ssm 框架整合及 Rest 风格接口实现

1.前言

SSM 框架整合即 Spring+SpringMVC+Mybatis 三大框架进行整合。属于现在 Java 项目主流的框架选择。
关于这三大框架的介绍，本篇幅不做说明。项目中结合了 Maven 的使用。
关于 Eclipse 构建 Maven 项目详情文章：
http://blog.csdn.net/it_faquir/article/details/54562242

2.Maven 依赖

首先创建一个 Maven 的 Web 项目。这里将其命名为“SSMIntegration”及 SMM 整合。

通过 Maven 的 pom.xml 加入各所需 jar 包。

```
1.      <properties>
2.          <spring-version>4.3.5.RELEASE</spring-version>
3.       </properties>
4.       <dependencies>
5.          <dependency>
6.              <groupId>junit</groupId>
7.              <artifactId>junit</artifactId>
8.              <version>4.12</version>
9.              <scope>test</scope>
10.         </dependency>
11.         <!--    https://mvnrepository.com/artifact/javax.servlet/servlet-api
-->
12.         <dependency>
13.             <groupId>javax.servlet</groupId>
14.             <artifactId>servlet-api</artifactId>
15.             <version>2.5</version>
16.         </dependency>
17.         <!-- spring & mvc start -->
18.         <!-- http://projects.spring.io/spring-framework/ -->
19.         <dependency>
20.             <groupId>org.springframework</groupId>
21.             <artifactId>spring-context</artifactId>
```

```
            <version>${spring-version}</version>
        </dependency>
        <!--
https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>${spring-version}</version>
        </dependency>

        <!--
https://mvnrepository.com/artifact/org.springframework/spring-test -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <version>${spring-version}</version>
        </dependency>

        <!-- datasource 必须 -->
        <!--
https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>
            <version>${spring-version}</version>
        </dependency>
        <!-- spring & mvc end -->


        <!-- mybaits start -->
        <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.4.1</version>
        </dependency>
        <!-- spring 整合 mybatis 必备 -->
        <!--   https://mvnrepository.com/artifact/org.mybatis/mybatis-spring
-->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>1.3.0</version>
        </dependency>
```

```
        <!-- mybatis end -->

        <!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.0.26</version>
        </dependency>
        <!--   https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>6.0.5</version>
        </dependency>

        <!-- 使用@RequestBody @ResponseBody 时得用到下面两 jar 包 -->
        <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-core</artifactId>
            <version>2.8.5</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.8.5</version>
        </dependency>
    </dependencies>
```

上面加入了 Spring/SpringMVC/Mybatis 及其它所需的依赖，其中 `jackson-core` 和 `jackson-databind` 千万别忘了，否则你会发现 Spring 无法解析传来 json 数据，具体请看注释。

3.各种配置

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
 version="2.5">

 <display-name>SSHIntergration</display-name>
 <welcome-file-list>
     <welcome-file>index.jsp</welcome-file>
 </welcome-file-list>

 <!-- needed for ContextLoaderListener -->
 <context-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>classpath:applicationContext.xml</param-value>
 </context-param>

 <!-- Bootstraps the root web application context before servlet
initialization -->
 <listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</l
istener-class>
 </listener>

 <!-- SpringMVC servlet -->
 <servlet>
     <servlet-name>springDispatcherServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servle
t-class>
     <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>classpath:applicationContext-web.xml</param-value>
     </init-param>
     <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
     <servlet-name>springDispatcherServlet</servlet-name>
     <url-pattern>/</url-pattern>
 </servlet-mapping>

