如何用Junit进行并开展单元测试

author--黄永雄

1 背景

单元测试最初兴起于敏捷社区。1997年，设计模式四巨头之一Erich Gamma和极限编程发明人Kent Beck共同开发了JUnit，而JUnit框架在此之后又引领了xUnit家族的发展，深刻的影响着单元测试在各种编程语言中的普及。当前，单元测试也成了敏捷开发流行以来的现代软件开发中必不可少的工具之一。同时，越来越多的互联网行业推崇自动化测试的概念，作为自动化测试的重要组成部分，单元测试是一种经济合理的回归测试手段，在当前敏捷开发的迭代中非常流行和需要。

本技术文档将从将会集中的从单元测试的价值、单元测试的编写以及单元测试的实践等三个方面来进行讲解。

本文开发工具平台为：Idea 2018.2；
使用开发语言：java 8
Junit版本：Junit 5
所测试对象：车联网后端controller层和service层代码，版本为：2.4.1
所测试环境：http://192.168.1.107:6060
后端所用技术： SpringBoot、Spring Cloud、MyBatis、Hibernate Validator、Maven、Jackson、Jedis。

2 单元测试的价值

2.1 什么是单元测试

单元测试（Unit Test），是一段自动化的代码，用来调动被测试的方法或类，而后验证基于该方法或类的逻辑行为的一些假设。单元测试几乎总是用单元测试框架来写的。它写起来很顺手，运行起来不费时。它是全自动的、可信赖的、可读性强的和可维护的。

而单元测试框架是一个应用程序的半成品。框架提供了一个可复用的公共结构，无论是开发工程师还是测试工程师都可以在多个应用程序之间进行共享该结构，并且可以加以扩展以便满足它们的特定的要求。

2.2 单元测试的价值

无论是Ui测试自动化，接口测试自动化还是单元测试的自动化，它们都是非常有威力的魔法，但是如果使用不当也会浪费我们大量的时间，从而对项目造成巨大的不利影响。另一方面，如果没有恰当的编写和实现这些测试，在维护和调用这些测试上面，也会很容易的浪费很多时间，从而影响产品代码和整个项目。所以我理解的这些所有的自动化测试，优秀且出色的它们都应该具备这样的特点：

- 自动的、可重复的执行的测试，并且可移植，不受外部环境影响
- 开发人员甚至是测试人员比较容易实现编写的测试，且具备扩展性
- 一旦写好，将来任何时间都依旧可以用
- 团队的任何人都可运行的测试
- 一般情况下单击一个按钮就可以运行，且快速运行
- 测试报告自动生成，甚至可以以email形式发送报告
- 能引入git版本控制，甚至接入jenkins，docker容器，从而实现持续部署持续测试的目标
- 其它....

总结以上，我们可以得到一些基本的应该遵循的简单原则，它们能够让不好的单元测试远离我们的项目，这些原则定义了一个优秀的单元测试（包括接口自动化，Ui自动化）应该具备的品质，且通过分析以下原则，以此来更好的实现单元测试代码的编写。

自动化：这里的自动化包括了三个方面，测试数据生成的自动化，测试过程执行的自动化，测试结果验证的自动化。

测试数据生成的自动化：无论是谁只要输入相关测试参数，就能自动生成测试数据被单元测试代码引用，并且在测试完毕后，还能自动清除测试数据，还原测试场景，达到测试数据可复用；

测试过程执行的自动化：代码首先需要能够正确的被调用，并且所有的测试可以有选择的依次执行。在junit5，它可以帮助我们自动的运行我们指定的测试，当然也可以选择所有测试用例自动的执行。

测试结果验证的自动化：测试结果必须在测试的执行以后，让测试工具读取并展示出来。如果一个项目需要通过雇佣一个人来读取测试的输出，然后验证代码是否能够正常的工作，那么这是一种可能导致项目失败的做法。而且自动化回归的一个重要特征就是能够让测试工具/代码来检查自身是否通过了验证，我们不希望由人类来检验这些重复性的手工行为，而且这也是人类不擅长且容易出错的地方。

彻底的：好的单元测试应该尽可能是彻底的，它们测试了所有可能会出现问题的情况。一个极端是每行代码、代码可能每一个分支、每一个可能抛出的异常等等，都作为测试对象。另一个极端是仅仅测试最可能的情形——边界条件、残缺和畸形的数据等等，不管是哪个，单元测试框架都应具备这样的条件来对这些进行扩展从而能够做到尽可能彻底的测试。

可重复：每一个测试必须可以重复的，多次执行，并且结果只能有一个。这样说明，测试的目标只有一个，就是测试应该能够以任意的的顺序一次又一次的执行，并且产生相同的结果。意味着，测试不能依赖不受控制的任何外部因素。这个话题引出了Mock的概念，必要的时候，需要用Mock来隔离所有的外界因素，但本文档的单元测试都是真实测试，且会引用其他外部依赖，Mock这个不在此文档讨论范围。

独立的：测试应该是简洁而且精炼的，这意味着每个测试都应该有强的针对性，并且独立于其它测试和环境。有些时候，测试可能在同一时间点被多个开发人员运行。那么在编写测试的时候，确保我们一次只测试了一样东西。独立的，意味着我们可以在任何时间以任何顺序运行任何测试。每一个测试都应该是一个孤岛。

