UT的本质问题是什么

UT的本质是开发的一部分，是保证开发人员写出来的代码是按既定设计逻辑实现的，是为了让开发人员正面检测代码执行的正确性和逻辑结构的周密性，以确定实际运行状态与预期一致；保证后期修改或重构不影响原来的正确性；提高代码扩展和维护的信心。

        所以，UT不是为了发现问题，也不能用前后端发现问题比来度量。如果按照TDD的方法来做，那这个前端发现问题可以无限的放大，这个比想让它弄成多少，就可以弄成多少。

**UT的价值**

        之前很多时候被挑战，让讲清楚UT的价值，这个确实没法讲，因为无法度量。像通常说的问题发现前移比例增加了XX%，实际也是不靠谱的，因为上面讲的，这个比例你想看多少，我就可以做成多少http://3ms.huawei.com/hi/public/js/ewebeditor/skin/hi/emoticons/small/13.gif。 当然问题发现前移，或者说低级问题不流到后端，这个说法是正确的，只是无法度量。

        另外UT的另一个重要价值就是后面的维护省心，一劳永逸。（有人可能会说自己的代码UT很多，覆盖率很高，但是维护成本也非常好，经常加一个需求，功能代码半天改完了，但UT可能需要一天甚至更长的时间来改。那只能说是UT做残了。）

        UT不可能防护100%的问题，因为本身的软件实现逻辑就不可能100%考虑清楚，而且事物也是在不断变化发展的。但是没关系，出了问题，针对问题增加相应的测试防护即可，那么又是后面一劳永逸的事情了。

        UT过程，实际是从另一个角度验证自己的设计，验证自己代码的program smart性。比如，一个大函数、上帝类，测试起来会觉得很别扭，要写很多用例时，可能会自然想到是不是该拆小点了。

        UT提供WIKI功能，看代码功能和入门，UT是个非常便捷的途径。

**UT与软件可信的最高境界**

        我认为软件可信的最高境界是推脱甩手三联：**我不知道，我啥都没改，这事跟我没关系**。 其实仔细想想软件的高内聚，低耦合，SOLID原则，无不是就为了达成这甩手三联而设。

        我们多少开发同学为周边依赖所累，无休止的打扰，实难静下心来搞自己的事情，但实际上被找的问题，可能大多数是非问题。虽然我们很多时候埋怨对方自己不多看一步，老是鸡毛蒜皮小事都找来，但是，行有不得，还是得反求诸己，还是自己在边界处理上没有做到十足的自信。如果我们UT（LLT）做的充分，完全可以拍胸脯来下推脱三联。而且这样下来会形成一定口碑效应，就像我们通常不先怀疑业界成熟软件一样。

        关于UT的问题和做不好的原因，我觉得高博这篇<http://3ms.huawei.com/km/blogs/details/7618441>文章写的已经非常详细清楚了，我再简单啰嗦几个观点：

        1 UT好不好做，做的好不好，都跟软件架构有非常强的关系。软件的本质问题、UT的本质问题还是软件架构问题。

        2 如果是一堵烂泥墙，就不要勉强了，就把整个墙当UNIT吧，先把整个墙防护起来，再慢慢搞里面的砖块块。

PS: 因为我认为UT是成本最低的LLT，所以这里很多观点是普适LLT的，也都用UT代替了。

如何做好UT？ 且听下回分解。

二

**JAVA UT实践分享 - UT还是APITEST？UT如何衡量、做到什么程度？**

  日期：2020-11-07 16:10浏览：200[评论：0](http://3ms.huawei.com/km/blogs/details/9294383?l=zh-cn#comment_pop)

这个话题实在如老太婆裹脚布，臭又长。

但却总是被问起，这里就再写下个人观点，留作砖扔。

这个问题常被问起，根源还是我们有太多烂泥墙样的软件代码，无从下手。因此，UT还是API TEST，关键就是看自己代码的块块（稳定的领域）边界在什么地方，从边界对这个块块进行整体防护。 UT还是API TEST，只是技术手段。其实UT框架也可以做API TEST，而且成本更低。 API TEST对异常数据构造不友好。

ManageOne对UT范围的建议：

* ​ (1)核心算法需要编写UT，包括数据转换等
* ​ (2)对业务依赖较小的方法（类似于kafka生产消费消息等），需要编写UT
* ​ (3)内部模块提供的接口（Interface）需要编写UT
* ​ (4)简单的逻辑不用覆盖UT(比如异常里只打印日志，不需要覆盖此异常)
* ​ (5)建议：Rest接口编写UT，构筑测试金字塔型防护网

个人建议用UT框架做到集成测试。怎么做集成测试，后面例子讲解。

关于UT如何衡量，做到什么程度，这个事情就一个标准：投入产出比，用最少的用例，做到完整的防护。只要觉得能够防护住了逻辑，那就多一个用例都不要写了。

投入：

    （1）开发UT的成本，前期投入还是蛮大的，有时UT代码量不比业务代码少。

    （2）维护UT的成本，UT如果没做到独立正交，又变更频繁，那么维护成本就会非常高。业务代码改半天就完成，而UT要同步改一周的情况也曾听说过。另外维护成本还有一个比较坑的点是MOCK桩管理，如果管理不善，可能会像乱麻一样，一个问题解半天才找到桩是哪个，

产出：

    （1）交付质量，能够按设计既定逻辑交付。

    （2）后期交付效率提升，能够快速验证该代码逻辑。

    （3）后期重构变更保障，有了UT防护，心里踏实多了。

比如，一个数据类，测试new一下，判非空。 这个就简直了，而且本身测SERVICE层时其实可以顺带把它也给测了。实属不必要。

以上都是正确的废话，写UT痛苦时，根本感受不到后面效率提升、重构保障这些滞后的反馈，没有及时反馈，基本都死在了起步阶段。有想这个事情的时间，不及动手写个用例来的实惠。

那么到底怎么样的UT算好的，怎么样算防护住了，怎么样才能做好UT呢？且听下回

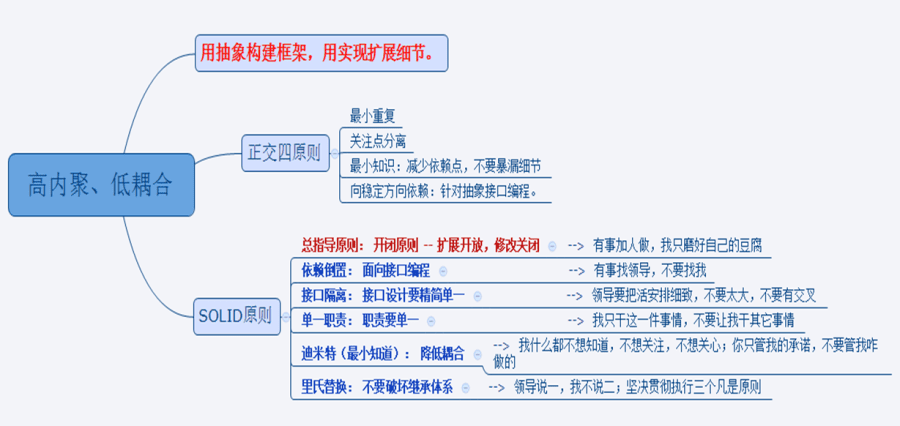
### JAVA UT实践分享 - 如何做好UT设计？

  日期：2020-11-17 16:22浏览：108[评论：0](http://3ms.huawei.com/km/blogs/details/9342447?l=zh-cn#comment_pop)

