# 1 需求分析

# 1.1 需求描述

从Blade的架构来看，Blade提供了相当充足的功能用于简化Java开发。但从开发过程来看，Blade尚有不足。其中最为明显的地方在于Blade并未针对测试提供解决方案。世界上不存在没有漏洞的系统，无一例外地都会在运行时都会出现错误、异常以及其他非期待的运行结果。想要尽可能的减少运行时出现的问题，提前的测试是开发中必不可少的一环。软件测试能够提前发现生产环境中可能出现的问题并提前处理，避免财产、生命的损失。根据软件工程和软件测试的规范，开发人员应先进行单元测试。单元测试中的黑盒测试将测试用例的输入作为模块的输入，将预期结果与和模块输出相比较来判断模块工作是否正常。

人工对一整个系统的测试时非常耗时费力的，因此开发人员可以选用各种常用的测试框架对程序进行测试。JUnit是Java语言中最为广泛使用和强力的测试框架，其能够提供Java运行必须的测试环境，并运行指定的待测试方法。JUnit能够采用断言的方式判断预期结果是否与模块输出结果相同，并循环或批量对不同的方法进行测试。这给开发者测试提供了极大的便利。

Blade是基于IOC技术的JavaWeb框架，其特性就在于开发者无需声明变量即可使用。但IOC必须在Blade框架完全启动才能生效，JUnit无法自动生成IOC环境，因此无法对基于Blade框架编写的应用程序进行测试。这使得Blade开发的应用程序变得不安全，我们小组认为易用的JavaWeb框架不应该有此缺陷，因此致力于完善Blade框架在测试方面的不足。

# 1.2 RUCM建模

测试不应该加重开发者的工作负担，因此借助IOC容器应该同样能够提供简单有效的测试方案。结合Spring的测试方案，使用RUCM描述测试用例的行为模型如图 1：

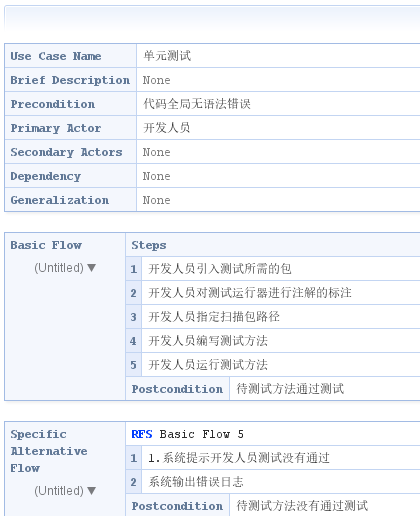


图 1 单元测试用例RUCM（a）

# 1.3 需求目标

根据需求描述，小组计划给予Blade的基本结构设计一套测试组件BladeTest，期望能够满足开发人员的日常测试功能。BladeTest应满足如下要求：

1）开发人员无需关心IoC容器细节，可在测试环境内直接使用IOC容器。

2）开发人员能够在测试环境内获取IOC容器内容

3）IOC容器在单元测试开始和结束时自动初始化

4）测试可以批量进行

5）尽可能减少与Blade的重复代码，保证同步更新的可能性

6）添加其他方便开发人员测试的功能并保证兼容性

# 2 可行性分析

作为一个Java语言开发的框架，Blade本应能与JUnit之间共同合作。但由于IOC容器的缺失，JUnit与Blade之间无法正常的交互。小组认为手动在JUnit运行时提供适合Blade使用的IOC容器能够解决该问题。

小组参考了Spring的Spring是JavaWeb框架的领头羊，其在SpringTest子项目中提供了基于JUnit的测试方案。JUnit在运行之前会通过@RunWith注解获取Spring默认IOC容器，该IOC进针对测试类进行管理，足够支持待测试方法的运行和检查。因此小组认为这是一条可行的技术方案。

# 3 测试模块类图

测试模块主要有4个类图，绘制类图如图 3所示。其中主要对运行器类进行测试，一个包可以包含0到多个运行器类，测试类主要用于测试方法的编写，日志类用于输出测试过程中的日志信息。具体类图如图2所示。

