EJB 组件的单元测试方法研究与实现①

Research and Implementation on Unit Testing Method of EJB Components

陶小玲 张恒锋 杨丰玉 刘琳岚 (南昌航空大学 软件学院 江西 南昌 330063)

摘 要: EJB 是 J2EE 架构中的核心技术,它功能强大、结构复杂且运行在容器内,单元测试 EJB 组件一直是个难题本文在研究 EJB 组件体系结构的基础上,分析了几种 EJB 组件的独立单元测试方法,分别将每种方法应用到实际项目中,并对其进行比较得出各自的优缺点及适用性。

关键词: EJB 组件 单元测试 Junit

1 引言

随着多层次、异构、分布式企业开发的趋势,J2EE 技术依然被很多企业采用。J2EE 技术包括很多组件,开发大型、复杂的企业应用软件需要集成不同的组件,集成前必不可少的是对每个组件进行严格的单元测试。

EJB(Enterprise JavaBean)是 J2EE 的核心组件,由于它的结构复杂且运行在容器内,单元测试 EJB 组件一直成为难点[1]。使用 Junit、Junitee、Cactus 等等测试框架对 EJB 单元测试都是可行的,但是不同的 EJB 组件的特点,最优测试方法的选择决定了测试的质量。对 EJB 进行单元测试可分容器内测试和独立单元测试。在容器内,使用 Cactus、Junitee 可有效的对 EJB 进行单元测试,对某些特定的 EJB 使用 Junit 也是可行的。本文讨论 EJB 组件的独立单元测试方法,通过分析 EJB 体系结构,介绍 Junit 测试框架的特点以及几种 EJB 组件独立单元测试的方法,并应用到项目开发中分析比较各种方法的优缺点和适用性。

2 EJB体系结构

EJB 分布式应用系统是基于对象组件模型的,包含了处理企业数据的应用逻辑,是分布式事务处理的企业级应用程序组件。J2EE 规范把它划分为业务逻辑层,用于响应客户端的服务请求并进行业务数据的处理[2]。

2.1 EJB 的组成

EJB 分为三种不同类型的 EJB 组件[3]: Session Bean(会话 Bean 分为有状态和无状态的 Session Bean)、Entity Bean(实体 Bean 分为容器持久化 Bean 和管理持久化 Bean)、Message-Driven Bean(消息驱动 Bean)。EJB 的组成部分主要包括了 EJB 容器、EJB 服务器、Home 接口、Remote 接口等[4]。

(1) EJB 容器是一个管理一个或多个 EJB 类或实例的抽象,通过规范中定义的接口使 EJB 类访问所需的服务,也能在容器或服务器中提供额外服务的接口。

(2) EJB服务器提供对系统服务的访问,并管理EJB容器的高级进程或应用程序,也可提供自己的特性。每个 EJB 服务器都支持可访问 JNDI(Java Naming and Directory Interface)的名字服务和事务服务。

(3) Home 接口通过 Home 对象来实现,必须包含所有定位、创建、删除 EJB 类实例的方法,由 EJB 容器提供 Home 接口产生 Home 对象实现的方法。

(4) Remote 接口是开发 EJB 类开发者定义的,并由 EJB Object 来实现,包含了 EJB 类中的商业方法。客户端应用程序通过 Home 对象来定位、创建、删除 EJB 类的实例,并通过 EJB Object 来调用实例中的商业方法。

① 基金项目:国家自然科学基金(60773055)
收稿时间:2008-11-18

2.2 EJB 的调用关系

客户端与服务器端之间的通信使用了 RMI-IIOP 通信协议。EJB 利用 JNDI 来解决名称问题，首先客户端使用 lookup 方法查找 JNDI，然后 EJB 服务器根据登录过的信息来匹配 lookup 的查询内容。客户端必须知道 EJB 组件的 IP、端口和名称才能调用 EJB 组件，最终得到对 EJB 组件对象的引用[5]。EJB 的调用过程如图 1 所示。

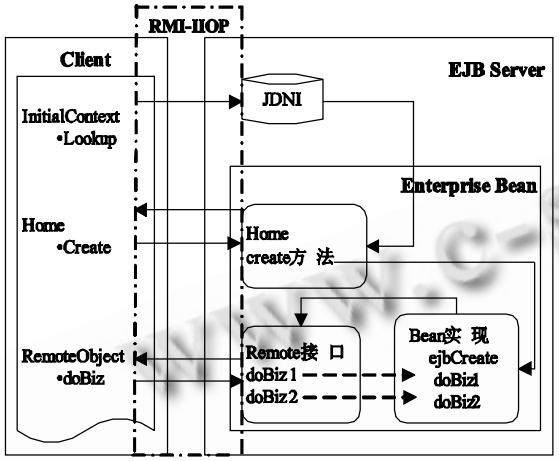

图 1　EJB 调用关系图

3　Junit单元测试框架

Junit 是一个优秀的开源单元测试框架，已成为测试框架的事实标准[6]。Junit 可以使测试代码与被测代码分离，遵守 Junit 编写测试代码的约定就可以对测试代码编写测试，只需要改动少量代码就能实现测试用类在其它类中的重用。Junit 框架设计良好且易扩展，如 Cactus、Junitee 都是 Junit 的扩展框架。Junit 主要包括 framework、runner、textui、swingui、awtui 五个包[7]，framework 是 Junit 的基础框架，包含了 Junit 测试类所需要的所有基类，如图 2 所示。