专业的：测试代码需要是专业的。意味着，在多次编写测试的时候，需要注意抽取相同的代码逻辑，进行封装设计，且测试代码是真实的代码。在必要的时候，需要创建一个框架进行测试。测试的代码应该和产品的代码量大体相当。所以测试代码需要保持专业，有良好的设计。

如果按照以上的原则进行单元测试框架搭建，以及单元测试代码编写，那么我们就有可能会实现以下的效果：

- 两周或者两个月、甚至半年前写的单元测试，现在可以运行并得到结果。
- 可以数分钟内写一个基本的单元测试，甚至是全新功能的单元测试。
- 可以简单修改下测试数据以及对应想要验证的结果，就可轻松运行并得到。
- 可以在数秒以内跑完某个页面的所有的单元测试，并得到测试结果。
- 可以在数分钟内跑完某个功能模块所有的单元测试，并得到测试结果。
- 可以在1个小时内跑完所有功能模块的单元测试，并得到测试结果。
- 可以通过单击一个按钮就能运行所写的单元测试，并得到测试结果。
- 可以不受环境影响，轻松移植到别的环境就可运行测试，并得到测试结果。

这样，单元测试的价值就能体现出来，它是任何人都可以运行的。在这个前提下，测试的运行能够足够快，运行起来不费力、不费事、不费时，并且即便写新的测试，也应该能够顺利、不耗时的完成。

3 单元测试的编写

首先我们先来引用后端的代码，以基础数据模块的个人车主信息为例，以此来介绍控制层的单元测试。
通过设置测试参数，实例化PO对象并转成VO对象，再由Junit来执行测试并校验测试结果，最后返回测试报告。
在此过程，实例化PO对象和VO对象将组件化，为自动生成。Junit的执行测试和结果检验也将组件化，为自动生成。
另外，由于要实现单元测试的自动的、独立的、彻底的特性，我们需要引用被测对象的服务类来新增测试数据，以此来作为查询、修改以及删除操作的依据，这样不可避免的牺牲了对外部不可依赖的特性。。但从另一角度来看，控制层本与服务层息息相关，如果服务层有问题，那么本文所编写的控制层的单元测试（真实测试）也必定会报错。

3.1 被测对象的代码

由于篇幅原因，被测对象代码不详尽列出，具体可进入后端自行查看。

OwnerPeople PO对象类

```
public class OwnerPeople extends TailBean {

    /** * 主键 *
     */
    private String id;
    /**姓名 *
     */
    private String ownerName;
    /**单位id *
     */
    private String unitId;
    /**单位名称
     */
    private String unitName;
    /**岗位 *
     */
      ....
      后面省略
      ....
}
```

2.3.2 OwnerPeopleModel VO对象类

```
public class OwnerPeopleModel extends BaseModel {

    @ApiModelProperty(value = "主键")
    private String id;

    @ColumnHeader(title = "联系人姓名", example = "汪涵", desc = "联系人姓名长度1-10个字符")
    @NotEmpty(message = "姓名不能为空", groups = {GroupInsert.class, GroupUpdate.class, GroupExcelImport.class})
    @Length(min = 1, max = 10, message = "姓名长度1-10个字符", groups = {GroupInsert.class, GroupUpdate.class,
GroupExcelImport.class})
    @ApiModelProperty(value = "姓名")
    private String ownerName;


      ....
      ....
      后面省略
      ....
      ....
    }
```

2.3.3 VehOwnerService 服务实现类

```
public class VehOwnerService extends BaseService implements IVehOwnerService {
```

```
    @Resource
    private DictMapper dictMapper;
    @Resource(name = "webRedisKit")
    private RedisKit  redisKit;

    @Override
    public Object list(PagerInfo pagerInfo) {
        // 获取当权限的map
        Map<String, Object> params = DataAccessKit.getAuthMap("sys_veh_owner", "vo");
        params.putAll(ServletUtil.pageInfoToMap(pagerInfo));
        if(params.containsKey("ids")) {
            String ids = (String) params.get("ids");
            params.put("ids", StringUtils.split(ids,","));
        }
        // 非分页查询
        if (pagerInfo.getLimit() == null ||  pagerInfo.getLimit()<0){
            List<VehOwner> entries = findBySqlId("pagerModel", params);
            List<VehOwnerModel> models = new ArrayList<>();
            for(VehOwner entry: entries){
                models.add(VehOwnerModel.fromEntry(entry));
            }
            return models;
        }
        // 分页查询
        else {
            PagerResult pr = findPagerModel("pagerModel", params, pagerInfo.getStart(), pagerInfo.getLimit());

            List<VehOwnerModel> models = new ArrayList<>();
            for(Object entry: pr.getData()){
                VehOwner obj = (VehOwner)entry;
                models.add(VehOwnerModel.fromEntry(obj));
            }
            pr.setData(Collections.singletonList(models));
            return pr;
        ....
        ....
        后面省略
        ....
        ....

}
```

