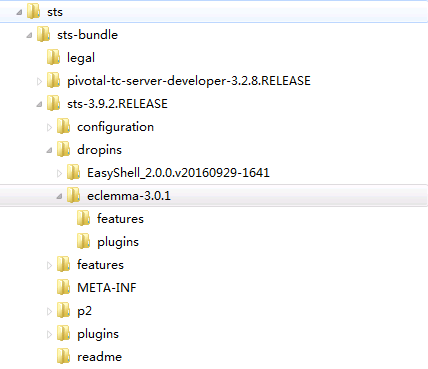
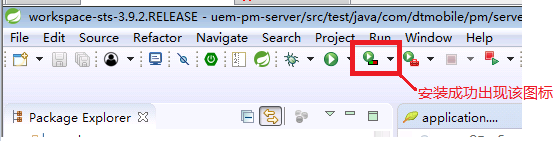


图 42

* 第3步

重新启动Eclipse，Eclipse会自动安装插件，如果安装成功，则Eclipse界面如下：



### 4.2设置范围

默认情况Eclemma是对整个工程的源代码(包括产品代码和测试用例代码)进行代码覆盖分析，如果该工程依赖于其他工程，则依赖工程的源代码也将作为分析对象。鉴于目前使用该工具主要用于检测单元测试用例对被测代码的覆盖率，因此选择范围仅包含该工程的产品代码，不包含测试用例代码(如果有依赖工程，也不包含其源代码)。设置代码分析范围的步骤如下图所示。