这个话题，依然是个空大上，臭又长的话题，不过我打算换个角度，举个例子简单讲下。

前文有说到，软件可信的最高境界是推脱甩手三联：我不知道，我啥都没改，这事跟我没关系。 不管软件设计，还是测试设计都是要围绕这个宗旨开展。

何以说这甩手三联 == SOLID原则呢？ 我们来看下高内聚、低耦合，还有SOLID原则的通俗理解：



所以说，软件设计、测试设计归纳总结就一个原则：独立正交。

独立： 把一个软件拆成几个相互独立的模块、微服务，最好这几个东西老死不相往来。

正交： 这几个独立的东西，拼起来，就又是这个完整的软件，没有漏东西。

那么，到哪里能买到这么好的东西呢？

其实呢，所有的设计，都是多思考一点。（我问柱子哥为什么这么牛，怎么做到的。 他说他可能就是他比别人认真一点。）

具体执行层面当然就是合理的拆分，在各个拆分后的领域内独立干活，完成自己的生命周期。最后交给上层集成。

下面举例说明

|  |  |
| --- | --- |
| 1 | public JSONObject queryPerfDataSetFromEs(String dataSet, PerfQueryCondition perfQueryCondition, int pageNo, |
| 2 | int pageSize) throws ServiceException { |
| 3 | TimeRangeCondition timeRangeCondition = perfQueryCondition.getTimeRange(); |
| 4 | List<MetricMeta> metricList = perfQueryCondition.getMetrics(); |
| 5 | if (timeRangeCondition == null || metricList == null) { |
| 6 | LOG.error("paramter 'timeRange' or 'metric' is null"); |
| 7 | throw ExceptionUtil.getServiceException("paramter 'timeRange' or 'metric' is null", |
| 8 | HttpStatus.BAD\_REQUEST, "paramter illegal", "", ""); |
| 9 | } |
| 10 |  |
| 11 | List<String> attributeList = perfQueryCondition.getAttributes(); |
| 12 | if (CollectionUtils.isEmpty(attributeList)) { |
| 13 | LOG.error("no 'attributes' in condition"); |
| 14 | throw ExceptionUtil.getServiceException("paramter 'attributes' is null or empty", |
| 15 | HttpStatus.BAD\_REQUEST, "paramter illegal", "", ""); |
| 16 | } |
| 17 |  |
| 18 | // 解析options参数 |
| 19 | OptionInfo optionInfo = perfQueryCondition.getOptions(); |
| 20 | if (optionInfo == null) { |
| 21 | LOG.warn("no 'options' in condition"); |
| 22 | throw ExceptionUtil.getServiceException("paramter 'options' is null or empty", HttpStatus.BAD\_REQUEST, |
| 23 | "paramter illegal", "", ""); |
| 24 | } |
| 25 |  |
| 26 | int topnSize = 0; |
| 27 | boolean topnAscending = true; |
| 28 | MetricMeta topnSortMetric = null; |
| 29 | String interval = null; |
| 30 | if (optionInfo.getTopN() != null) { |
| 31 | topnSize = optionInfo.getTopN().getSize(); |
| 32 | topnAscending = optionInfo.getTopN().getAscending(); |
| 33 | topnSortMetric = optionInfo.getTopN().getSortBy(); |
| 34 | } |
| 35 | if (optionInfo.getTrend() != null) { |
| 36 | interval = optionInfo.getTrend().getGranularity(); |
| 37 | } |
| 38 |  |
| 39 | // 趋势和topn只能选其一，以趋势选项优先 |
| 40 | if (interval != null) { |
| 41 | topnSize = 0; |
| 42 | topnSortMetric = null; |
| 43 | } |
| 44 |  |
| 45 | // 取topn时，对pageNo做校验 |
| 46 | if (topnSize > 0 && pageSize \* pageNo >= topnSize && pageNo > (topnSize % pageSize == 0 |
| 47 | ? topnSize / pageSize |
| 48 | : topnSize / pageSize + 1)) { |
| 49 | LOG.error("paramter 'topnSize':{},'pageSize':{},'pageNo':{} is illegal", topnSize, pageSize, pageNo); |
| 50 | throw ExceptionUtil.getServiceException("paramter 'topnSize','pageSize','pageNo' is illegal", |
| 51 | HttpStatus.BAD\_REQUEST, "paramter illegal", "", ""); |
| 52 | } |
| 53 |  |
| 54 | // 获取dimensionMetricRuleNameMap、attributeRuleNameMap和refDataSet |
| 55 | // dimensionMetricRuleNameMap用于保存指标和维度规则名称与完整域名称的关系 |
| 56 | // attributeRuleNameMap用于保存属性规则名称与完整域名称的关系 |
| 57 | Triple<Map<String, String>, Map<String, String>, String> triple = getRuleNameMap(dataSet); |
| 58 | Map<String, String> dimensionMetricRuleNameMap = triple.getLeft(); |
| 59 | Map<String, String> attributeRuleNameMap = triple.getMiddle(); |
| 60 | String refDataSet = triple.getRight(); |
| 61 | LOG.warn("dimensionMetricRuleNameMap:{}", dimensionMetricRuleNameMap); |
| 62 | LOG.warn("attributeRuleNameMap:{}", attributeRuleNameMap); |
| 63 | LOG.warn("refDataSet:{}", refDataSet); |
| 64 |  |
| 65 | // 解析attributes参数 |
| 66 | List<String> needDisplayList = attributeList; |
| 67 | // attributes保存需要补全的维度，dimensions保存数据集定义的维度 |
| 68 | List<String> attributes = new ArrayList<String>(); |
| 69 | List<String> dimensions = new ArrayList<String>(); |
| 70 | parseDisplayList(dataSet, dimensionMetricRuleNameMap, attributeRuleNameMap, needDisplayList, attributes, |
| 71 | dimensions); |
| 72 |  |
| 73 | // 解析filters参数 |
| 74 | AttributeFilterCondition filterCondition = perfQueryCondition.getFilters(); |
| 75 | if (filterCondition == null) { |
| 76 | filterCondition = new AttributeFilterCondition(); |
| 77 | } |
| 78 | extractFilterCondition(dataSet, attributeRuleNameMap, filterCondition); |
| 79 |  |
| 80 | // 解析metrics参数 |
| 81 | if (topnSortMetric != null) { |
| 82 | metricList.add(topnSortMetric); |
| 83 | } |
| 84 | parseMetricList(dataSet, metricList, dimensionMetricRuleNameMap, needDisplayList); |
| 85 |  |
| 86 | // 拼接查询body体 |
| 87 | JSONObject conditionJsonObject = assembleConditionJson(timeRangeCondition, interval, dimensionMetricRuleNameMap, |
| 88 | needDisplayList, dimensions, metricList, dataSet, filterCondition); |
| 89 | // 拼接属性补全addtions参数 |
| 90 | putAdditionsToConditionJson(attributeRuleNameMap, refDataSet, attributes, conditionJsonObject, filterCondition); |
| 91 |  |
| 92 | // 拼接查询参数 |
| 93 | JSONObject flatQueryParasJSONJsonObject = assembleQueryParam(topnAscending, topnSortMetric, interval, |
| 94 | dimensions); |
| 95 |  |
| 96 | // 调用es接口并返回 |
| 97 | String url = ES\_PERF\_QUERY\_URL.replace("{DATASET}", dataSet) |
| 98 | .replace("{PAGENO}", pageNo + "") |
| 99 | .replace("{PAGESIZE}", pageSize + "") |
| 100 | .replace("{FLATQUERYPARAS}", flatQueryParasJSONJsonObject.toJSONString()); |
| 101 | LOG.warn("start to invoke url:{}, conditionJsonObject:{}", url, conditionJsonObject); |
| 102 | RestfulResponse response = RestUtil.invokeURLWithRetry(RequestMethod.POST, url, conditionJsonObject); |
| 103 | if (response == null) { |
| 104 | LOG.error("failed to invoke url:{}, conditionJsonObject:{}", url, conditionJsonObject); |
| 105 | return null; |
| 106 | } |
| 107 | LOG.warn("es response:{}", response.getResponseContent()); |
| 108 | // 根据needDisplayList筛选 |
| 109 | JSONObject esResultJson = JSONObject.parseObject(response.getResponseContent()); |
| 110 | JSONArray dataJsonArray = esResultJson.getJSONArray("datas"); |
| 111 | JSONArray newDataJsonArray = filterEsResult(needDisplayList, dataJsonArray); |
| 112 |  |
| 113 | // 如果不需要排序，直接返回 |
| 114 | if (topnSize == 0) { |
| 115 | return esResultJson; |
| 116 | } |
| 117 | // 如果需要排序，根据topnSize缩减 |
| 118 | int totalSize = esResultJson.getIntValue("totalSize"); |
| 119 | if (topnSize >= newDataJsonArray.size()) { |
| 120 | return esResultJson; |
| 121 | } else { |
| 122 | JSONArray topnDataJsonArray = JSONArray.parseArray( |
| 123 | newDataJsonArray.subList(0, topnSize - pageSize \* (pageNo - 1)).toString()); |
| 124 | JSONObject topnEsResultJson = esResultJson; |
| 125 | topnEsResultJson.put("datas", topnDataJsonArray); |
| 126 | return topnEsResultJson; |
| 127 | } |
| 128 | } |