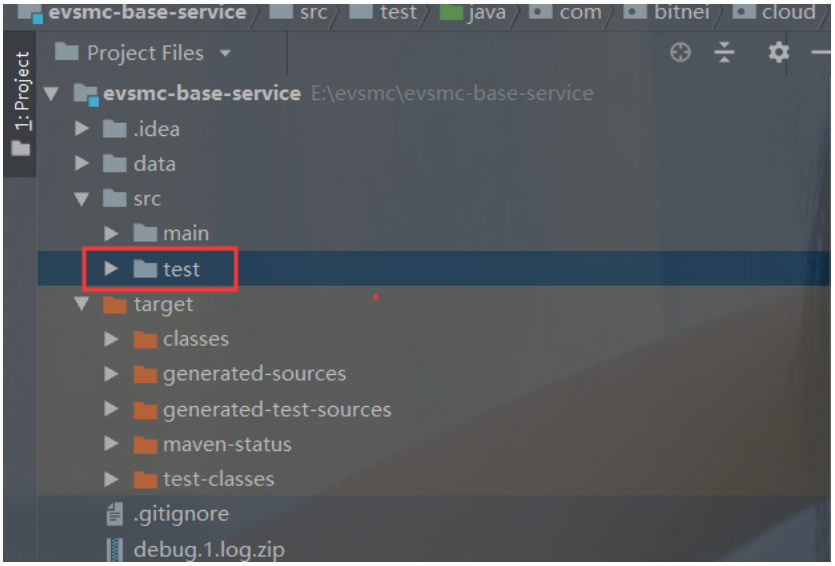
2.3.4 VehOwner Controller **控制类**

```
public class VehOwnerController {

    /* 模块基础请求前缀 *
     */
    public static final String BASE_AUTH_CODE = "VEHOWNER";
    /**查看 *
     */
    public static final String AUTH_DETAIL = BASE_AUTH_CODE + "_DETAIL";
```

```
    /** 列表 *
     */
    public static final String AUTH_LIST = BASE_AUTH_CODE + "_LIST";
    /**新增 *
     */
    @Autowired
    private IVehOwnerService vehOwnerService;

    /** 根据id获取对象
     */
    @ApiOperation(value = "详细信息", notes = "根据ID获取详细信息")
    @ApiImplicitParam(name = "id", value = "ID", required = true, dataType = "String", paramType = "path")
    @GetMapping(value = "/vehOwners/{id}")
    @ResponseBody
    @RequiresPermissions(AUTH_DETAIL)
    public ResultMsg get(@PathVariable String id) {
        VehOwnerModel vehOwner = vehOwnerService.get(id);
        return ResultMsg.getResult(vehOwner);
    }
    /**
     * 保存
     * @param demo1 VehOwnerModel
     */
    @ApiOperation(value = "新增", notes = "新增信息")
    @PostMapping(value = "/vehOwner")
    @ResponseBody
    @RequiresPermissions(AUTH_ADD)
    public ResultMsg add(@RequestBody @Validated({GroupInsert.class}) VehOwnerModel demo1, BindingResult
bindingResult) {
        if (bindingResult.hasErrors()) {
            return ResultMsg.getSchemaMsg(bindingResult);
        }
        vehOwnerService.insert(demo1);
        return ResultMsg.getResult(I18nUtil.t("common.addSuccess"));
    }
      ....
      ....
      后面省略
      ....
      ....
    }
}
```

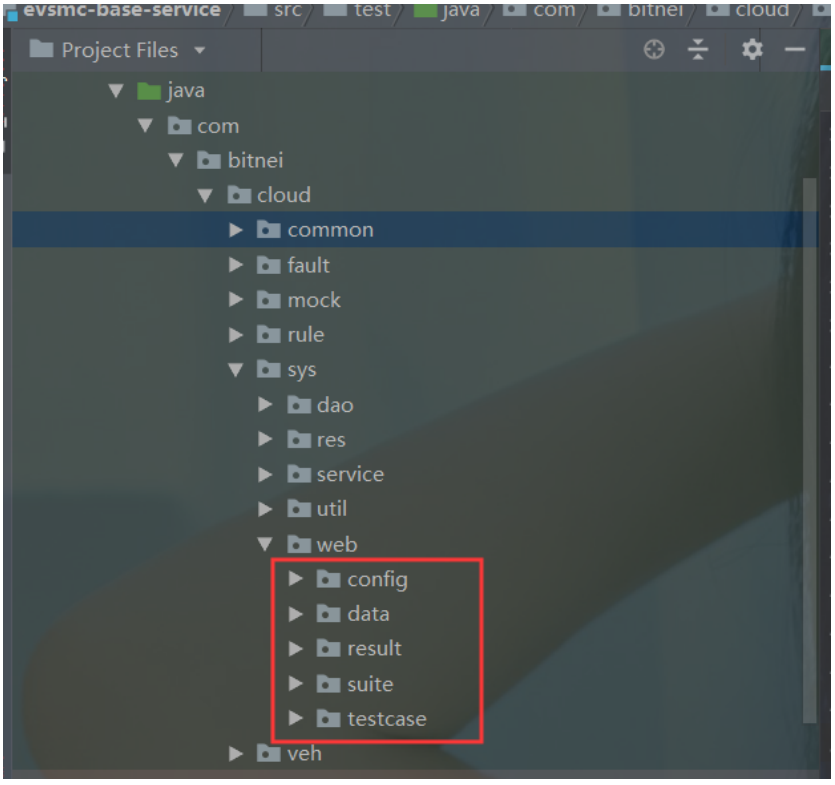
3.2 单元测试代码编写

3.21 后端主要是以SpringBoot、Spring Cloud实现，我们使用Idea引入后端代码，结构图如下：



其中main目录为项目实现的代码目录，test目录则为单元测试代码目录。

3.22 进入test/com/bitnei/cloud/sys/web，这是控制层的单元测试代码目录，在此目录依次建立config,data,result,suite,testcase这5个目录，如下图：