 <!-- 防止乱码 -->
```

```
        <filter>
            <filter-name>encodingFilter</filter-name>
            <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
            <init-param>
                <param-name>encoding</param-name>
                <param-value>UTF-8</param-value>
            </init-param>
            <init-param>
                <param-name>forceEncoding</param-name>
                <param-value>true</param-value>
            </init-param>
        </filter>
        <filter-mapping>
            <filter-name>encodingFilter</filter-name>
            <url-pattern>/*</url-pattern>
        </filter-mapping>
        <!-- 用来过滤 rest 中的方法，在隐藏域中的 put/delete 方式,注意 由于执行顺序原因 一定要放在编码过滤器下面，否则会出现编码问题 -->
        <filter>
            <filter-name>HiddenHttpMethodFilter</filter-name>
            <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
            <init-param>
                <param-name>methodParam</param-name>
                <param-value>_method</param-value>
            </init-param>
        </filter>
        <filter-mapping>
            <filter-name>HiddenHttpMethodFilter</filter-name>
            <url-pattern>/*</url-pattern>
        </filter-mapping>
    </web-app>
```

需要注意：

1.html 不支持 put、delete 的方式，但 Spring mvc 支持 REST 风格的请求方法，GET、POST、PUT 和 DELETE 四种请求方法分别代表了数据库 CRUD 中的 select、insert、update、delete。因此想要实现四种请求，需要在表单中使用隐藏域，并在 web.xml 中配置 HiddenHttpMethodFilter 过滤器。

2.为了防止乱码，因此需要通过 CharacterEncodingFilter 实现全局编码过滤。注意一点要放在 web.xml 内容中的其它过滤器上方，否则由于执行顺序原因，导致依然出现乱码问题，具体原因需进一步探讨。

做好基本的准备之后，接下来进行具体的代码编写

在根目录下创建
applicationContext-web.xml　　SpringMVC 配置文件
applicationContext.xml　　Spring 配置文件
jdbc.properties　　jdbc 配置文件
mybatis.cfg.xml　　mybatis 的配置文件
在 web.xml 中 SpringDispatcherServlet 指定 SpringMVC 的配置位置，contextConfigLocation 指定 Spring 的配置位置。四个文件其具体内容如下，详情请看到代码中的注解。

applicationContext-web.xml ：

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.   xmlns:context="http://www.springframework.org/schema/context"
5.   xmlns:mvc="http://www.springframework.org/schema/mvc"
6.   xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
7.       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
8.       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
9.   <context:component-scan base-package="priv.hgs.ssm.controller" >
10.      <context:include-filter                    type="annotation"
expression="org.springframework.stereotype.Controller"/>
11.  </context:component-scan>
12.  <!--配置只要解析器-->
13.  <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
14.      <property name="prefix" value="/WEB-INF/"/>
15.      <property name="suffix" value=".jsp"/>
16.  </bean>
17.  <mvc:annotation-driven/>
18.
19.  </beans>
```

applicationContext.xml ：

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
4.   xmlns:c="http://www.springframework.org/schema/c"
```

```
5.          xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
6.              http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
7.          <context:component-scan base-package="priv.hgs.ssm">
8.              <context:exclude-filter type="annotation"
9.                  expression="org.springframework.stereotype.Controller" />
10.         </context:component-scan>
11.         <!-- 加载配置文件 -->
12.         <bean
13.             
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfi
gurer">
14.             <property                                          name="location"
value="classpath:jdbc.properties"></property>
15.         </bean>
16.         <!-- 配置数据源 -->
17.         <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
18.             init-method="init" destroy-method="close">
19.             <property name="driverClassName" value="${jdbc.driver}" />
20.             <property name="url" value="${jdbc.url}" />
21.             <property name="username" value="${jdbc.username}" />
22.             <property name="password" value="${jdbc.password}" />
23.             <!-- 初始化连接大小 -->
24.             <property name="initialSize" value="${initialSize}"></property>
25.             <!-- 同一时间连接池最大数量 0 则无限制 -->
26.             <property name="maxActive" value="${maxActive}"></property>
27.             <!-- 连接池最大空闲 池里不会被释放的最多空闲连接数 0 则无限制 -->
28.             <property name="maxIdle" value="${maxIdle}"></property>
29.             <!-- 连接池最小空闲 在不创建新连接的情况下，池中保持空闲的最小连接数 -->
30.             <property name="minIdle" value="${minIdle}"></property>
31.             <!-- 获取连接最大等待时间 -->
32.             <property name="maxWait" value="${maxWait}"></property>
33.         </bean>
34.
35.         <!-- DAO 接口所在包名，Spring 会自动查找其下的类,,这里通过配置文件的方式 -->
36.         <bean                                          id="sqlSessionFactory"
class="org.mybatis.spring.SqlSessionFactoryBean">
37.             <property name="dataSource" ref="dataSource" />
38.             <!-- 添加 mybatis 的配置文件，如果没有可以不添加 -->
39.             <property                                      name="configLocation"
value="classpath:mybatis.cfg.xml"></property>
40.         </bean>
41.
```