* 第1步

选中含有Junit实现测试用例方法的类或一个包，点击右键，在弹出菜单选择Converage As，然后选择Converge Configurations。

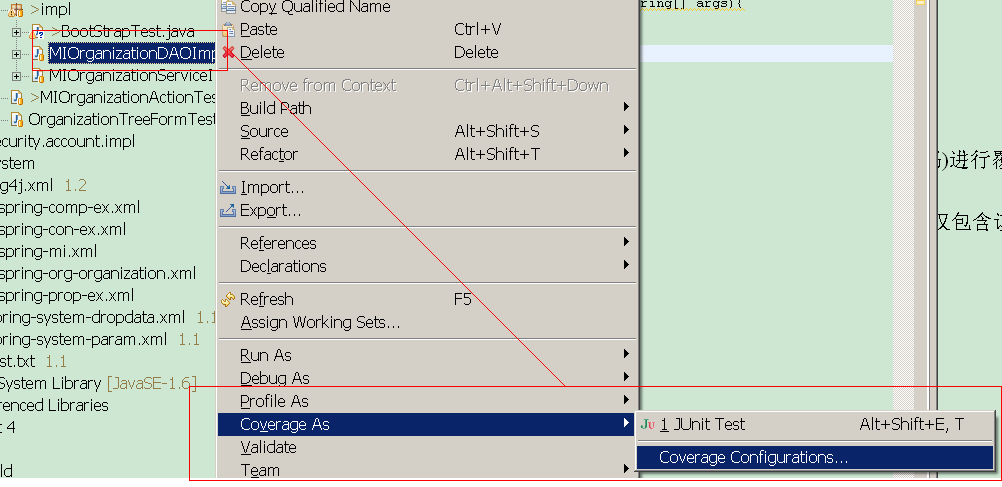


图 44

* 第2步

在converage功能页选中java代码；

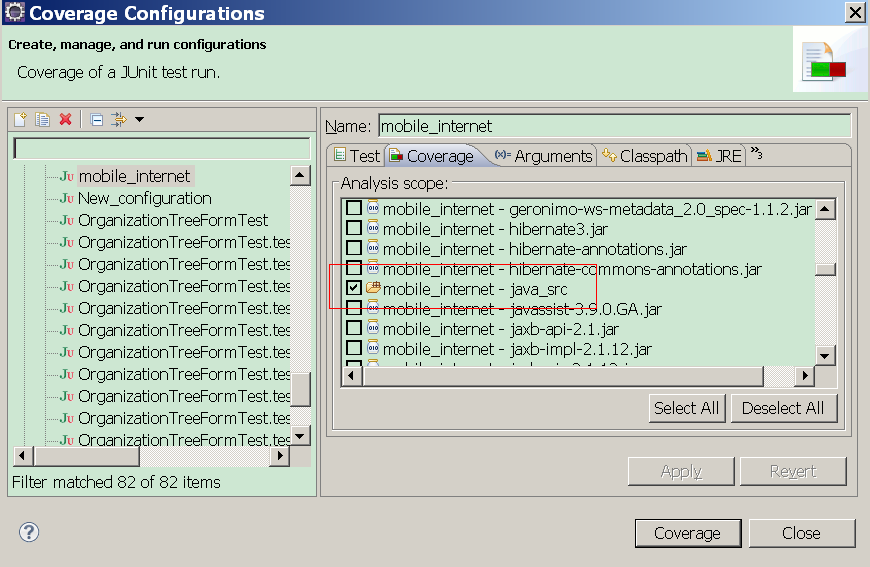


图 45

* 第3步

在converage功能页不选单元测试代码，然后点击Converage可以直接运行。

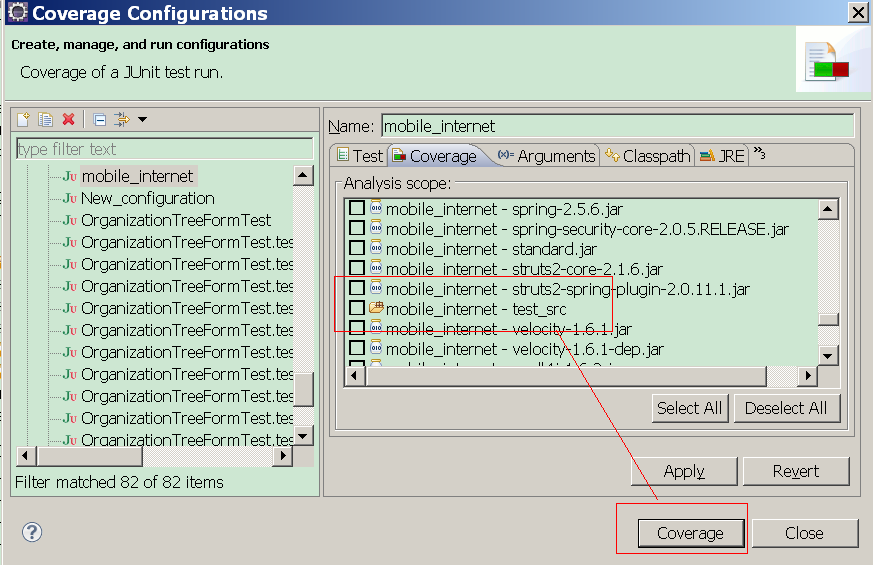


图 46

### 4.3运行方式

代码覆盖率统计是嵌入在一般的运行过程中，既可以在运行应用程序(通过main方法)时，也可以在进行单元测试时(通过junit插件)，但必须通过该插件提供的覆盖模式的运行程序。

应用程序运行方式如下图所示，选中含有main方法的类，点击右键，在弹出菜单选择Converage As，然后选择Java Application运行。

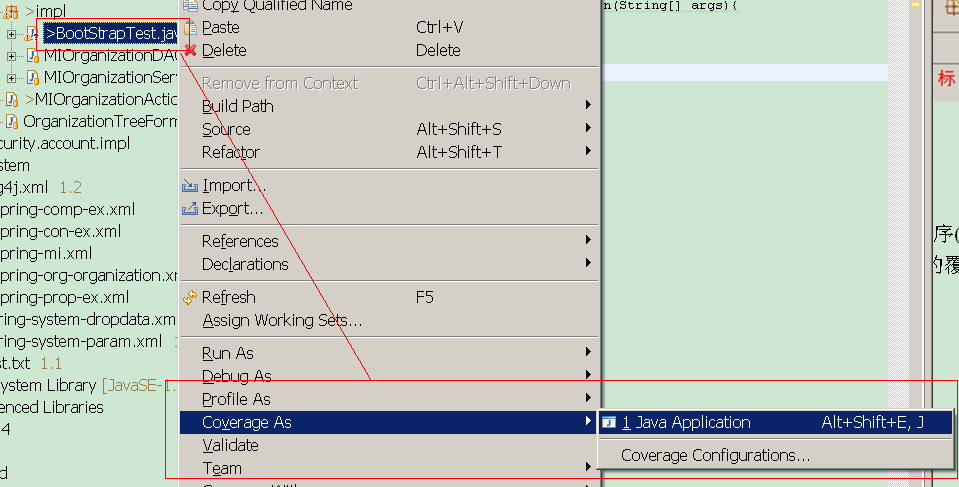


图 47

单元测试代码运行方式如下图所示，选中含有Junit实现测试用例方法的类或一个包，点击右键，在弹出菜单选择Converage As，然后选择Junit Test运行。