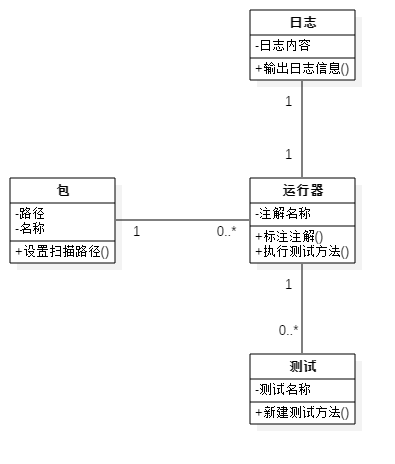


图 2测试模块类图

# 3.1 测试模块类图与RUCM对应表

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 类图名称 | RUCM名称 | 基本事件流 | 对应类图中的类 | 属性/方法 | 描述 |
| 测试模块类图 | 单元测试 | Step2 | 运行器 | 标注注解（） | 进行注解的标注 |
| 测试模块类图 | 单元测试 | Step3 | 包 | 设置扫描路径（） | 指定扫描包路径 |
| 测试模块类图 | 单元测试 | Step4 | 测试 | 新建测试方法（） | 编写测试方法 |
| 测试模块类图 | 单元测试 | Step5 | 运行器 | 运行测试方法（） | 开发人员运行测试方法 |
| 测试模块类图 | 单元测试 | RFS 2 | 日志 | 输出日志信息（） | 系统输出错误日志信息 |

图 3 测试模块类图与RUCM对应表

# 4 测试模块时序图

开发人员首先在blade系统引入测试所需的包，之后在运行器中进行注解的标注，第三步设置包的扫描路径，之后再进行测试方法的编写与测试，在测试过程中，输出测试过程的日志信息。

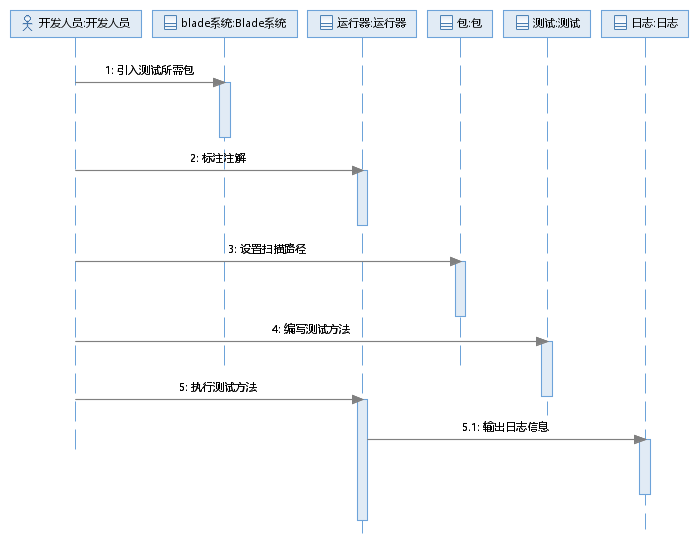


图 4 测试模块时序图

# 5 测试模块状态图

对于测试过程，程序在初始状态属于空闲状态，在开发人员选择执行测试时系统转化为忙碌状态，在测试完成并未关闭系统之前系统将转化为空闲状态，当开发人员关闭系统时，状态图结束。