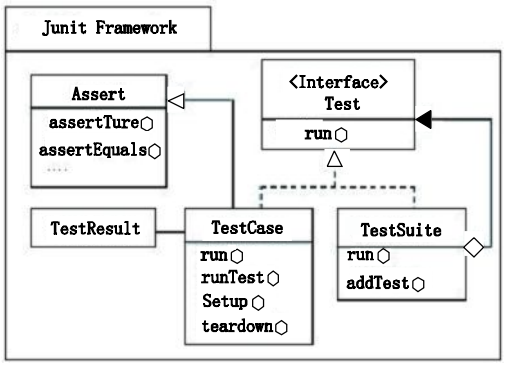

图 2　framework 结构图

用户用 Junit 编写测试用例一般从继承 TestCase 开始，在 TestCase 子类中重载 setUp()和 tearDown()方法，并且定义被测试类的每一个测试方法的测试方法。一般被测方法标识前加“test”作为测试方法，并定义公有属性。同时在此子类中使用断言(Assert)、测试驱动(TestRunner)、集合测试(TestSuite)和测试输入(TestResult)。对不同性质的测试对象，需要运用不同的测试方法来使用 Junit 进行测试。mock objects 可以隔离被测试代码之间、被测试代码和测试代码之间的关联程度。EJB 组件在容器内执行且代码之间存在粘联执行，对它们进行单元测试需要引入 mock objects 来简化测试结构。

4　独立单元测试EJB组件

对 EJB 组件进行独立单元测试需要把测试代码隔离并抽象出容器服务且方法多种多样，针对特定的 EJB 代码，方法的选择成为影响测试质量的主要因素。

4.1 EJB 应用程序

精简开发项目——基于 B/S 的综合信息管理系统的 Session Bean 应用程序为例：

```
public class SCEJB implements SessionBean{
//只用一个方法为例
public int crtApp(Date appD, String appIt)
throws RemoteException{
//创建一个 AppEJB EJB 实例
AppLocal app = AppUt.crtApp(appD, appIt);
//发送申请 ID 到 JMS 队列
try{JMSUt.sTJMS(JNDINames.Q_APP，app.getAppId()，false); }
catch(Exception e){throw new EJBException(e);}
return app.getAppId().intValue();}}
```

Session Bean 中的 crtApp 方法调用了两个静态方法 AppUt.crtApp 和 JMSUt.sTJMS：

(1) AppUt 类包含一个受保护的 appEJB.getApp- Home()方法，可以通过它来获取 App EJB home 实例，并且提供了公共静态 helper 方法 crtApp()和 getApp()，crtApp()方法创建了一个 AppEJB 实例，getApp()通过申请 ID 得到对应的 AppEJB。

(2) JMSUt 类定义了一个单独的静态方法 sTJMS，

通过它发送对象到 JMS 队列中。

SCEJB 使用了 JNDI(Java Naming and Directory Interface)，为了提高性能，EJB home 放在静态变量里且 Session Bean 的代码由静态方法构成，但 JNDI API 没有遵守 IOC(Inversion of Control)模式，要访问 JNDI 对象要创建一个 InitialContext(new Initial Context())，这些都增加了单元测试 SCEJB 的难度。

4.2 单元测试方法

对 SCEJB 进行单元测试主要是 crtApp 方法的测试，下面分别描述模拟 JNDI 实现方法、Factory 类方法来进行单元测试。

4.2.1 模拟 JNDI 实现方法

模拟 JNDI 实现方法通过创建 JNDI 实现，并在任何调用 lookup 方法的时候都返回一个模拟对象来简化单元测试，被测代码不受影响。因为 JNDI 提供了良好的 API，添加 JNDI 实现可以使用多种方法，如应用程序资源文件、System 属性、NamingManager 的静态方法 setInitialContextFactoryBuilder API 等等。选择 NamingManager 来测试 SCEJB，因为这个 API 可以设置一个初始的 Context Factory，每调用一次 new InitialContext 都会调用这个 Factory。但是 setInitialcontextFactoryBuilder 方法在 JVM 的生命周期里只能调用一次，所以在 Junit TestSetup 里添加这个方法的初始化。模拟 JNDI 实现创建后，编写单元测试就变得简单了：

```
public class TestSCEJB extends TestCase{
public static Test suite(){
TestSuite suite = new TestSuite();
suite.addTestSuite(TestSCEJB.class);
//利用 JNDITestSetup 类设置模拟 JNDI 实现
jndiTestSetup =new JNDITestSetup(suite);
return jndiTestSetup;}
//把所有的 mock 创建工作放进 setUP 方法
protected void setUp() throws Exception{
sc = new SCEJB(){};
//由于整个测试中共享一个 MockContext，所以 jndiTestSetup 变量是静态的，每调用一次 jndi TestSetup.getMockContext() 返回同一个 MockContext 实例
jndiTestSetup.getMockContext().reset();
setUpAppMocks();
setUpJMSMocks();
setUpJNDILoopups();
jndiTestSetup.getMockContext().matchAndReturn("close",null);}
……//创建所有的 mock，定义它们在所有测试中的预定行为
//在 tearDown 方法中，检验该对象设置的预期
protected void tearDown()}
```