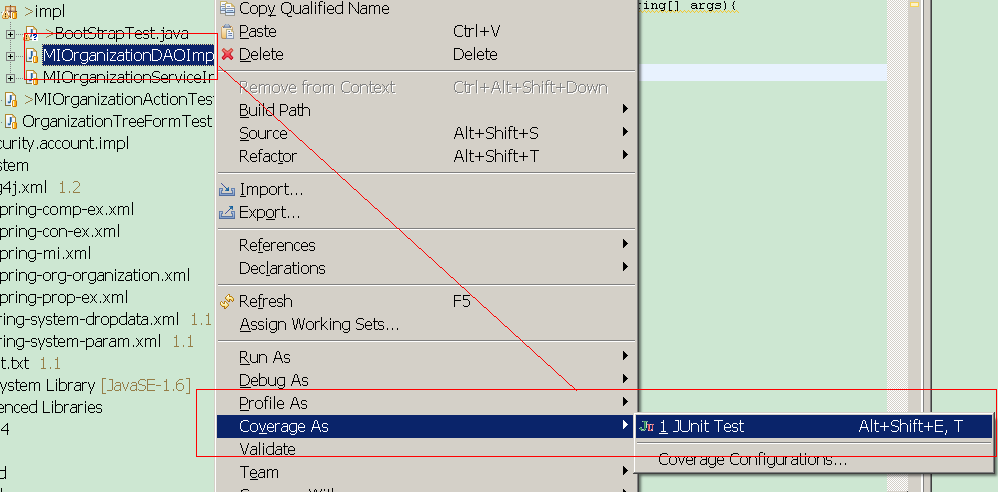


图 48

按照上述方式运行一次之后，Eclemma能够记录下本次的运行设置，下次可以直接选择运行，如下图所示：

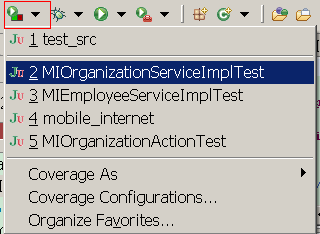


图 49

### 4.4覆盖视图

* 第1步

根据运行方式章节的方法运行后，打开覆盖视图(【windows】->【Show View】->【Others】)，如下图所示：

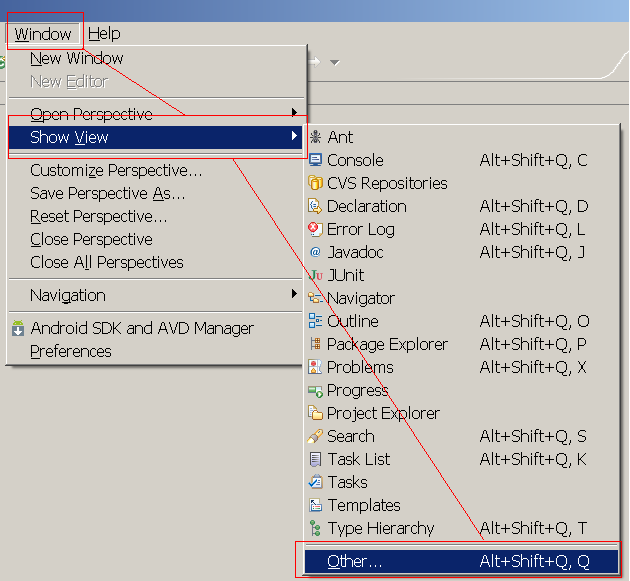


图 410

* 第2步

选择Java->Converage；

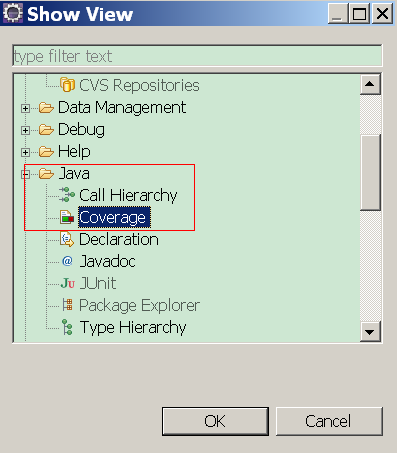


图 411

覆盖视图如下图所示，后四列分别为：覆盖率、覆盖行数、未覆盖行数和总行数。这是按照代码行覆盖进行统计的结果，如果是按照其他覆盖类型进行统计，列的标题和值会有相应的变化。