以上函数很多，很啰嗦，但通过注释搭眼看就是个三段式：构造入参，执行请求，整理返回。

所以，上面这段函数应该按照理解切成三个上下文：参数构造， ES查询执行，返回值构造。

根据上下文，分别构筑测试防护：

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void should\_successfully\_generate\_input\_when\_construct\_invoke\_parameter() { |
| 3 |  |
| 4 | } |
| 5 |  |
| 6 | @Test |
| 7 | public void should\_successfully\_get\_dataset\_when\_invoke\_elasticsearch() { |
| 8 |  |
| 9 | } |
| 10 |  |
| 11 | @Test |
| 12 | public void should\_successfully\_get\_return\_info\_when\_construct\_return\_data() { |
| 13 |  |
| 14 | } |

当然，以上只是基础的正常功能防护。可以继续增加边界值、异常输入，甚至用等价类、判定表等测试设计方法设计详细全面的用例，进行防护。

PS：

1 测试防护可以不一下子做的特别全，因为这样容易工作量太大导致丧失信心和动力。 可以重构一点加一点。

2 最好在外面再加一个拉通测试如下，就perfect了（有没有点金字塔的感觉？）。

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void should\_successfully\_get\_return\_info\_when\_query\_by\_xx\_parameter() { |
| 3 |  |
| 4 | } |

有了这一步的测试防护，可以将上面的大函数按上下文拆成三个小函数。（当然用IDEA直接抽函数有点问题，需要处理下）

然后再看下周边，有类似的拼入参、整出参的，可以继续抽象公共代码，框架也就会出来了。

这样代码就越来越美观优雅了，测试防护也越来越有效了，整体软件也越来越独立正交了，软件交付也越来越能做到甩手三联了，人也越来越有成就感了。

以上事虽小，但足以以小见大。再大的软件也是一层层分拆下来，就考验架构师的分拆能力了（我理解这也是我司当前软件TOP1问题所在）。

PS：

以上涉及DDD、重构、软件设计模式等理论，建议学一下。

### JAVA UT实践分享 - JAVA UT框架推荐

  日期：2020-11-17 16:43浏览：102[评论：2](http://3ms.huawei.com/km/blogs/details/9342695?l=zh-cn#comment_pop)

关于UT开发框架，我觉得这个链接就够了：[https://code.huawei.com/mo-architecture/mo-flag/blob/master/ut/Java UT Standard.md](https://code.huawei.com/mo-architecture/mo-flag/blob/master/ut/Java%20UT%20Standard.md)

冯勇整理的确实比较全了。当然JUNIT 5等一些新框架也欢迎使用的同学分享经验。

关于UT度量框架，建议使用JACOCO，配置参照：

|  |  |
| --- | --- |
| 1 | <build> |
| 2 | <plugins> |
| 3 | <plugin> |
| 4 | <groupId>org.jacoco</groupId> |
| 5 | <artifactId>jacoco-maven-plugin</artifactId> |
| 6 | <version>${jacoco.version}</version> |
| 7 | <executions> |
| 8 | <execution> |
| 9 | <id>pre-test</id> |
| 10 | <goals> |
| 11 | <goal>prepare-agent</goal> |
| 12 | </goals> |
| 13 | <configuration> |
| 14 | <propertyName>jacocoArgLine</propertyName> |
| 15 | </configuration> |
| 16 | </execution> |
| 17 | <execution> |
| 18 | <id>post-test</id> |
| 19 | <phase>test</phase> |
| 20 | <goals> |
| 21 | <goal>report</goal> |
| 22 | </goals> |
| 23 | </execution> |
| 24 | </executions> |
| 25 | </plugin> |
| 26 | <plugin> |
| 27 | <groupId>org.apache.maven.plugins</groupId> |
| 28 | <artifactId>maven-surefire-plugin</artifactId> |
| 29 | <version>2.4.3</version> |
| 30 | <configuration> |
| 31 | <argLine>-javaagent:${settings.localRepository}/org/jmockit/jmockit/${jmockit.version}/jmockit-${jmockit.version}.jar ${jacocoArgLine}</argLine> |
| 32 | <useSystemClassLoader>true</useSystemClassLoader> |
| 33 | </configuration> |
| 34 | </plugin> |
| 35 | </plugins> |
| 36 | </build> |

运行编译后在target\site\jacoco目录下生成检查结果，如下图：

每个文件，每行，分支覆盖率都可以统计到，非常赞，建议试用。

### JAVA UT实践分享 - UT原则，常见问题及推荐函数命名方法

  日期：2020-11-17 16:50浏览：129[评论：0](http://3ms.huawei.com/km/blogs/details/9342795?l=zh-cn#comment_pop)

**UT原则：**

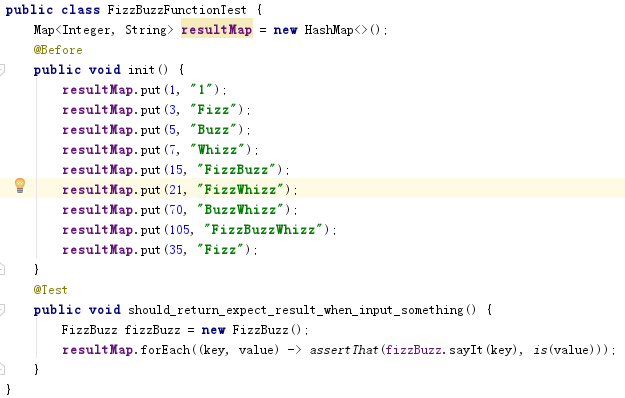
FIRST原则：  
F Fast -- 快速完成运行，秒级反馈结果。超过一分钟，就得让人烦躁了。  
I Isolated -- 独立正交原则，见上篇设计。  
R Repeatable -- 可重复执行。 要是每次执行结果不一样，那就没法自检了，难道人肉看？  
S Self-verifying -- 自检。 作弊的好办法。  
T Timely -- 即时。开发同时就写，得杜绝华为人开发业务代码，找外包写UT的事情。