各目录作用：

config: 封装各控制层的url地址信息，组件化并调用；
data：在此目录可输入测试参数，封装po对象，组件化为测试数据并调用；
result: 封装测试执行数据以及测试结果验证，组件化并调用；
testcase：编写测试用例，引用各组件并执行测试。
suite：为测试用例套件，封装各测试用例来调用测试。

3.23 首先建BaseControllerTest类，封装控制层需要调用的请求头信息。

```
public class BaseControllerTest {
```

```
    /**
     * 获取header
     * @return
     */
    public HttpHeaders httpHeader(){
        HttpHeaders headers = new HttpHeaders();
        headers.add("accept","application/json");

headers.add("cookie","CSRF=YzI5MzBiYzA3MmZhNGQzMDk1MDA0NTUyZGQwOGZkMDU=;R_SESS=sJPPDUjCHahFygSVdZVGc
        headers.add("x-api-csrf","YzI5MzBiYzA3MmZhNGQzMDk1MDA0NTUyZGQwOGZkMDU=");
        return headers;
    }
}
```

3.24 在test/java/com/bitnei/cloud/sys/web/config目录，新建VehOwnerUrl类，封装url地址信息，并使用@Data注解即可被引用

```
@Data
public class VehOwnerUrl {

    String add_url ="/" + Version.VERSION_V1 + "/sys/vehOwner";
    String getAll_url = "/" + Version.VERSION_V1 + "/sys/vehOwners";
    String update_url = "/" + Version.VERSION_V1 + "/sys/vehOwners/{id}";
    String getDetail_url =  "/" + Version.VERSION_V1 + "/sys/vehOwners/{id}";
    String delete_url =  "/" + Version.VERSION_V1 + "/sys/vehOwners/{id}";
}
```

3.25在test/java/com/bitnei/cloud/sys/web/data目录，新建VehOwnerUtil类，生成输入参数，实例化po对象并转化为vo对象作为测试数据被调用

```
public class VehOwnerUtil {

    /**姓名 */
    private String ownerName=RandomValue.getChineseName();
    /**性别 */
    private Integer sex=1;
    /**手机号 */
    private String telPhone=RandomValue.getTel();
    /**联系地址 */
    private String address=RandomValue.getRoad();
    /**电子邮箱 */
    private String email=RandomValue.getEmail(5,10);
    /**证件类型 1:居民身份证 2:士官证 3 学生证 4 驾驶证 5 护照 6 港澳通行证 */
    private Integer cardType=1;
    /**证件号码 */
    private String cardNo= RandomValue.getCardNo();
    /**证件地址 */
    private String cardAddress=RandomValue.getRoad();

    private VehOwner vehOwner;
    private VehOwnerModel vehOwnerModel;
```

```
    public VehOwnerModel  createModel(){

        vehOwner = new VehOwner();
        vehOwner.setOwnerName(ownerName);
        vehOwner.setSex(sex);
        vehOwner.setTelPhone(telPhone);
        vehOwner.setAddress(address);
        vehOwner.setEmail(email);
        vehOwner.setCardType(cardType);
        vehOwner.setCardNo(cardNo);
        vehOwner.setFrontCardImgId(frontCardImgId);
        vehOwner.setBackCardImgId(backCardImgId);
        vehOwner.setFaceCardImgId(faceCardImgId);
        vehOwner.setCardAddress(cardAddress);
        vehOwnerModel = new VehOwnerModel();
        vehOwnerModel = vehOwnerModel.fromEntry(vehOwner);
        return vehOwnerModel;
    }
}
```

代码讲解：输入参数是由RandomValue组件生成，里面封装了各种参数的生成方法，开发工程师和测试工程师可根据自己的测试需要对测试数据进行生成各种model并引用，在边界值上、字段校验、错误数据，甚至是畸形数据都可轻松实现，并实例化成VO对象，供单元测试调用。