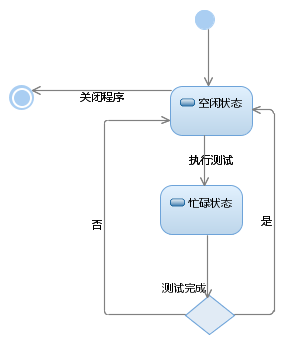


图 5 测试模块状态图

# 6. 模块划分

考虑到小组的工作是在JUnit和Blade之前添加连接，因此我们采用适配器模式进行开发。因此划分模块如下：

1）BladeTestIOC类是Blade框架的对象适配器，能够提供初始化接口、清零接口供JUnit调用。保证JUnit无需知道Blade工作的细节。

2）BladeTestIOC内部通过Blade框架的启动器对IOC容器进行初始化。

3）BladeRunner4JUnit实现了IOC容器的初始化的清零，继承了JUnitRunner接口，能够向JUnit注入预操作。

4）BladeConfiguration注解，用于从BladeTest中获取配置参数（扫描包位置等）

# 7.测试

软件测试是描述一种用来促进鉴定软件的正确性、完整性、安全性和质量的过程。据此，您可能会想，软件测试永远不可能完整的确立任意电脑软件的正确性。然而，在可计算理论（计算机科学的一个支派）一个简单的数学证明推断出下列结果：不可能完全解决所谓“死机”，指任意计算机程序是否会进入死循环，或者罢工并产生输出问题。换句话说，软件测试是一种实际输出与预期输出间的审核或者比较过程。

软件测试有许多方法，但对复杂的产品运行，有效的测试不仅仅是研究产品的运行，更为重要的是制定并严格遵守软件相关的一些规范。测试的其中一个定义：为了评估而质疑产品的过程；这里的“质疑”是测试人员试着对产品做的事，而产品以测试者脚本行为反应作为回答。虽然大部分测试的智力过程不外乎回顾、检查，然而“测试”这个词意味着产品动态分析──让产品流畅运行。程序质量可能，而且通常会随系统不同而有差异；不过某些公认特性是共通的：[**可靠性**](https://zh.wikipedia.org/wiki/%E5%8F%AF%E9%9D%A0%E6%80%A7)、[**稳定性**](https://zh.wikipedia.org/wiki/%E7%A9%A9%E5%AE%9A%E6%80%A7)、[**轻便性**](https://zh.wikipedia.org/w/index.php?title=%E8%BC%95%E4%BE%BF%E6%80%A7&action=edit&redlink=1)、[**易于维护**](https://zh.wikipedia.org/w/index.php?title=%E6%98%93%E6%96%BC%E7%B6%AD%E8%AD%B7&action=edit&redlink=1)、以及[**实用性**](https://zh.wikipedia.org/w/index.php?title=%E5%AF%A6%E7%94%A8%E6%80%A7&action=edit&redlink=1)。

对于Blade框架的测试，根据我们的需求设计，我们需要对其实现是否满足需求进行确认。确认的内容包括两方面，一方面是对Blade原始框架部分的确认，另一部分是对改进部分即BladeTest测试框架部分的确认。

## 7.1 BladeTest

### 7.1.1测试需求

根据我们的改进方案，即BladeTest的设计目标：即由于现有单元测试框架JUnit无法作用于以IOC容器实例化对象的环境，无法对基于Blade框架编写的应用程序进行集成环境下的单元测试，BladeTest就是为了解决当前这一困境而存在的。

根据Blade的注解驱动以及IOC对象实例化机制，以及BladeTest对JUnit的扩展，因此在BladeTest的测试中，需要对如下三个方面进行测试。

1. 在引入BladeTest后，JUnit可以在BladeRunner4JUnit的环境中正常工作。
2. BladeTest进行单元测试时，系统能够IOC实例化所有标记的对象。
3. BladeTest的单元测试类中可以使用@Inject注解注入已经通过IOC实例化了的对象。
4. BladeTest的单元测试能够较好的适应于Blade框架的MVC设计架构。

### 7.1.2 测试用例设计

对于测试需求(1)，考虑到JUnit框架本身的成熟性，故而一旦我们可以确认BladeTest能够顺利的整合JUnit框架就可以确认其正常工作，一般而言，如果可以通过编译及部署就可以认为其满足了测试需求（1）。

对于测试需求（2），我们需要确认集成了BladeTest测试框架的Blade框架是否能够维持IOC实例化对象这种机制。根据此种需要，我们可以把原来使用Blade框架构建对象实例的应用经过简单的修改，将对象实例化的入口置于BladTest下。观测所有对象是否正常实例化。如果同原始Blade框架一样，所有的对象都被正常实例化，那么我们就可以认为其满足了测试需求（2）。

对于测试需求（3）,我们需要确认注解@Inject是否正常工作，是否能够将IOC实例化的对象注入到我们的单元测试类中。如果我们能够确认所有的测试对象已经被实例化，那么我们就可以确认注解@Inject正常工作，即可以认为满足了测试需求（3）。