**常见问题：**

1 命名看不懂: test01。 几天过后，自己绝对凌乱在风中了。。。  
2 职责不单一： 不要一个函数中写N个用例，一旦错了一个，要花很长时间排查。  
3 不独立正交： 独立正交就不多说了。把握好投入产出比，多一个都不要写。  
4 不自检测： 千万别作弊。

5 MOCK管理混乱： 越往后面维护成本越高，绝对比开发业务代码更凌乱。

那么下面这个UT看着如何？

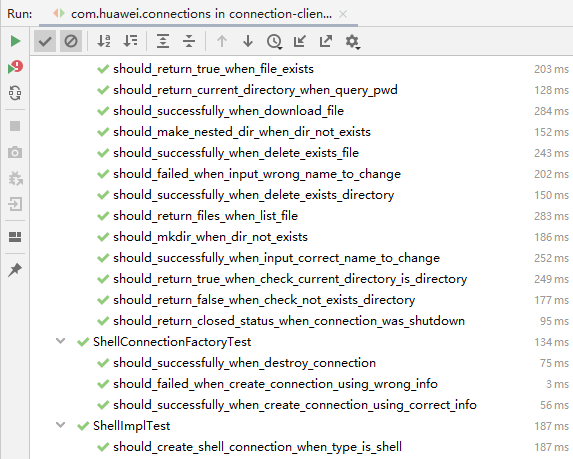


**推荐UT函数命名方法：**

1、MethodName\_StateUnderTest\_ExpectedBehavior  
eg:withdrawMoney\_InvalidAccount\_ExceptionThrown  
  
2、MethodName\_ExpectedBehavior\_StateUnderTest  
eg:withdrawMoney\_ThrowsException\_IfAccountIsInvalid  
  
3、test[Feature being tested]  
eg:testFailToWithdrawMoneyIfAccountIsInvalid  
  
4、Feature to be tested  
eg:FailToWithdrawMoneyIfAccountIsInvalid  
  
5、Should\_ExpectedBehavior\_When\_StateUnderTest  
eg:Should\_FailToWithdrawMoney\_ForInvalidAccount  
  
6、When\_StateUnderTest\_Expect\_ExpectedBehavior  
eg:When\_InvalidAccount\_Expect\_WithdrawMoneyToFail  
  
7、Given\_Preconditions\_When\_StateUnderTest\_Then\_ExpectedBehavior  
eg:Given\_UserIsAuthenticated\_When\_InvalidAccountNumberIsUsedToWithdrawMoney\_Then\_TransactionsWillFail

以上不多说了，英文都能看懂。

最终目的是要做到测试即文档，如下图：



**最后说个小问题，私有函数如何UT？**

如果是无关紧要的私有函数，也在其它测试中顺带着测了，就不必要再单独UT测了。

如果是很重要的逻辑，还是需要测一把。但这个时候是private的，怎么办？ 改成public？

显然也不能为了测试改大范围。 我建议就反射调用测试吧。反射的例子我就不举了，记得setAccess改为true一把。

### JAVA UT实践分享 - 依赖管理及MOCK实现介绍

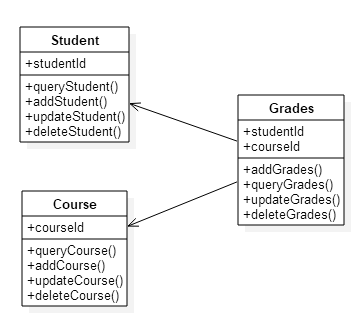
  日期：2020-12-01 17:29浏览：104[评论：0](http://3ms.huawei.com/km/blogs/details/9440391?l=zh-cn#comment_pop)

**什么是打桩**  
打桩，指把桩打进地里，使建筑物基础坚固。  
简单理解就在系统里植入一个稳定依赖，使自己的模块在这个稳定的依赖下，稳定运行。

所以，就有几点关键信息：  
1 这个桩是你的依赖  
2 这个桩是稳定的

**为什么要打桩**  
鉴于上面的打桩定义，如果依赖的模块\系统\服务不稳定，在自测时，鉴于UT的可重复和自检两个原则，就需要一个稳定的依赖（比如每次调用返回结果一样），以实现UT的自检。 所以，就需要将外部依赖打桩，以稳定其表现。

另外还有其他一些原因，比如依赖的模块开发进度慢于自己的进度，要先打桩调试。  
还有像依赖的模块不能本地调用，必须模拟等。

**如何打桩**  
首先要明确打桩对象：**外部依赖**  
举个例子：  
某学生成绩管理系统，分学生管理、课程管理、成绩管理三个模块。成绩管理时要查询学生和课程信息。假设类图如下：  


假设开发成绩管理addGrades，这时需要查询到学生和课程信息，才能录入成绩。  
那么这时就需要对Student的queryStudent接口和Course的queryCourse接口进行打桩测试。因为这两个接口返回值不稳定的话是无法正常测试添加结果是否正常的。  
代码如下：

|  |  |
| --- | --- |
| 1 | /\*学生管理\*/ |
| 2 | public class Student { |
| 3 | private String studentId; |
| 4 |  |
| 5 | public Student(String studentId) { |
| 6 | this.studentId = studentId; |
| 7 | } |
| 8 | } |
| 9 | public class StudentService { |
| 10 | List<Student> queryStudent(){ |
| 11 | return new ArrayList<>(); |
| 12 | } |
| 13 | } |
| 14 |  |
| 15 | /\*课程管理\*/ |
| 16 | public class Course { |
| 17 | private String courseId; |
| 18 |  |
| 19 | public Course(String courseId) { |
| 20 | this.courseId = courseId; |
| 21 | } |
| 22 | } |
| 23 | public class CourseService { |
| 24 | List<Course> queryCourse(){ |
| 25 | return new ArrayList<>(); |
| 26 | } |
| 27 | } |
| 28 |  |
| 29 | /\*成绩管理\*/ |
| 30 | public class Grades { |
| 31 | private String studentId; |
| 32 | private String courseId; |
| 33 | private float grades; |
| 34 |  |
| 35 | public Grades(String studentId, String courseId, float grades) { |
| 36 | this.studentId = studentId; |
| 37 | this.courseId = courseId; |
| 38 | this.grades = grades; |
| 39 | } |
| 40 | } |
| 41 | public class GradesService { |
| 42 | public void addGrades(List<Grades> grades) { |
| 43 | List<Student> students = new StudentService().queryStudent(); |
| 44 | List<Course> courses = new CourseService().queryCourse(); |
| 45 | // check grades exist. |
| 46 | if (!check(grades, students, courses)) { |
| 47 | throw new IllegalArgumentException("input error!"); |
| 48 | } |
| 49 | insertGrades(grades); |
| 50 | } |
| 51 |  |
| 52 | private boolean check(List<Grades> grades, List<Student> students, List<Course> courses) { |
| 53 | return true; |
| 54 | } |
| 55 |  |
| 56 | public void insertGrades(List<Grades> grades) { |
| 57 | // insert into database |
| 58 | } |
| 59 |  |
| 60 | public GradesService queryGrades(String studentId, String courseId) { |
| 61 | return new GradesService(); |
| 62 | } |
| 63 | } |