3.25在test/java/com/bitnei/cloud/sys/web/result目录，新建VehOwnerResult类，封装测试执行以及结果验证类。

```
public class VehOwnerResult extends BaseControllerTest {

    @Autowired
    private MockMvc mvc;

    public ResultActions getAddResult(String url,VehOwnerModel vehOwnerModel) throws Exception {

        ResultActions   actions = mvc.perform(
                MockMvcRequestBuilders.post(url).headers(httpHeader())
                        .contentType(MediaType.APPLICATION_JSON).content(new
ObjectMapper().writeValueAsString(vehOwnerModel)))
                .andExpect(status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.code").value(200))
                .andExpect(MockMvcResultMatchers.jsonPath("$.data").value("新增成功"))
                .andDo(print());
        return actions;
    }

    public ResultActions getAllResult(String url, PagerInfo pagerInfo) throws Exception {

        ResultActions  actions = mvc.perform(
                MockMvcRequestBuilders.post(url).headers(httpHeader())
                        .contentType(MediaType.APPLICATION_JSON).content(new ObjectMapper().writeValueAsString(pagerInfo)))
                .andExpect(status().isOk())
                .andExpect(MockMvcResultMatchers.jsonPath("$.code").value(200))
```

```
            .andExpect(MockMvcResultMatchers.jsonPath("$.pagination.total").isNotEmpty())
            .andDo(print());
        return actions;
    }

    public ResultActions getUpdateResult(String url, String id,VehOwnerModel vehOwnerModel) throws Exception {

        ResultActions   actions = mvc.perform(
            MockMvcRequestBuilders.put(url,id).headers(httpHeader())
                .contentType(MediaType.APPLICATION_JSON).content(new
ObjectMapper().writeValueAsString(vehOwnerModel)))
            .andExpect(status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$.code").value(200))
            .andExpect(MockMvcResultMatchers.jsonPath("$.data").value("更新成功"))
            .andDo(print());
        return actions;
    }

    public ResultActions getModelByNameResult(String url, PagerInfo pagerInfo) throws Exception {

        ResultActions  actions =mvc.perform(
            MockMvcRequestBuilders.post(url).headers(httpHeader())
                .contentType(MediaType.APPLICATION_JSON).content(new ObjectMapper().writeValueAsString(pagerInfo)))
            .andExpect(status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$.code").value(200))
            .andDo(print());
        return actions;
    }

    public ResultActions getDetailByIdResult(String url, String id) throws Exception {

        ResultActions  actions =mvc.perform(
            MockMvcRequestBuilders.get(url,id).headers(httpHeader())
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$.code").value(200))
            .andExpect(MockMvcResultMatchers.jsonPath("$.data.id").value(id))
            .andDo(print());
        return actions;
    }

    public ResultActions getDeleteByIdResult(String url, String id) throws Exception {

        ResultActions  actions =mvc.perform(
            MockMvcRequestBuilders.delete(url,id).headers(httpHeader())
                .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(MockMvcResultMatchers.jsonPath("$.code").value(200))
            .andExpect(MockMvcResultMatchers.jsonPath("$.data").value("删除成功，共删除了1条记录"))
            .andDo(print());
        return actions;
    }
```

}

代码讲解：此组件主要实现了增改查删的测试执行以及结果验证，在这里我们无需修改测试参数，只需要根据测试需求，引用model对象，并修改结果验证，即可被测试用例类调用来执行，并且这组件可以扩展成各种需求的验证，比如新增失败，更新失败，删除失败等场景来调用。

3.26 在test/java/com/bitnei/cloud/sys/web/testcase目录，新建VehOwnerControllerRealTest类，这个为测试用例类，引用测试数据，并调用各组件来执行测试。

```
public class VehOwnerControllerRealTest extends BaseControllerTest {

    @Resource
    IVehOwnerService iVehOwnerService;
    @Resource
    VehOwnerUtil vehOwnerUtil;
    @Resource
    PagerInfoUtil pagerInfoUtil;
    @Resource
    VehOwnerResult vehOwnerResult;

    private VehOwner vehOwner;
    private VehOwnerModel vehOwnerModel;
    private VehOwner u;
    private PagerInfo pagerInfo;
    private List<Condition> conditionList;
    private VehOwnerUrl vehOwnerUrl;


    @Before
    public void before(){
        //0.先实例url地址对象
        vehOwnerUrl=new VehOwnerUrl();
        //1.实例model并生成测试数据
        vehOwnerModel = vehOwnerUtil.createModel();
        //2.实例pagerInfo对象并赋值
        pagerInfo=pagerInfoUtil.createPagerInfo();
    }

    @Test
    public void insertVehOwner() throws Exception {
        //1.执行测试并验证结果
        ResultActions actions = vehOwnerResult.getAddResult(vehOwnerUrl.getAdd_url(),vehOwnerModel);
        //2.把新增的数据查询出id 再进行删除 还原数据
        vehOwner=new VehOwner();
        vehOwner=iVehOwnerService.findByOwnerName(vehOwnerModel.getOwnerName());
        String id=vehOwner.getId();
        iVehOwnerService.deleteMulti(id);
    }

    @Test
    public void getAllVehOwner() throws Exception {
        //1.执行测试并验证结果
        ResultActions actions=vehOwnerResult.getAllResult(vehOwnerUrl.getGetAll_url(),pagerInfo);
```

```
}

@Test
public void updateVehOwner() throws Exception {
    //1.先进行新增数据
    iVehOwnerService.insert(vehOwnerModel);
    //2.再实例化是一个Vo对象进行修改
    u=new VehOwner();
    //3.先查询需要修改的Vo对象数据
    u=iVehOwnerService.findByOwnerName(vehOwnerModel.getOwnerName());
    //4.取得所要修改的对象的id
    String id=u.getId();
    //5.将所要修改的Vo对象 重新编辑用户名
    u.setOwnerName(RandomValue.getChineseName()+1);
    //6.取得id并设置值
    u.setId(id);
    vehOwnerModel = vehOwnerModel.fromEntry(u);
    //7.执行测试并验证结果
    ResultActions actions=vehOwnerResult.getUpdateResult(vehOwnerUrl.getUpdate_url(),id,vehOwnerModel);
    //8. 进行删除新增的数据 还原数据
    iVehOwnerService.deleteMulti(id);
}

@Test
public void getVehOwnerByOwnerName() throws Exception {

    //1.先进行新增数据
    iVehOwnerService.insert(vehOwnerModel);
    //2.取得新增成功的对象
    vehOwner=iVehOwnerService.findByOwnerName(vehOwnerModel.getOwnerName());
    //3.设置查询对象
    conditionList=new ArrayList<>();
    Condition c1=new Condition();
    c1.setName("ownerName");
    c1.setValue(vehOwner.getOwnerName());
    conditionList.add(0,c1);
    pagerInfo.setConditions(conditionList);
    //4.执行测试并验证结果
    ResultActions actions=vehOwnerResult.getModelByNameResult(vehOwnerUrl.getGetAll_url(),pagerInfo);
    //5.进行删除新增的数据 还原数据
    String id=vehOwner.getId();
    iVehOwnerService.deleteMulti(id);
}

@Test
public void getVehOwnerDetailById() throws Exception {

    //1.先进行新增数据
    iVehOwnerService.insert(vehOwnerModel);
    //2.取得新增成功的对象
    vehOwner=iVehOwnerService.findByOwnerName(vehOwnerModel.getOwnerName());
    String id=vehOwner.getId();
```

```
        //3.执行测试并验证结果
        ResultActions actions=vehOwnerResult.getDetailByIdResult(vehOwnerUrl.getGetDetail_url(),id);
        //4.进行删除新增的数据 还原数据
        iVehOwnerService.deleteMulti(id);
    }

    @Test
    public void deleteVehOwner() throws Exception {

        //1.先进行新增数据
        iVehOwnerService.insert(vehOwnerModel);
        //2.取得新增成功的对象的id进行删除
        vehOwner=iVehOwnerService.findByOwnerName(vehOwnerModel.getOwnerName());
        String id=vehOwner.getId();
        //3.执行测试并验证结果
        ResultActions actions=vehOwnerResult.getDeleteByIdResult(vehOwnerUrl.getDelete_url(),id);

    }
}
```