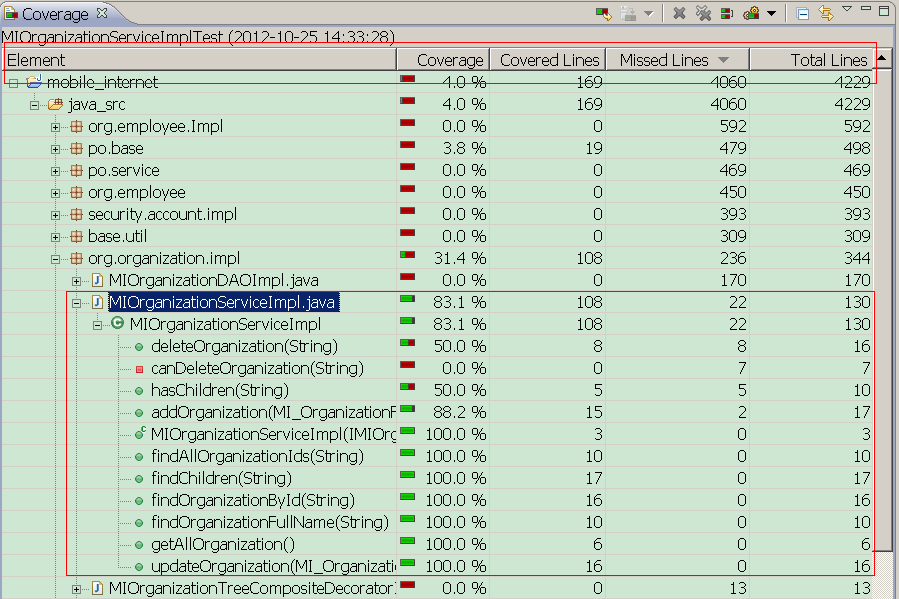


图 412

从上图可以清晰地看到每个包、类及其方法的代码覆盖情况。

Eclemma将每次运行的覆盖结果称为一个Session, 多次运行形成多个Session，可以在Session之间切换统计结果，如下图所示：

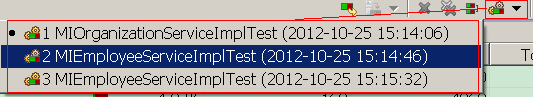


图 413

多个Session可以合并成一个Session如下图所示，则原来的三个Session被一个Session(Merged 2012-10-25 15:19:09)取代。

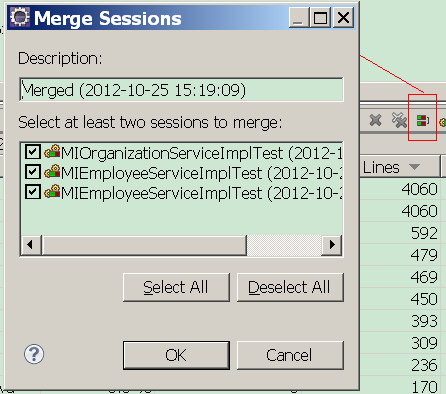


图 414

重新加载和删除session功能如下图所示：

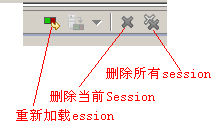


图 415

项目文件显示方式、覆盖计数器类型和过滤器功能如下图所示：

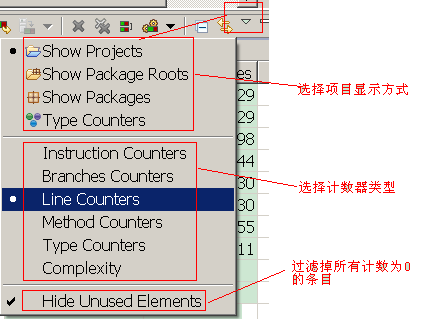


图 416

### 4.5覆盖图示

可以设置在Java工程视图中直接显示覆盖率，设置方法为：

* 第1步

【Window】->【Preferences】->【Genera】l->【Appearance】->【Label Decorators】->选择【Java Code Coverage】。