```
    <bean class="org.mybatis.spring.SqlSessionTemplate"
        c:sqlSessionFactory-ref="sqlSessionFactory">
    </bean>
</beans>
```

jdbc.properties:

```
jdbc.driver=com.mysql.cj.jdbc.Driver
#定义连接的 URL 地址，设置编码集，时间域，允许多条 SQL 语句操作
jdbc.url=jdbc:mysql://localhost:3306/dbtest?characterEncoding=utf-8&serverTimezone=UTC&allowMultiQueries=true
jdbc.username=root
jdbc.password=asd123asd
#jdbc.password=StrLDvcH92s8tsn3
#定义初始连接数
initialSize=0
#定义最大连接数
maxActive=20
#定义最大空闲
maxIdle=20
#定义最小空闲
minIdle=1
#定义最长等待时间
maxWait=30000
```

4.相关数据表

本次测试使用到了 MySQL 数据库，在数据库中创建了两种表，分别为 student 表、score 表,其 SQL 如下：

student 表

```
CREATE TABLE `student` (
  `id` int(12) NOT NULL AUTO_INCREMENT,
  `name` varchar(32) NOT NULL,
  `password` varchar(32) NOT NULL,
  `gender` int(2) NOT NULL DEFAULT '1' COMMENT '0-女；1-男',
  `grate` varchar(32) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

score 表

```
CREATE TABLE `score` (
  `scid` int(11) NOT NULL,
  `course` varchar(10) NOT NULL,
  `score` int(3) unsigned zerofill NOT NULL,
  `stid` int(11) NOT NULL,
  PRIMARY KEY (`scid`),
```

```
  KEY `stukey` (`stid`),
  CONSTRAINT `stukey` FOREIGN KEY (`stid`) REFERENCES `student` (`id`) ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

在上面两种表中，需要注意的是，score 的 scid 设置了外键为 student 表中的 id,并且设置了级联删除和级联更新。

5.MyBatis 上场

一切就绪，java 代码该上场了。
首先针对以上两张表在 domain 包下创建两个 javaBean。
Student.java

```
public class Student {
 private int stid;
 private String name;
 private String password;
 private int gender;
 private String grate;
 private List<Score> scores;// 一对多
       get/set...
       toString...
}
```

Score.java