代码讲解：此测试用例类为个人车主页面的增更查删的成功用例。

在这个类里，我们需要先执行before方法来先引用url信息，model信息以及pagerinfo信息等通用信息给测试用例调用。

测试用例实现的理念则是通过新增一条测试数据作为测试使用，比如修改，查询和删除，就需要新增1条测试数据被其调用来执行验证，验证完毕后，则需要调用服务来删除掉新增数据。反而新增的测试用例则不需要，直接执行测试并验证方法。

如果before（）方法里有引用了其他页面服务来新增别的测试数据，则就需要增加after（）方法，再引用服务来清楚掉新增的数据。

总而言之，测试用例类无非就是引入url信息（config层），再调用测试数据组件（data层），再通过结果类组件（Result层），最后通过其他服务，从而实现了单元测试用例的编写。

3.27 在test/java/com/bitnei/cloud/sys/web/suite目录，新建TestSysController类，这个为测试用例的套件类，方便把Sys(基础数据)这个模块所有的测试用例进行统一管理和执行。

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
                VehOwnerControllerRealTest.class,           //个人车主信息
                UnitControllerRealTest.class,               //单位信息管理
                OwnerPeopleControllerRealTest.class,        //单位联系人
                GroupControllerRealTest.class,              //数据权限组管理
                RoleControllerRealTest.class,               //角色管理
                UserControllerRealTest.class,               //账号管理
                FuelSystemModelControllerRealTest.class,    //燃料电池系统型号
                FuelGeneratorModelControllerRealTest.class, //燃油发电机型号
                PowerDeviceControllerRealTest.class,        //发电装置信息
                EngineModelControllerRealTest.class,        //发动机型号
                DriveMotorModelControllerRealTest.class,    //驱动装置型号
                DriveDeviceControllerRealTest.class,        //驱动装置信息
                SuperCapacitorModelControllerRealTest.class,//超级电容型号
                BatteryDeviceModelControllerRealTest.class, //动力蓄电池型号
                EngeryDeviceControllerRealTest.class,       //可充电储能装置信息
                SimManagementRealTest.class,                //SIM卡管理
                TermModelControllerRealTest.class,          //车载终端型号
                TermModelUnitControllerRealTest.class,      //车载终端信息
                VehTypeControllerRealTest.class,            //车辆种类信息
```