具体操作步骤如下图所示：

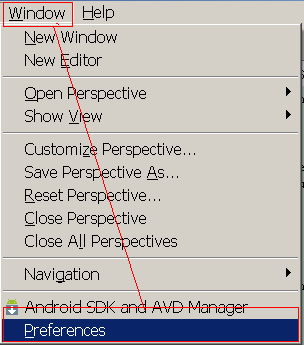


图 417

* 第2步

选择java Code Coverage项；

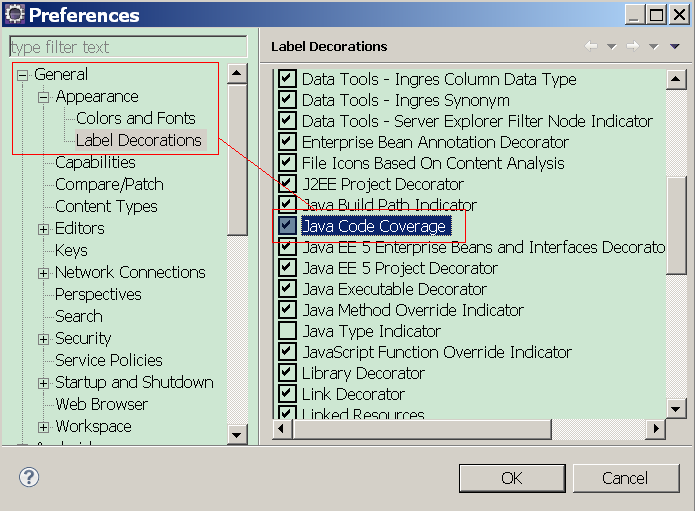


图 418

* 第3步

根据以上设置之后，java工程视图的显示效果如下图所示：

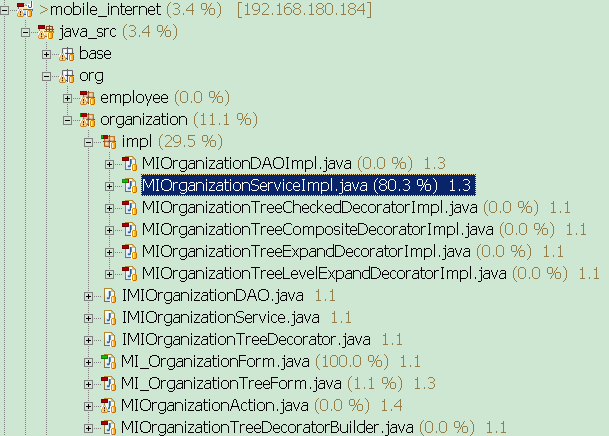


图 419

### 4.6代码标注

Elemma通过不同颜色区分代码的覆盖情况，如下图所示：

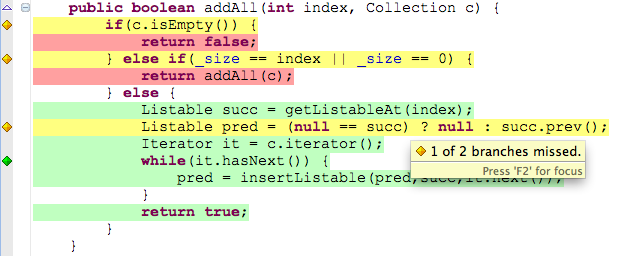