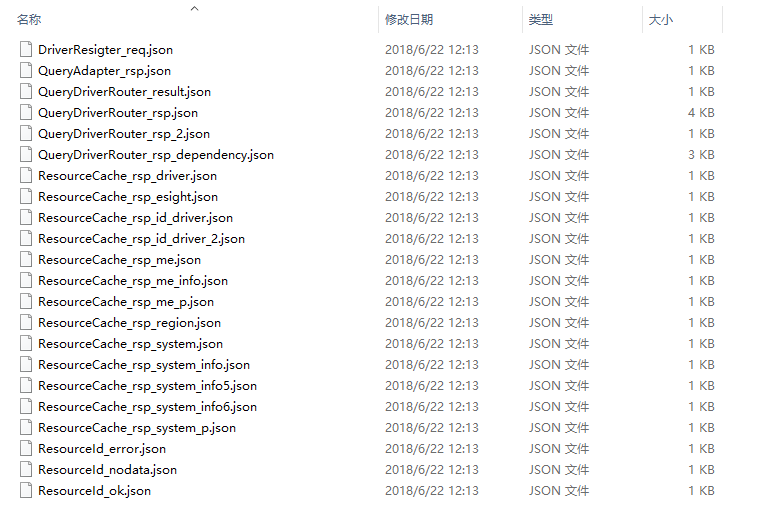
Grades类的addGrades UT应该这样写：

|  |  |
| --- | --- |
| 1 | public class GradesTest { |
| 2 | @Mocked |
| 3 | private StudentService studentService; |
| 4 |  |
| 5 | @Mocked |
| 6 | private CourseService courseService; |
| 7 |  |
| 8 | @Before |
| 9 | public void init() { |
| 10 | new Expectations() { |
| 11 | { |
| 12 | List<Student> students = new ArrayList<>(1); |
| 13 | students.add(new Student("001")); |
| 14 | studentService.queryStudent(); |
| 15 | result = students; |
| 16 | List<Course> courses = new ArrayList<>(1); |
| 17 | courses.add(new Course("001")); |
| 18 | courseService.queryCourse(); |
| 19 | result = courses; |
| 20 | } |
| 21 | }; |
| 22 | } |
| 23 | @Rule |
| 24 | public ExpectedException expectedException = ExpectedException.none(); |
| 25 |  |
| 26 | @Test |
| 27 | public void should\_add\_grades\_successfully\_when\_student\_and\_course\_exist() { |
| 28 | GradesService gradesService = new GradesService(); |
| 29 | List<Grades> gradesList = new ArrayList<>(1); |
| 30 | gradesList.add(new Grades("001", "001", 96)); |
| 31 | gradesService.addGrades(gradesList); |
| 32 | new Verifications(){ |
| 33 | { |
| 34 | studentService.queryStudent(); |
| 35 | times = 1; |
| 36 | courseService.queryCourse(); |
| 37 | times = 1; |
| 38 | } |
| 39 | }; |
| 40 | } |
| 41 |  |
| 42 | @Test |
| 43 | public void should\_failed\_to\_add\_grades\_successfully\_when\_student\_not\_exist() { |
| 44 | expectedException.expect(IllegalArgumentException.class); |
| 45 | expectedException.expectMessage("input error!"); |
| 46 | GradesService gradesService = new GradesService(); |
| 47 | List<Grades> gradesList = new ArrayList<>(1); |
| 48 | gradesList.add(new Grades("002", "001", 96)); |
| 49 | gradesService.addGrades(gradesList); |
| 50 | } |
| 51 |  |
| 52 | @Test |
| 53 | public void should\_failed\_to\_add\_grades\_successfully\_when\_course\_not\_exist() { |
| 54 | expectedException.expect(IllegalArgumentException.class); |
| 55 | expectedException.expectMessage("input error!"); |
| 56 | GradesService gradesService = new GradesService(); |
| 57 | List<Grades> gradesList = new ArrayList<>(1); |
| 58 | gradesList.add(new Grades("001", "002", 96)); |
| 59 | gradesService.addGrades(gradesList); |
| 60 | } |
| 61 | } |

如上代码，实现了对Student和Course依赖的稳定依赖打桩，确保UT的可重复性和自检。  
注：MOCK框架使用的JMockit  
JMockit使用参照：[https://code.huawei.com/mo-architecture/mo-flag/blob/master/ut/Java UT Standard.md](https://code.huawei.com/mo-architecture/mo-flag/blob/master/ut/Java%20UT%20Standard.md)

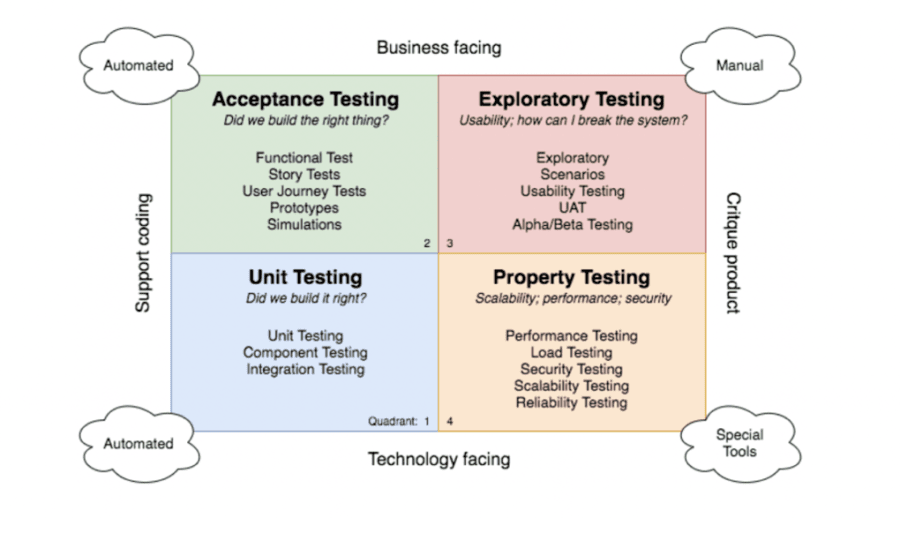
**什么情况下不打桩**  
1 依赖方有稳定依赖的情况。比如一个稳定输入情况下会有稳定输出地，像路由策略这种的，就是一个算法，给定输入会准确返回一个输出的。  
2 自身代码有核心业务逻辑需要依赖方进行处理检验的。比如上述代码中，insert入库，那么sql中包含了核心业务逻辑，需要数据库进行校验。

**使用UT MOCK技术实现集成测试**  
对于上述第二点不打桩的情况，需要进行真实模拟一个数据库。可以通过内存数据库实现真实的SQL执行，以检查核心业务逻辑正确性。进而实现通过UT MOCK技术做到集成测试效果。具体案例且听下次分解。

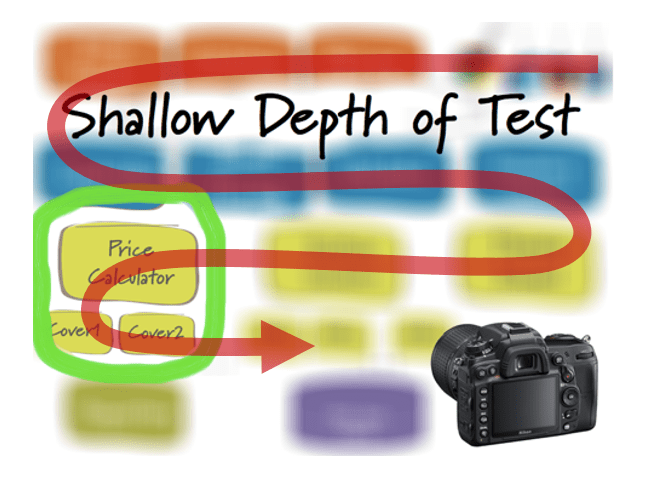
**MOCK要慎重，管理要规范**  
这里不提供实践，只提供警示说明。  
MOCK桩能不用尽量别用，毕竟对方如果变更了，你的桩也需要跟着变更。  
另外对于有些场景下MOCK桩特别多的情况，一定要合理规划管理，比如有文件规范命名等方式。否则MOCK桩多而不规范的话，会导致维护成本激增。  
反例：  
  