```
        VehBrandControllerRealTest.class,          //车辆品牌
        VehSeriesControllerRealTest.class,         //车辆系列
        VehNoticeControllerRealTest.class,         //车辆公告
        VehModelControllerRealTest.class           //车辆型号
        ...
        ...
    })
public class TestSysController {
}
```

代码讲解：某个功能模块的测试单元类应为1个套件，整理成1个类，可点击一个按钮全部执行。

3.28 在test/java/com/bitnei/cloud/目录，新建TestAllr类，这个也为测试用例的套件类，方便把所有功能模块的测试用例进行统一管理和执行。

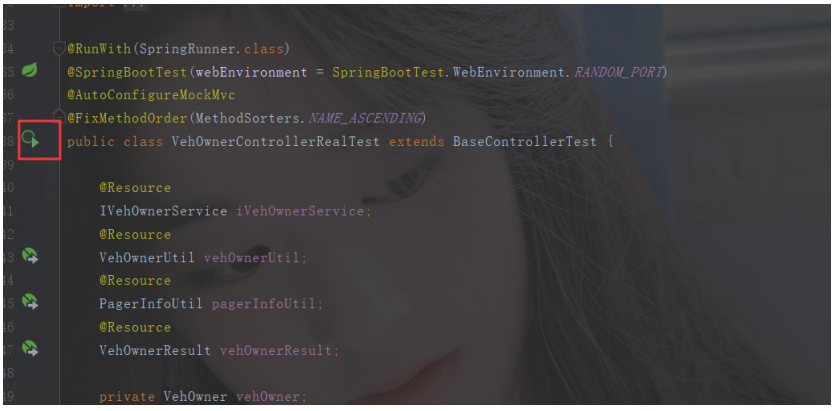
```
@RunWith(Suite.class)
@Suite.SuiteClasses({
        TestSysController.class，//基础数据
        TestVehContorller.class，//统计分析
        TestScreenContorller.class  //大屏监控
        ...
        ...
})
public class TestAll {
}
```

代码讲解：全部功能模块的测试单元类应为1个套件，整理成1个类，可点击一个按钮全部执行。

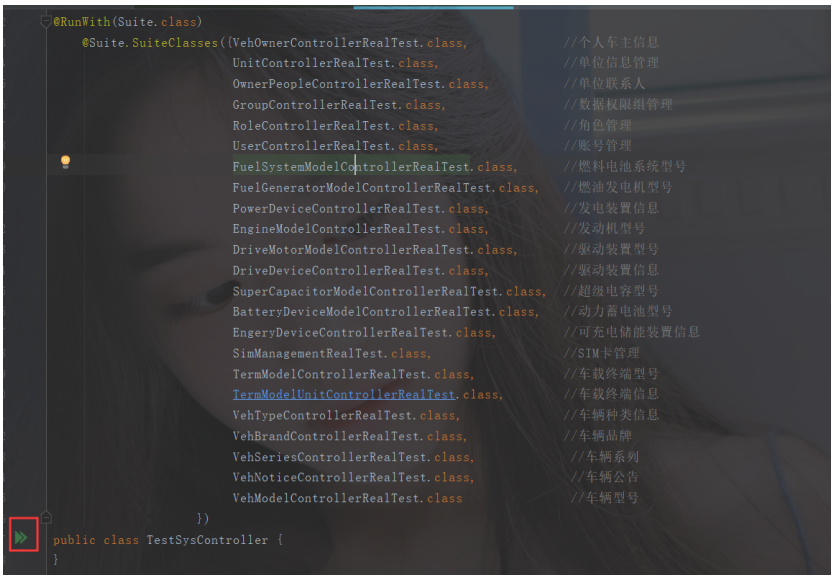
3.29 执行测试用例的全部方法

1）测试类的某个测试用例的执行，点击下图的左上角绿色执行按钮，即可执行执行的测试用例，绿色表示已经执行并测试通过；

2）测试类全部测试用例的执行，点击下图的左上角绿色执行按钮，即可执行该类的全部测试用例，绿色表示已经执行并全部测试通过；



3）某个功能模块的全部测试类的执行，点击下图的左上角绿色执行按钮，即可执行该功能模块的全部测试用例，绿色表示已经执行并全部测试通过；



4）全部功能模块的全部测试类的执行，点击下图的左上角绿色执行按钮，即可执行全部功能的测试用例，绿色表示已经执行并全部测试通过；

3.30 查看单元测试报告，为HTML网页形式展现

图1：表示跑完了基础数据模块的124个测试用例，124个通过，100%通过率，用时2分24秒；

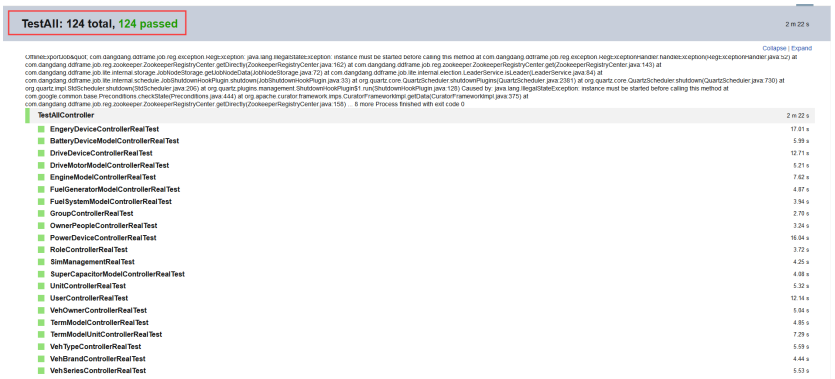


图2：表示可以对各测试用例点击展开并查看其请求信息以及响应信息，以下是新增的测试用例