图 420

完全覆盖：绿色

部分覆盖：黄色

没有覆盖：红色

可以通过如下方式修改“完全覆盖”代码的颜色或显示风格：

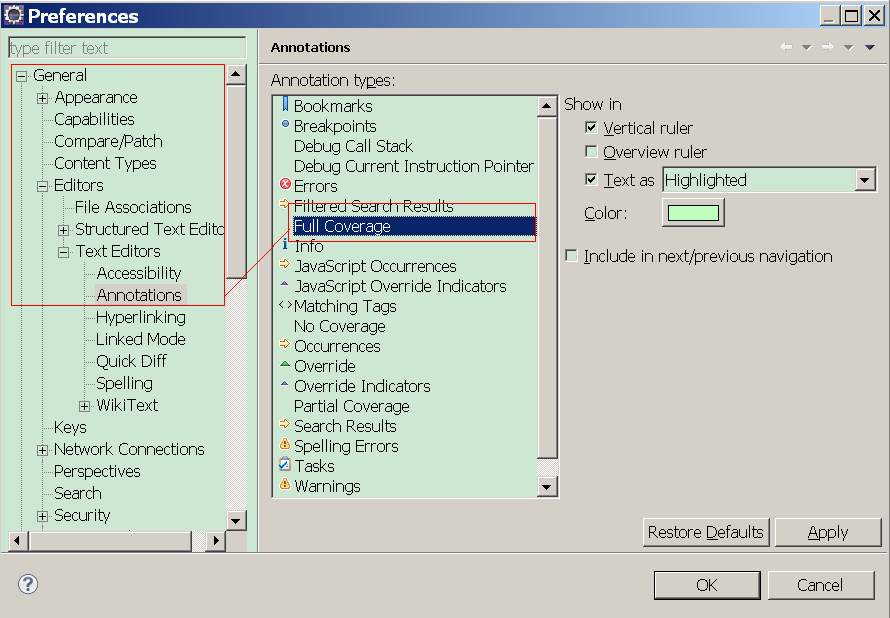


图 421

可以通过如下方式修改“部分覆盖”代码的颜色或显示风格：

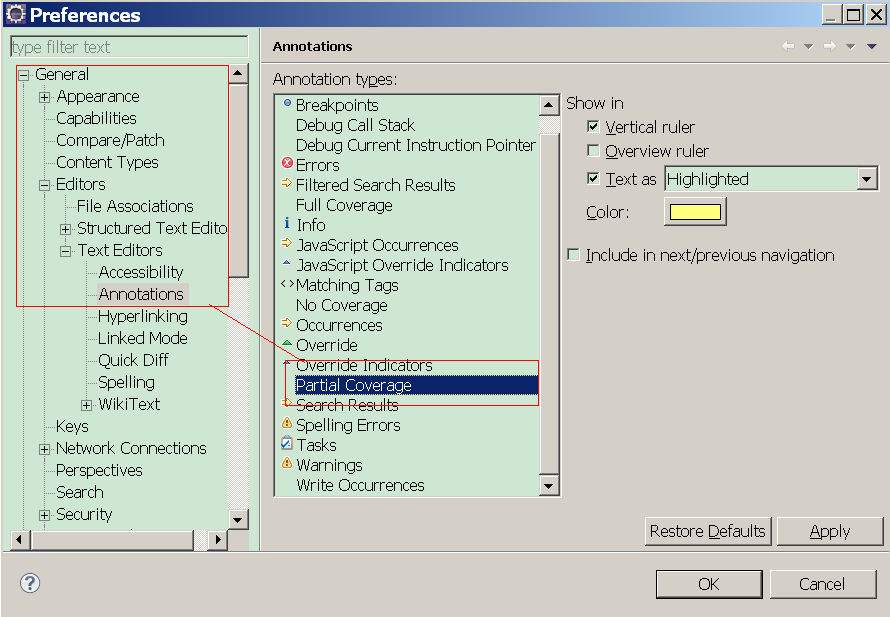


图 422

可以通过如下方式修改“没有覆盖”代码的颜色或显示风格：

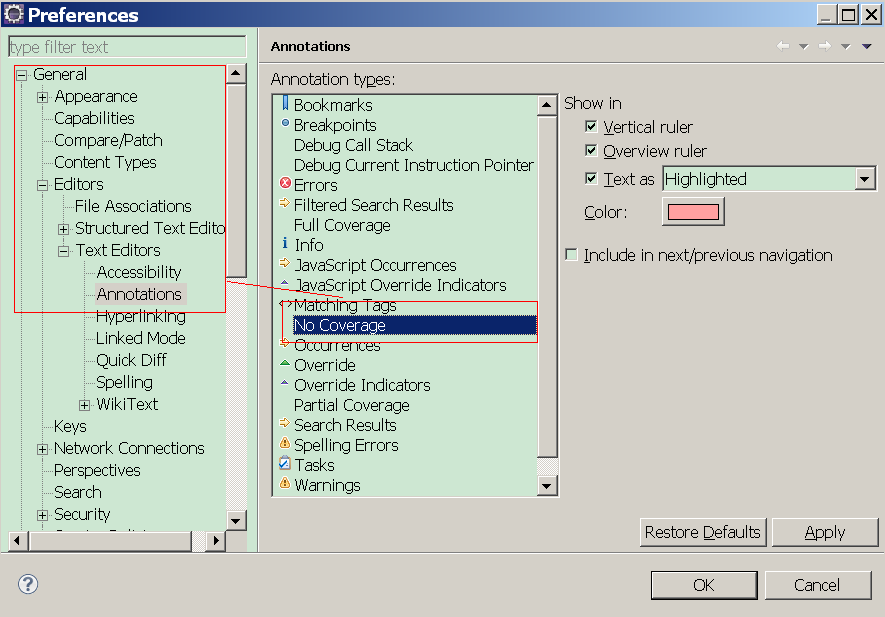


图 423

删除所有的Session后，源代码的标注自动消除。