上图是我曾经开发的某服务路由策略的桩，基本上一周后我就不知道这些桩是干啥的了。

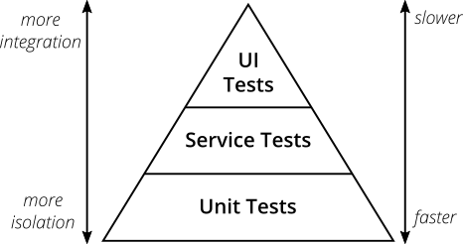
### JAVA UT实践分享 - UT中的测试景深、金字塔、四象限理论

  日期：2020-12-02 09:11浏览：257[评论：0](http://3ms.huawei.com/km/blogs/details/9442379?l=zh-cn#comment_pop)

**测试四象限**  
  
根据技术难度和自动程度分四个象限：  
一象限：自动化程度低、技术难度大，做探索测试。由专业技术人员做。  
二象限：自动化程度低、技术难度小，如Story验收，页面测试等。由普通测试工程师做。  
三象限：自动化程度高、技术难度小，如UT，由开发人员做。  
四象限：自动化程度高、技术难度大，如压力测试，由DFX测试人员做。

不同的测试分层有不同的技术和自动化能力限制，分不同的人实施。开发人员聚焦LLT范围。

**测试景深理论**  
  
当我们在给遗留系统加测试的时候，会习惯性的要给每个模块加测试，而由于遗留系统的设计不好，代码腐化，本身代码都揉是在一起的，每一个模块很难测，于是需要重构，重构又要加测试……于是先有好测的代码，还是先有好的单元测试在很多人眼中看来就变成了一个鸡生蛋蛋生鸡的问题。  
这也是前文提到的，要测试的对象是一堵烂泥墙，没有单元。这个时候，我们可以从整堵墙入口开测，完整的把整个墙跑完，但观察点只聚焦你要观察的单元。这就是测试景深理论。  
测试景深理论就是用景深作为一个隐喻，讲的是虽然我们有很多端到端测试，但其实他们每一组关注的重点是有偏差的。尽管每个用例都走过了所有的代码行，但实际上，可能它只关心其中的一段代码。它关心的代码就像是景深调焦到的地方，显得格外清楚而其他地方其实是我们不在乎的，就像照片中模糊的地方。  
但是景深理论是应用于烂泥墙不得已的场景下，正常单元划分比较好的时候，或者后续单元划开了，还是尽量只跑单元自身代码，毕竟全跑下来，如果别人修改了，影响到自己的用例，维护成本会大大增加。总之，还是根据投入产出比来衡量。

**测试金字塔**  
  
测试金字塔大家不陌生，不同的地方对于上两层可能不太一样，但最下面这层绝对都是UT。  
简单理解，就是成本低的测试（UT最低）多做，成本高的少做（端到端测试），这样总的测试成本最低。  
PS：实际情况下，我们大多是冰淇淋模型：上面多，下面少。 也有菱形模型的。  
今天我想讲的是，UT中也可以有金字塔。

再举上篇文章中的例子。  
成绩管理有以下函数：

|  |  |
| --- | --- |
| 1 | public class GradesService { |
| 2 | public void addGrades(List<Grades> grades) { |
| 3 | List<Student> students = new StudentService().queryStudent(); |
| 4 | List<Course> courses = new CourseService().queryCourse(); |
| 5 | // check grades exist. |
| 6 | if (!check(grades, students, courses)) { |
| 7 | throw new IllegalArgumentException("input error!"); |
| 8 | } |
| 9 | insertGrades(grades); |
| 10 | } |
| 11 |  |
| 12 | private boolean check(List<Grades> grades, List<Student> students, List<Course> courses) { |
| 13 | return true; |
| 14 | } |
| 15 |  |
| 16 | public void insertGrades(List<Grades> grades) { |
| 17 | // insert into database |
| 18 | } |
| 19 |  |
| 20 | public GradesService queryGrades(String studentId, String courseId) { |
| 21 | return new GradesService(); |
| 22 | } |
| 23 | } |

可以看出有查询和添加两大功能，添加又分为校验和入库两个子功能。  
那么UT应该这样写更好：

|  |  |
| --- | --- |
| 1 | /\*正常录入成绩能够走到insertData函数\*/ |
| 2 | public void should\_add\_grades\_successfully\_when\_student\_and\_course\_exist() |
| 3 |  |
| 4 | /\*学生信息不正确时抛异常\*/ |
| 5 | public void should\_failed\_to\_add\_grades\_successfully\_when\_student\_not\_exist() |
| 6 |  |
| 7 | /\*课程信息不正确时抛异常\*/ |
| 8 | public void should\_failed\_to\_add\_grades\_successfully\_when\_course\_not\_exist() |
| 9 |  |
| 10 | /\*成绩信息小于0时抛异常\*/ |
| 11 | public void should\_failed\_to\_add\_grades\_successfully\_when\_grade\_less\_than\_0() |
| 12 |  |
| 13 | /\*成绩信息大于100时抛异常\*/ |
| 14 | public void should\_failed\_to\_add\_grades\_successfully\_when\_grade\_grate\_than\_100() |
| 15 |  |
| 16 | /\*正确的学生数据能够录入到数据库中\*/ |
| 17 | public void should\_succefully\_insert\_data\_when\_grades\_correct() |
| 18 |  |
| 19 | /\*能够查询到已录入的成绩信息\*/ |
| 20 | public void should\_get\_right\_data\_when\_query\_grades() |

以上为各个功能单元点的UT。  
可以在上面基础上，增加一个端到端的测试：

|  |  |
| --- | --- |
| 1 | /\*当录入正常的学生成绩信息后，能够从数据库中查询出来\*/ |
| 2 | public void should\_get\_right\_date\_from\_db\_when\_add\_correct\_grades() |

这样，测试就有两层，第一层是各函数粒度的UT，第二层是整个GradesService的端到端测试。第一层7个用例，第二层只用1个用例。这也是测试金字塔。

更多案例参照：<https://gitlab.huawei.com/mo-architecture/connection-client>

### JAVA UT实践分享 - 小技术之异步代码调用UT方案

  日期：2020-12-02 09:58浏览：58[评论：0](http://3ms.huawei.com/km/blogs/details/9442829?l=zh-cn#comment_pop)

UT绕不过异步问题，所以今天就讲讲异步情况下怎么破。  
**四大线程同步神器**  
对于正常情况下的异步，建议尽量在代码设计阶段预留好测试口子，尽量用CountDownlatch, Semaphore, Barrier, FutureTask线程同步器搞定。  
PS：**千万别sleep循环**，因为各种机器执行时长有差异，sleep时间不好设置，设置长了会造成UT时间延长，体验差；设置短了可能会UT失败。。。

**异步UT神器**  
今天重点推荐一款异步代码UT测试神器  
Awaitility官网：<https://github.com/awaitility/awaitility/wiki/Usage>

特性大概总结：  
1 多线程同步状态校验。  
2 支持设置超时时间。  
3 支持访问类的静态或对象实例的私有变量或函数。  
4 支持各种构建工具  
5 支持自检  
6 … …