4.2.2 Factory 类方法

Factory 类方法通过在工厂类中隔离领域对象的创建，并为这些工厂添加 setter 方法引入要测试方法所需要的所有领域对象，然后调用这些 setter，将测试类的 mock objects 传给它们。为了单元测试 SCEJB，Factory 类方法要要把 setAppUt(AppUt)和 setJMSUt(JMSUt)这两个受保护的方法引入到 SCEJB 类中，通过创建两个接口：AppUt 和 JMSUt 接口来实现。

```
/*创建 AppUt 接口：*/
public interface AppUt{}
public class DefaultAppUt implements appUt{}
//把 AppUt 类更名为 DefaultAppUt
/*创建 JMSUt 接口：*/
public interface JMSUt{}
public class DefaultJMSUt implements JMSUt{}
```

创建了 AppUt 和 JMSUt 接口后，为了能在 create App 方法中使用，要在 SCEJB 类中再添加两个静态的私有变量来保留默认的 AppUt 和 JMSUt 领域对象：

```
private Static AppUt appUt =new Default－AppUt();
private Static JMSUt JMSUt = new DefaultJM－SUt();
```

由于新的领域对象和默认的是不一样的，要添加两个方法来设置 AppUt 和 JMSUt 领域对象：

```
protect void setAppUt(AppUt factory){appUt = factory;}
protect void setJMSUt(JMSUt util){JMSUt = util;}}
```

最后 SCEJB 的单元测试中可以直接用 AppUt 和 JMSUt 接口来简化测试：

```
public class TestSCEJB extends TestCase{
```

```
protected void setup(){sc = new SCEJB();
mockAppLocal = new Mock(AppLocal.clas-
s);
appLocal = (AppLocal)mockAppLocal.prex-
y();
//调用 AppUt 接口和 JMSUt 接口创建 mock
objects
mockAppUt = new Mock(AppUt.class);
appUt = (AppUt) mockAppUt.proxy();
mockJMSUt = new Mock(JMSUt.class);
JMSUt = (JMSUt) mockJMSUt.proxy();
//调用引入的 setter 方法设置 SCEJB 类中的
mock objects
sc.setAppUt(appUt);sc.setJMSUt(JMSUt);
……//mock objects 如何对不同的测试处理
//在 tearDown 方法中，检验该对象设置的预期
protected void tearDown()}
```

对比模拟 JNDI 实现方法和 Factory 类方法，Factory 类方法编写速度快，代码的质量和灵活性提高了，模拟 JNDI 实现方法则需要更多的时间，所有使用的领域对象都需要模拟，相应就增加了 setUp 代码长度。但是 Factory 类方法重构量大，还需要分别对 AppUt.crtApp 和 JMSUt.sTJMS 方法编写测试，且用户提供的 factory 不适应 EJB 模型，这种方法对更适合在单元测试使用少量领域对象的 EJB 代码中，对结构复杂的 EJB 并没提高测试效率相对让测试变得复杂繁琐。模拟 JNDI 方法代码重构量不大，且为所有的 EJB 单元测试提供一个公共的 fixture 配置，不仅对 SCEJB.crtApp 方法进行了单元测试，而且同时测试了 AppUt.crtApp 和 JMSUt.sTJMS 方法，也测试了 SCEJB、AppUt 和 JMSUt 三者之间的关系，相较 Factory 类方法，这种方法使用在重构量大且结构复杂的 EJB 中更能发挥它的优越性。

5 结束语

单元测试 EJB 组件是非常重要的，本文分析了单元测试 EJB 的测试方法——模拟 JNDI 实现方法和 Factory 类方法，并分析比较它们的优缺点及适用性。除了这两方法，还有很多测试方法，它们都有各自的特点，针对不同的 EJB，通过分析比较找出最优测试方法才能更好的提高单元测试效率，从而更有效的减少软件开发时间、更有利的保证软件质量。

参考文献

1 Vincent M, Ted H. Junit in action. http://www. ebookcn.net.

2 徐刚,应时,袁胜琼.基于设计模式的 EJB 体系结构模型.计算机工程与应用, 2003,(28):136－138.

3 Roman ED, Sriganesh RP, Wiley GB. Mastering Enterprise JavaBeans, 2004.

4 陈立岩.EJB 组件技术及应用.计算机技术与发展, 2007,(3):62－64.

5 Matena V. EJB 应用指南.北京:清华大学出版社,2004.

6 王东刚.软件测试与 Junit 实践.北京:人民邮电出版社, 2004:100－104.

7 Erich G, Beck K. Junit version3.8.1. http://www.junit.org.