### 4.7覆盖属性

可以针对包或单个类查看其代码覆盖统计数据，如下图所示：

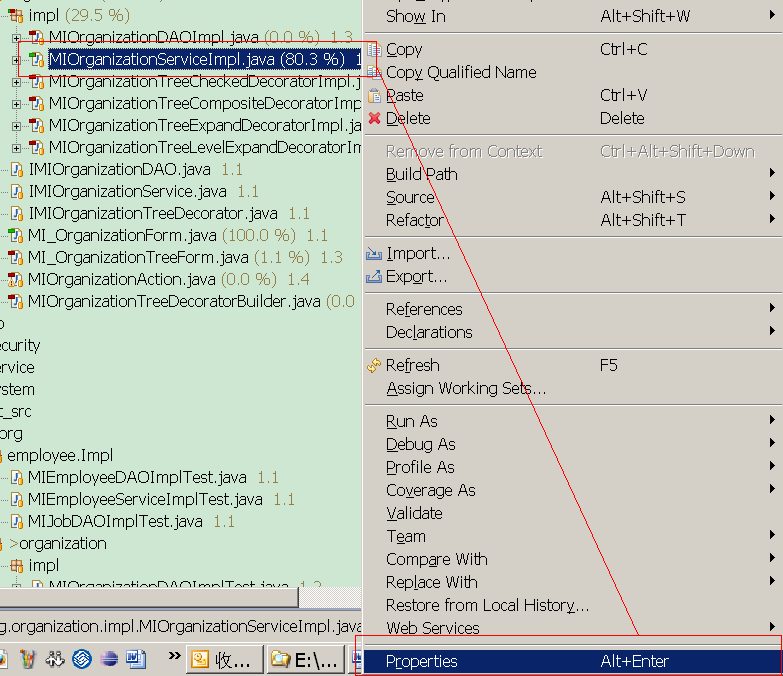


图 424

选择【Coverage】项:

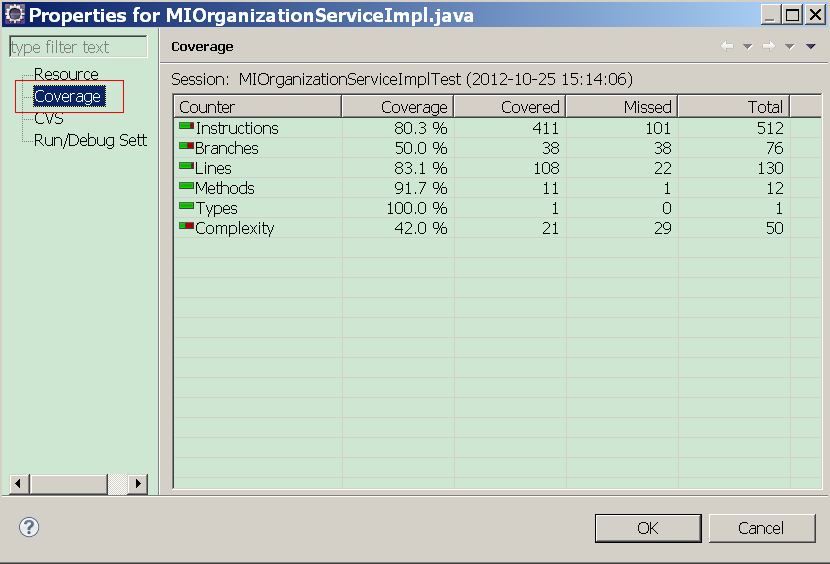


图 425

### 4.8快捷方式

|  |  |
| --- | --- |
| **按键序列** | **功能说明** |
| Ctrl+Shift+F11 | 按照最后一次运行设置再运行一次。 |
| Alt+Shift+E, J | 以Java应用程序的方式运行选中内容。 |
| Alt+Shift+E, T | 以Junit单元测试的方式运行选中内容。 |