```
public class Score {
 private int id;
 private int stid;
 private String course;
 private int score;
 get/set...
 toString...
}
```

编写 mapper 文件
在 mapping 包下创建一个 SSMHelloMapper.xml 的关系映射文件
我这里，分别列出了增删改查的示例，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="priv.hgs.ssm.dao.IHelloDao">
 <resultMap type="Student" id="Stu">
     <id property="stid" column="id" />
```

```
        <result property="name" column="name" />
        <result property="password" column="password" />
        <result property="genter" column="genter" />
        <result property="grate" column="grate" />
        <!-- 一对多关系设置,集合 .注意 两个表的 id 最好不要一样，否则只能查到一条数据，也可以通过别名解决 -->
        <collection property="scores" ofType="Score">
            <id column="scid" property="id" />
            <result property="course" column="course" />
            <result property="score" column="score" />
            <result property="stid" column="stid" />
        </collection>
    </resultMap>

    <!-- 对于数据表与实体类属性不一致情况，使用 resultMap 进行处理 -->
    <!--这里不适合使用内连接的方式，即 SELECT st.*,sc.* FROM student st,Score sc WHERE st.id
        = #{id} and st.id = sc.sid; 因为：当 score 没有对应值时，将查不出信息。 所有：最好使用左连接 -->
    <select id="getStuById" resultMap="Stu">
        SELECT st.*,sc.* FROM student
        st LEFT JOIN score sc ON st.id = sc.stid WHERE st.id=#{id};
    </select>

    <delete id="delStuById">
        DELETE FROM student WHERE id = #{stid};
    </delete>

    <update id="updateStu" parameterType="Map">
        UPDATE student
        <set>
            <if test="password != null">
                password = #{password},
            </if>
            <if test="grate != null">
                grate = #{grate}
            </if>
        </set>
        where id = #{stid};
    </update>

    <insert id="insetStu" parameterType="Student">
        INSERT INTO
        student(name,password,gender,grate)
```

```
        VALUES(#{name},#{password},#{gender},#{grate});
    </insert>
</mapper>
```

在查询中，演示了左连接的查询方式，由于用内连接有可能对应的数据，在 score 表中不存在的情况，引起异常。

6.dao 的编写

dao 层

```
public interface IHelloDao {
    // 对于其参数名与 mapper 中不一致情况，可适应@Param("名称") 注解进行校正
    Student getStuById(@Param("id") int stid);

    int delStuById(int stid);

    int updateStu(Map param);

    int insetStu(Student student);
}
```

注意：接口名一定要与映射文件中的各 id 的名字一样，否则 MyBatis 无法识别，从而实现半自动化。

dao imp

```
@Component
public class HelloDao implements IHelloDao {
    @Autowired
    SqlSessionTemplate ssTemplate;

    @Override
    public Student getStuById(int stid) {
        return ssTemplate.getMapper(IHelloDao.class).getStuById(stid);
    }

    @Override
    public int delStuById(int stid) {
        return ssTemplate.getMapper(IHelloDao.class).delStuById(stid);
    }

    @Override
    public int updateStu(Map param) {
        return ssTemplate.getMapper(IHelloDao.class).updateStu(param);
```

```
    }

    @Override
    public int insetStu(Student student) {
        return ssTemplate.getMapper(IHelloDao.class).insetStu(student);
    }
}
```

这里使用了 SpringData 的模板，SqlSessionTemplate 也就是 Spring 配置文件中所配的，如下：

```
<!-- DAO 接口所在包名，Spring 会自动查找其下的类,,这里通过配置文件的方式 -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <!-- 添加 mybatis 的配置文件，如果没有可以不添加 -->
    <property name="configLocation" value="classpath:mybatis.cfg.xml"></property>
</bean>
```

通过 SqlSessionTemplate，可以让我们很轻松的实现增删改查。

Spring 常用数据库模板

模板类	用途
jdbc.core.JdbcTemplate	jdbc 连接
com.hibernate3.HibernateTemplate *（org.springframework.orm.hibernate5.HibernateTemplate 最新）*	Hibernate 3.x 以上的 Session
org.mybatis.spring.SqlSessionTemplate	Mybatis 模板
org.springframework.data.redis.core.StringRedisTemplate	Redis
org.springframework.data.mongodb.core.MongoTemplate	Mongodb
orm.jdo.JdoTemplate	Java 数据对象（Java Data Object）实现
orm.jpa.JpaTemplate	Java 持久化 API 的实体管理器

到了这一步，可以说已经完成了大部分工作，我们来测试下所写的代码正不正确。

7.Dao 的 junit 测试

创建一个 HelloTest 类用于我们的单元测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:applicationContext.xml"})
public class HelloTest {
 @Autowired
 IHelloDao helloDao;

 @Test
 public void testQueryStu(){
     Student stu = helloDao.getStuById(1);
     System.out.println(stu);
 }