示例：

|  |  |
| --- | --- |
| 1 | /\*功能代码，单独线程执行初始化，初始化成功后标记成功，记录到flagMap中\*/ |
| 2 | public class TestAwait { |
| 3 | private volatile boolean initFlag = false; |
| 4 | Map<String, Boolean> flagMap = new HashMap<>(); |
| 5 | public void init() { |
| 6 | //定义线程 |
| 7 | Thread thread = new Thread(() -> { |
| 8 | try { |
| 9 | Thread.sleep(1000); |
| 10 | //记录标识 |
| 11 | flagMap.put("test", true); |
| 12 | initFlag = true; |
| 13 | } catch (InterruptedException e) { |
| 14 | e.printStackTrace(); |
| 15 | } |
| 16 | }); |
| 17 | thread.start(); |
| 18 | } |
| 19 |  |
| 20 | public boolean isInitFlag() { |
| 21 | return initFlag; |
| 22 | } |
| 23 | } |

异步测试如下：

|  |  |
| --- | --- |
| 1 | /\*异步测试代码\*/ |
| 2 | public class TestAwaitTest { |
| 3 | /\*对异常变更的私有成员变量进行监视\*/ |
| 4 | @Test |
| 5 | public void should\_set\_init\_flat\_to\_true\_when\_finish\_initalize() throws Exception{ |
| 6 | TestAwait testAwait = new TestAwait(); |
| 7 | testAwait.init(); |
| 8 | await().until(isTrue(fieldIn(testAwait).ofType(Map.class).andWithName("flagMap").call())); |
| 9 | assertTrue(testAwait.isInitFlag()); |
| 10 | } |
| 11 |  |
| 12 | private Callable<Boolean> isTrue(Map flag) { |
| 13 | System.out.println("flag = " + flag); |
| 14 | return () -> { |
| 15 | Boolean flagB = (Boolean) flag.get("test"); |
| 16 | System.out.println("-----flag = " + flag); |
| 17 | return flagB != null && flagB; |
| 18 | }; |
| 19 | } |
| 20 | } |

依赖包：

|  |  |
| --- | --- |
| 1 | <dependency> |
| 2 | <groupId>org.awaitility</groupId> |
| 3 | <artifactId>awaitility</artifactId> |
| 4 | <version>3.0.0</version> |
| 5 | <scope>test</scope> |
| 6 | </dependency> |

### JAVA UT实践分享 - 小技巧之用UT技术实现集成测试

  日期：2020-12-02 12:47浏览：58[评论：0](http://3ms.huawei.com/km/blogs/details/9444191?l=zh-cn#comment_pop)

**UT最大的拦路虎莫过于复杂的各种外部中间件、服务依赖难以有稳定的桩**

所以，要实现有效UT，必须有效地将各种外部依赖有效模拟。  
而且，UT成本是最小的，如果能够用UT实现外部依赖的集成测试，会极大的降低测试成本。

不过，仅用JMockit是不够的，毕竟像数据库那样式的，关键逻辑在SQL里，如果把DAO层MOCK掉了，那整个DAO层测试就缺失了。

**如何用UT实现集成测试？ 举三个典型例子：**  
**例一：数据库\中间件等模拟**  
数据库使用h2或者hsqldb引入内存数据库，真实模拟SQL。  
KAFKA等中间件可以使用其内存版本模拟。  
详细参照：[<https://code.huawei.com/mo-architecture/mo-flag/blob/master/ut/Java> UT [Standard.md](http://standard.md/)](<https://code.huawei.com/mo-architecture/mo-flag/blob/master/ut/Java> UT [Standard.md](http://standard.md/))

**例二：协议真实请求模拟**  
比如ManageOne的RUCOR框架，其中核心逻辑是组装HTTPS请示报文，发送请示，然后解析转换消息返回。  
对于HTTPS请求，一是验证消息通道，二是验证证书加载校验。  
测试代码示例：

首先在测试代码中启动一个HTTPS MOCK SERVER

|  |  |
| --- | --- |
| 1 | /\*spring boot启动https mock server\*/ |
| 2 | @RestController |
| 3 | public class DataController { |
| 4 | @RequestMapping(value = "/data") |
| 5 | public ResponseEntity metrics() { |
| 6 | String monitorResult = "{\"code\":\"0\", \"data\":[{\"DN\":\"testid\", \"NAME\":\"testname\"}," |
| 7 | + "{\"DN\":\"testid2\", \"NAME\":\"niuniu\"}]}"; |
| 8 | return ResponseEntity.ok().header("Content-Type", "application/json").body(monitorResult); |
| 9 | } |
| 10 | } |
| 11 | @SpringBootApplication(exclude= {DataSourceAutoConfiguration.class}) |
| 12 | public class MockServerApplication { |
| 13 | public static void main(String[] args) { |
| 14 | SpringApplication.run(MockServerApplication.class); |
| 15 | } |
| 16 | } |
| 17 | public class HttpsMockServer { |
| 18 | private volatile static boolean isServerStarted = false; |
| 19 | public static void startServer() throws Exception { |
| 20 | if (isServerStarted) { |
| 21 | return; |
| 22 | } |
| 23 | synchronized (HttpsMockServer.class) { |
| 24 | if (isServerStarted) { |
| 25 | return; |
| 26 | } |
| 27 | CertificateUtil.createJksKeyStoreFile(); |
| 28 | CountDownLatch countDownLatch = new CountDownLatch(1); |
| 29 | new Thread(() -> { |
| 30 | MockServerApplication.main(null); |
| 31 | countDownLatch.countDown(); |
| 32 | }).start(); |
| 33 | countDownLatch.await(); |
| 34 | isServerStarted = true; |
| 35 | } |
| 36 | } |
| 37 | } |

application.yml如下：（证书提前生成）

|  |  |
| --- | --- |
| 1 | server: |
| 2 | port: 32927 |
| 3 | context-path: / |
| 4 | ssl: |
| 5 | key-alias: mockserver |
| 6 | key-password: test123 |
| 7 | key-store: classpath:mockserver.jks |
| 8 | key-store-provider: SUN |
| 9 | key-store-type: JKS |

然后，UT实现代码如下：

|  |  |
| --- | --- |
| 1 | public class TestExecutorWithLocalCerts { |
| 2 | @Before |
| 3 | public void startMockServer() throws Exception { |
| 4 | HttpsMockServer.startServer();//启动http mock server |
| 5 | } |
| 6 |  |
| 7 | @Test |
| 8 | public void should\_successfully\_get\_convert\_result\_when\_execute\_template\_with\_local\_cert() throws Exception { |
| 9 | KeyStore keyStore; |
| 10 | try (FileInputStream is = new FileInputStream(keyStorePath);){ |
| 11 | keyStore = KeyStore.getInstance("jks"); |
| 12 | keyStore.load(is, "test123".toCharArray()); |
| 13 | } catch (KeyStoreException | IOException | NoSuchAlgorithmException | CertificateException e) { |
| 14 | throw new RucorException("Failed to load key store", e); |
| 15 | } |
| 16 | RequestParameters request = new RequestParameters(); |
| 17 | /\*设置请求参数\*/ |
| 18 | request.setType("eSight"); |
| 19 | request.setVersion("V300R002C00"); |
| 20 | request.setCategory("MODEL"); |
| 21 | request.setItem("DATA"); |
| 22 | Node node = new Node(); |
| 23 | node.setId("00"); |
| 24 | node.setIp("127.0.0.1"); |
| 25 | node.setPort("32927"); |
| 26 | request.setNode(node); |
| 27 | ... ... |
| 28 | Map<String, Object> map = Maps.newHashMap(); |
| 29 | map.put("CI\_ID", "HOST"); |
| 30 | request.setTemplateParams(map); |
| 31 |  |
| 32 | ResponseParameters response; |
| 33 | try { |
| 34 | /\*发送请求\*/ |
| 35 | response = new RucorAdapterImpl().executeAsClient(request); |
| 36 | } catch (RucorException e) { |
| 37 | LOG.error("RucorException", e); |
| 38 | throw e; |
| 39 | } |
| 40 | Assert.assertEquals( |
| 41 | ((Map<String, String>) response.getReserveParams().get(RucorConstant.CATEGORY\_RES\_HEADER).get("Header")) |
| 42 | .get("HttpCode"), "200"); |
| 43 | } |
| 44 | } |