### 4.9生成报告

可以通过Eclemma提供的导出功能可以直接将覆盖结果导出成html格式的覆盖统计报告。具体的操作方式如下图所示：

* 第1步

在File菜单中选择Export导出项；

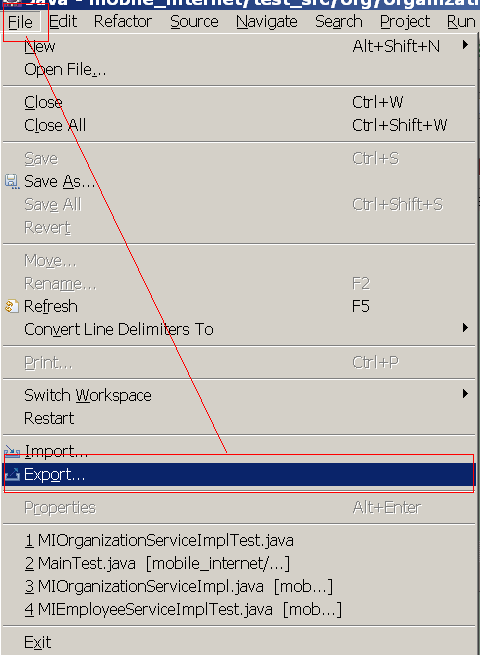


图 426

* 第2步

在Java目录下选择Coverage Report；

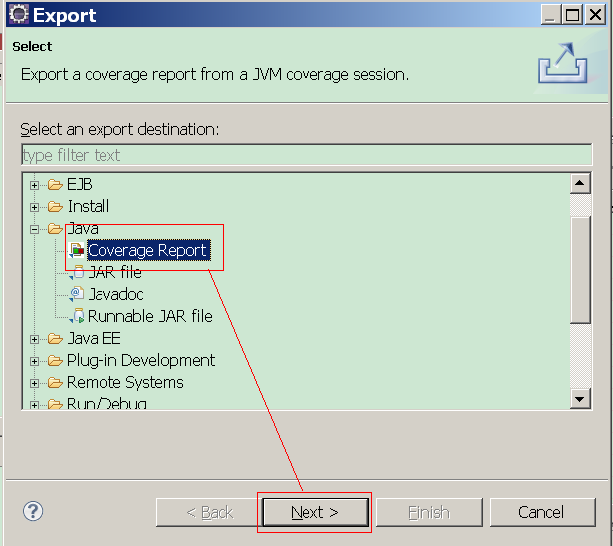


图 427

* 第3步

选择要生成报告的Session名称，并输入报告的生成目录；

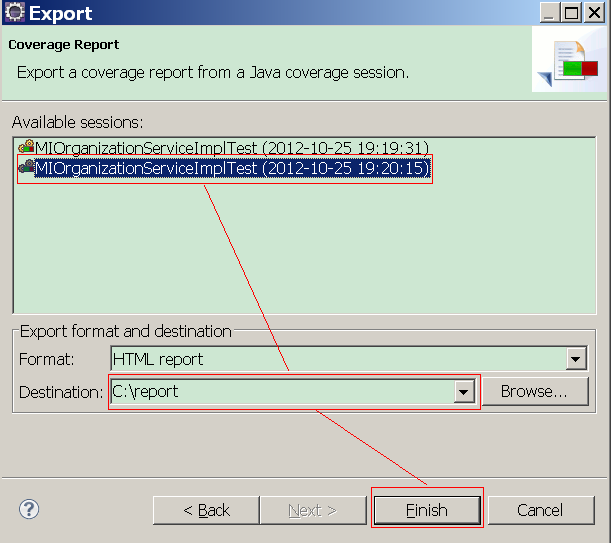


图 428

* 第4步

报告的生成目录如下图所示：



图 429

* 第5步

生成报告的内容，如下图所示：

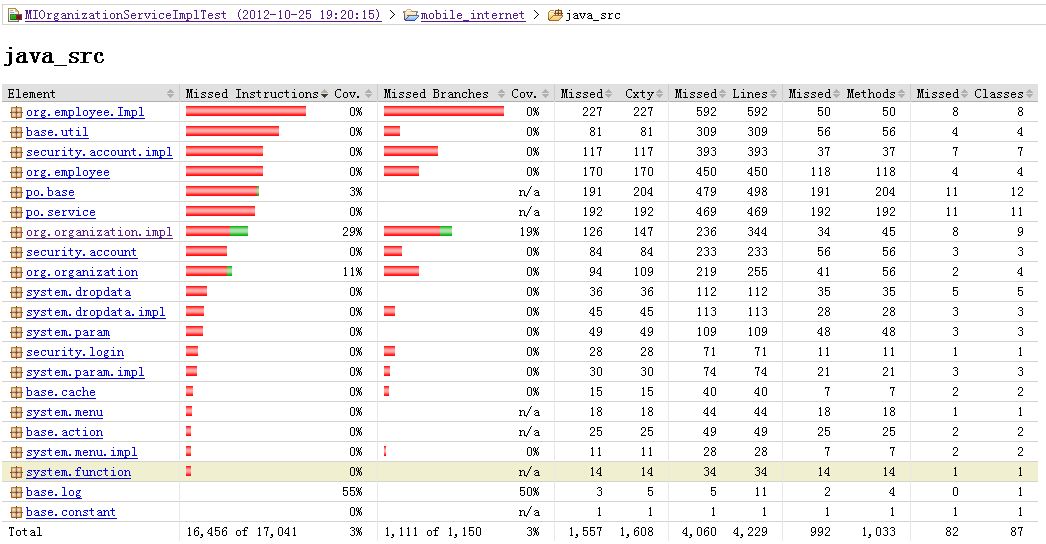


图 430

## 5.总结

每个单元测试，都应该是相互独立的，从而你可以在任何时候以任何顺序运行每一个单元测试。Spring依赖注入，最主要的优点之一就是，使得测试变得更加简单。