对于测试需求（4),我们需要确认BladeTest单元测试在MVC设计架构下能否较好工作，MVC开发时可以设计如下四个包结构：repository，service，controller，test其中repository封装数据库操作，service封装业务逻辑，controller封装控制逻辑。如果需要对某个实例或对某个流程进行测试时，可以为此在test包下编写该特定测试用例。

7.1.3 测试结果

简要列出一些测试用例。

测试用例1：helloController，helloService，helloRepository的测试输出如图7.1所示。测试表明IOC实例化正常。满足了测试需求（1）、（2）、（3）；

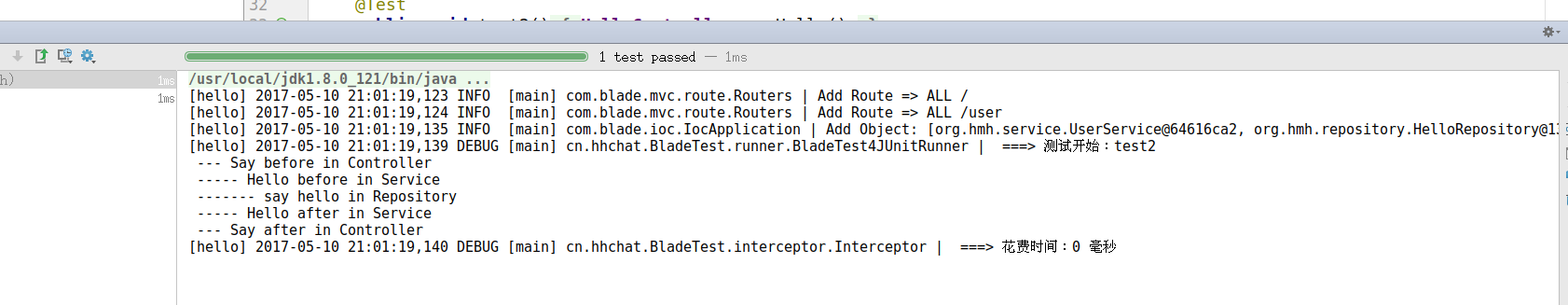


图7.1 测试用例1

测试用例2： userController，userService，userRepository的测试输出如下。其中userController通过路由匹配控制，使用userService服务，userService使用userRepository的数据操作。测试输出如图7.2所示。

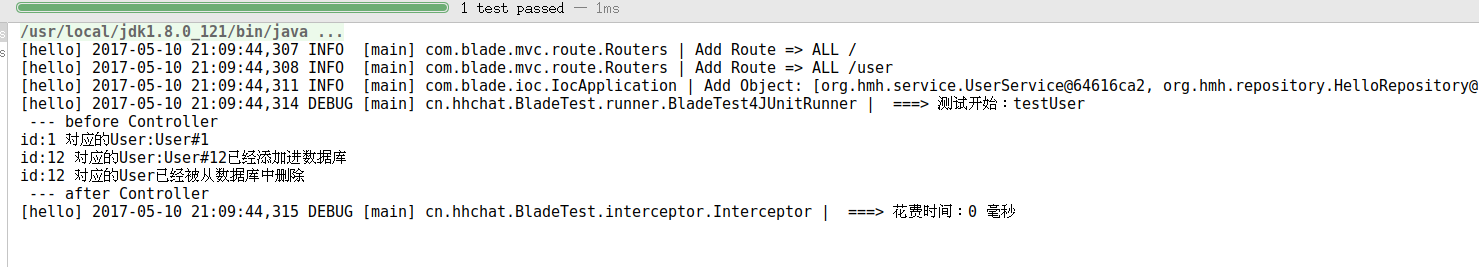


图7.2 测试用例2

测试的初步结果表明，BladeTest测试框架满足它的用户需求，达到了设计目标。

## 7.2 Blade测试

根据制定的用户需求文档，我们需要对Blade软件需求文档的业务需求，功能需求，以及各种非功能需求进行测试。下面初步制定如下三个部分。具体内容后续补充。

### 7.2.1业务需求测试

### 7.2.2 功能性需求测试

### 7.2.3 非功能性需求测试