 @Test
 public void testDelStu(){
     int result = helloDao.delStuById(2);
     System.out.println(result);
 }

 @Test
 public void testUpdateStu(){
     Map<String,String> param = new HashMap<>();
     param.put("stid","1");
     param.put("password", "asd");
     int result = helloDao.updateStu(param);
     System.out.println(result);
 }

 @Test
 public void testInsertStu(){
     Student stu = new Student();
     stu.setName("魔女");
     stu.setGender(0);
     stu.setPassword("asd123asd");
     stu.setGrate("八年二班");
     int result = helloDao.insetStu(stu);
     System.out.println(result);
 }
}
```

测试模块实现了对增删改查操作的测试。

如果测试成功，数据库中的数据将会发生变化，如果没变化，请检查代码是否有误。
注意：该测试类需要 junit 和 spring-test 相关依赖。

8.Service 的编写

web 的实现
对 dao 层的实现基本 OK，接着需要实现 service 层，及 controller 的实现。
service 层和 dao 层差不多，就是在 dao 的基础上套了一层。
IHelloService 接口：

```
1.        public interface IHelloService {
2.         // 对于其参数名与 mapper 中不一致情况，可适应@Param("名称") 注解进行校正
3.         Student getStuById(@Param("id") int stid);
4.
5.         int delStuById(int stid);
6.
7.         int updateStu(Map param);
8.
9.         int insetStu(Student student);
10.       }
```

实现类 HelloService：

```
1.
2.        @Service
3.        public class HelloService implements IHelloService {
4.         @Autowired
5.         IHelloDao helloDao;
6.
7.         @Override
8.         public Student getStuById(int stid) {
9.             return helloDao.getStuById(stid);
10.        }
11.
12.        @Override
13.        public int delStuById(int stid) {
14.            return helloDao.delStuById(stid);
15.        }
16.
17.        @Override
18.        public int updateStu(Map param) {
19.            return helloDao.updateStu(param);
20.        }
21.
```

```
    @Override
    public int insetStu(Student student) {
        return helloDao.insetStu(student);
    }

}
```

注意:这里要用@Service注解用于标注业务层组件(以上我说的dao层和service层，只是为了个人表述方便而起的）。

9.Controller 对 Rest 的实现

控制器：

```
package priv.hgs.ssm.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import priv.hgs.ssm.domain.Result;
import priv.hgs.ssm.domain.Student;
import priv.hgs.ssm.service.IHelloService;

@RequestMapping(value = "/student")
@RestController
public class HelloControler {
    @Autowired
    IHelloService helloService;

    @RequestMapping(value = "/{stid}", method = RequestMethod.GET)
    public Student getStudentById(@PathVariable int stid) {
        Student student = helloService.getStuById(stid);
        if (student == null) {
            student = new Student();
        }
        return student;
    }

```

```
    @RequestMapping(value = "/{stid}", method = RequestMethod.DELETE)
    public Result delStuById(@PathVariable int stid) {
        int r = helloService.delStuById(stid);
        if (r == 0)
            return new Result(403, "删除失败");
        return new Result(200, "删除操作成功");
    }

    @RequestMapping(value = "/addStu", method = RequestMethod.POST)
    public Result addAStu(Student student) {
        int result = helloService.insetStu(student);
        if (result == 0)
            return new Result(403, "添加失败。");
        return new Result(200, "添加成功。");
    }

    @RequestMapping(value="/updateStu",method = RequestMethod.PUT)
    public Result updateStu(Student student){
        System.out.println(student);
        Map<String,String> param = new HashMap<>();
        param.put("stid", ""+student.getStid());
        param.put("password", student.getPassword());
        param.put("grate", student.getGrate());
        int result = helloService.updateStu(param);
        if (result == 0)
            return new Result(403, "更新失败。");
        return new Result(200, "更新成功。");
    }
}
```

注意：

1. 该控制器使用了@RestController 而不是@Controller，这样我们可以偷懒，从而轻松实现 REST，而不必为实现将对象转为 Json 数据而在每个 Mapping 上添加@ResponseBody。
2. Result 对象为了方便而编写的一个 POJO