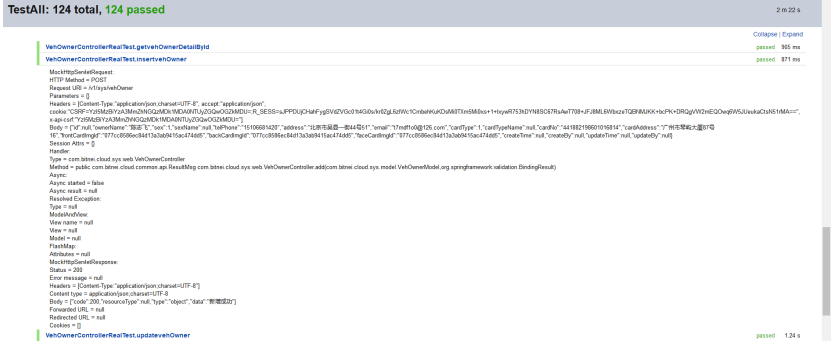


图3：以下是修改的测试用例的请求信息以及响应信息

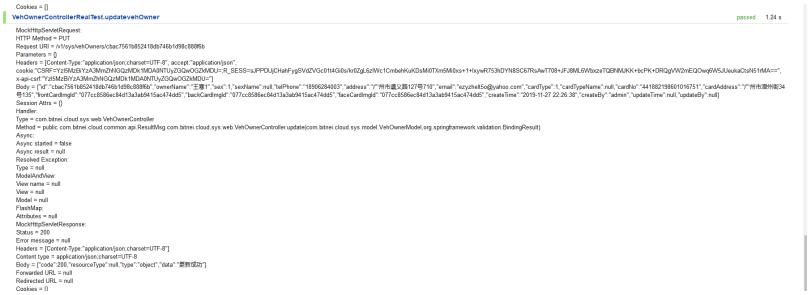
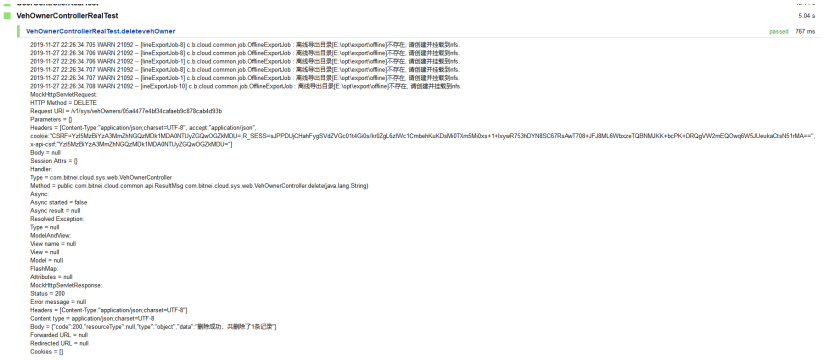


图4：以下是删除的测试用例的请求信息以及响应信息



4 总结

4.1 我为何要先做controller层的单元测试

Dao层就是持久层，负责与数据库打交道，具体到某个表，某个实体的增删改查。

Sservice层就是服务层，封装Dao层的操作，使一个方法对外表现为实现一种功能，举个例子，在进行新增车辆的时候，首先还是要确定这个用户是否有权限，有权限，还得查询用户新增的vin码，数据库是否已存在等等。

Controller层也就是业务控制层，负责接收数据和请求，并且调用Service层实现这个业务逻辑。

Controller层像是一个服务员，他把客人（前端）点的菜（请求数据、请求的类型等）进行汇总比如什么菜，口味、咸淡、量的多少，交给厨师长（Service层），厨师长则告诉沾板厨师（Dao 1）、汤料房（Dao 2）、配菜厨师（Dao 3）等（这些都是Dao层）我需要什么样的半成品，副厨们（Dao层）就负责完成厨师长（Service）交代的任务，最后制造完成返回给客人（前端），也就是最终展现给的用户。

这就是我们想要做的，先从业务的场景来驱动测试，无论用户进行的是成功的操作，还是失败的操作，只要能返回我们想要的结果，那就是最快捷的回归测试，当然这一块是很讲究测试的覆盖率。

再往深一些，就要做用户不能做的事情，比如抛出异常，畸形数据，每一个代码以及代码的分支，那就需要更深入的进入server层，dao层进行测试。

4.2 controller层单元测试的好处

测试人员可以在研发阶段就进行测试，提高迭代更新的效率。
测试人员接近了代码实现的逻辑，有助于在部署前发现问题。
测试人员和开发人员都共用单元测试代码，有助于对bug的定位和对业务的理解。
相比接口测试，controller层的单元测试更能在网络不稳的情况下快速的进行测试。
相比接口测试，controller层的单元测试具备更好的维护条件以及纠错环境。
相比接口测试，contoller层的单元测试 。

4.3 一些问题

有些时候，我也在担心这些单元测试并没有有效的改善生产力，甚至单元测试有时候变成一种负担。
比如我们想要实现真实测试，而不是mock的模拟测试，那不得不依赖别的服务。

比如我们想要实现测试用例的简洁以及可阅读性，可能会过度的封装，导致执行效率变慢。

比如我们想要追求覆盖率，但又因为自身能力，无法提升代码本身的质量。

....

....

这些都是有可能成为的拦路石，问题会一直在，解决了一个还会有另一个，但这些不能让我们因此而放弃。做了比什么都不做要好。只要目标在，那么我们就能一步一步的接近并实现。