```
public class Result {
    private int stateCode;
    private String message;

    public Result() {
        super();
    }

    public Result(int stateCode, String message) {
```

```
        super();
        this.stateCode = stateCode;
        this.message = message;
    }
    get/set...
    toString..
}
```

3.你会发现在控制器中适应了 PUT、DELETE 两个请求方法，用于更新和删除操作，对应 SpringMVC 来说，这是支持的。但是对于 HTML 的表单提交只支持 POST 和 GET 两种方式。（REST 的关键原则之一就是“使用标准接口”（the use of the Uniform Interface），也就是提倡根据不同的语义使用 GET, PUT, POST 和 DELETE 方法）。

那么该如何实现对 PUT 和 DELETE 的支持呢？（web.xml 的配置请往前翻）

在表单这边，需要使用隐藏域的方式，请详看页面代码

index.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <!-- 其中 get 和 post 方法是 html 中自带的，但是不支持 PUT 和 DELETE 方法，所以需要通过 POST 方法模拟这两种方法，只需要在表单中添加一个隐藏域，名为_method，值为 PUT 或 DELETE。 -->
    <form action="student/updateStu" method="POST">
    <input type="hidden" name="_method" value="PUT">
        <input type="hidden" name="stid" value="1" />
        密码：<input type="text" name="password" /> <span>
        班级：<input type="text" name="grate" />
            <input type="submit" value="修改" />
    </form>

    <form action="student/addStu" method="post">
    name:<input type="text" name="name" /> <br/>
   password:<input type="text" name="password" /> <br/>
      grate:<input type="text" name="grate" /> <br/>
     gender:<input type="text" name="gender"/> <br/>
         <input type="submit" value="添加" />
    </form>
```

```
        <form action="student/1" method="post">
            <input type="hidden" name="_method" value="DELETE"/>
            <input type="submit" value="删除" />
        </form>
    </body>
    </html>
```

注意代码段

```
    <input type="hidden" name="_method" value="PUT">
```

隐藏中 name 使用了"_method"，与 web.xml 所配置的对应过滤器 `param-value` 保持一致。

```
    <filter>
        <filter-name>HiddenHttpMethodFilter</filter-name>
        <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
        <init-param>
            <param-name>methodParam</param-name>
            <param-value>_method</param-value>
        </init-param>
    </filter>
```

10.Rest 功能测试

到此为止，代码部分完全结束了。我们来看看效果。这个结果还是很有必要给出的。
运行，部署到服务器之后可看到如下效果页
http://localhost:8080/SSMIntegration/

1.添加操作
填入对应值，如玛丽、123123、九年一班、0　　（这里 0 代表女）
添加成功，在页面中将会返回{"stateCode":200,"message":"添加成功。"}
在数据库中也将会多出一条添加的数据

id	name	password	gender	grate
1	玛丽	123123	0	九年一班

2.查询
http://localhost:8080/SSMIntegration/student/1
返回结果：
{"stid":1,"name":"玛丽","password":"123123","gender":0,"grate":"九年一班","score":[]}
3.修改
为了测试方便，在表单中，加 stid 设置了默认值 1，也就是只对 stid 为 1 的数据进行修改。
输入修改的数据如：asd、计算机科学与技术
{"stateCode":200,"message":"更新成功。"}

id	name	password	gender	grate
1	玛丽	asd	0	计算机科学

4.删除
同样为了操作方便，将 stid 设置成了 1，点击删除：
{"stateCode":200,"message":"删除操作成功"}
此查看数据库，会发现数据已经没有了。再次删除将会出现错误
{"stateCode":403,"message":"删除失败"}

11.结束

OK，基本上完成了。总算结束了，累死我了。。。
可以试着，模仿敲一敲。如有疑问请留言。
小弟的博客专栏：http://blog.csdn.net/IT_faquir/article/list/