**例三：外部服务依赖**  
对于外部服务依赖，可以使用rest mock server替代，比如wiremock:<http://wiremock.org/>。

闫浩同学在此基础上封装了更好使的的swagger-server: <http://rnd-isourceb.huawei.com/y00452130/swagger-server>

本地clone后执行以下命令：

|  |  |
| --- | --- |
| 1 | cd swagger-server |
| 2 | mvn clean install |
| 3 | java -jar ./target/swagger-server-1.0-SNAPSHOT.jar e:/MetricsManagement.yaml |

启动端口为9090。  
启动日志中显示正常注册的URL：

|  |  |
| --- | --- |
| 1 | 2020-12-02 15:40:18 [main] INFO Swagger20Parser:88 - reading from e:/MetricsManagement.yaml |
| 2 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics] |
| 3 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics/{metrics-id}] |
| 4 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-unit] |
| 5 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-unit/{unit-id}] |
| 6 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-template] |
| 7 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-template/{template-id}] |
| 8 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-object-templates] |
| 9 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-object-templates/{template-id}] |
| 10 | 2020-12-02 15:40:18 [main] INFO ParseRunner:70 - Added request path: [/metrics-object-template/ci-type/{ci-type}] |
| 11 | 2020-12-02 15:40:19 [main] INFO log:186 - Logging initialized @2154ms |
| 12 | 2020-12-02 15:40:19 [main] INFO Server:327 - jetty-9.2.z-SNAPSHOT |
| 13 | 2020-12-02 15:40:19 [main] INFO ContextHandler:744 - Started w.o.e.j.s.ServletContextHandler@5118388b{/\_\_admin,null,AVAILABLE} |
| 14 | 2020-12-02 15:40:19 [main] INFO ContextHandler:744 - Started w.o.e.j.s.ServletContextHandler@4a3e3e8b{/,null,AVAILABLE} |
| 15 | 2020-12-02 15:40:19 [main] INFO NetworkTrafficServerConnector:266 - Started NetworkTrafficServerConnector@2fab4aff{HTTP/1.1}{0.0.0.0:9090} |

在浏览器中输入: <http://localhost:9090/__admin/> 可以查看全部接口  
在浏览器中输入相应的注册接口，返回接口定义的值。剩下的就是愉快的玩耍了。

总而言之，用UT框架实现集成测试，只要肯想办法，总会搞定的。

**再次敲黑板：UT的成本综合来看是最小的，一定尽量用UT实现更多更高层面的测试。减少测试成本。**

### JAVA UT实践分享 - 小技巧之Spring Test

  日期：2020-12-02 17:24浏览：68[评论：0](http://3ms.huawei.com/km/blogs/details/9452233?l=zh-cn#comment_pop)

Spring大家太熟悉了，直接写示例吧。  
本例以springboot来讲解。

**包引用**

|  |  |
| --- | --- |
| 1 | <dependency> |
| 2 | <groupId>org.springframework.boot</groupId> |
| 3 | <artifactId>spring-boot-starter-web</artifactId> |
| 4 | <version>${sprintboot.version}</version> |
| 5 | </dependency> |
| 6 | <dependency> |
| 7 | <groupId>org.springframework.boot</groupId> |
| 8 | <artifactId>spring-boot-starter-test</artifactId> |
| 9 | <version>${sprintboot.version}</version> |
| 10 | </dependency> |

业务代码：

|  |  |
| --- | --- |
| 1 | @RestController |
| 2 | public class NewPortalController { |
| 3 | @RequestMapping(value = "/portal") |
| 4 | public String index() { |
| 5 | return "index.html"; |
| 6 | } |
| 7 | } |
| 8 |  |
| 9 | /\*依赖了NewPortalController的接口\*/ |
| 10 | @RestController |
| 11 | public class ForwardPortalController { |
| 12 | @Autowired |
| 13 | private NewPortalController newPortalController; |
| 14 |  |
| 15 | @RequestMapping(value = "/forward-portal") |
| 16 | public String index() { |
| 17 | return "/forward" + newPortalController.index(); |
| 18 | } |
| 19 | } |
| 20 |  |
| 21 | /\*启动类\*/ |
| 22 | @SpringBootApplication |
| 23 | public class ServiceMgrApplication { |
| 24 | public static void main(String[] args) { |
| 25 | SpringApplication.run(ServiceMgrApplication.class); |
| 26 | } |
| 27 | } |

**测试场景一：context加载**

|  |  |
| --- | --- |
| 1 | @RunWith(SpringRunner.class) |
| 2 | @SpringBootTest(classes = com.huawei.manageone.servicemgr.ServiceMgrApplication.class) |
| 3 | public class ApplicationTest { |
| 4 | @Autowired |
| 5 | private NewPortalController newPortalController; |
| 6 | @Test |
| 7 | public void should\_load\_successfully\_when\_init\_finished() { |
| 8 | assertThat(newPortalController).isNotNull(); |
| 9 | } |
| 10 | } |

**测试场景二：接口模拟调用**

|  |  |
| --- | --- |
| 1 | @RunWith(SpringRunner.class) |
| 2 | @SpringBootTest(classes = com.huawei.manageone.servicemgr.ServiceMgrApplication.class) |
| 3 | @AutoConfigureMockMvc |
| 4 | public class PortalControllerMockTest { |
| 5 | @Autowired |
| 6 | private MockMvc mockMvc; |
| 7 |  |
| 8 | @Test |
| 9 | public void should\_get\_default\_portal\_url\_when\_get\_portal() throws Exception { |
| 10 | mockMvc.perform(get("/portal")).andExpect(status().isOk()).andExpect(content().string(containsString("index.html"))); |
| 11 | } |
| 12 | } |

以上测试不会真实启动web容器，不会起HTTP监听。  
执行时间：230ms

**测试场景三：真实WEB容器启动测试**

|  |  |
| --- | --- |
| 1 | @RunWith(SpringRunner.class) |
| 2 | @SpringBootTest(classes = com.huawei.manageone.servicemgr.ServiceMgrApplication.class, |
| 3 | webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT) |
| 4 | public class PortalControllerTest { |
| 5 | @LocalServerPort |
| 6 | private int port; |
| 7 |  |
| 8 | @Autowired |
| 9 | private TestRestTemplate testRestTemplate; |
| 10 |  |
| 11 | @Test |
| 12 | public void should\_get\_default\_portal\_url\_when\_get\_portal() { |
| 13 | assertThat(testRestTemplate.getForObject("http://localhost:" + port +"/portal", String.class)).contains("index.html"); |
| 14 | } |
| 15 | } |

port为随机分配的本地端口，用以容器启动监听端口。  
执行时间：400ms

**测试场景四：模拟测试，并且只加载要测试的对象**

|  |  |
| --- | --- |
| 1 | @RunWith(SpringRunner.class) |
| 2 | @ContextConfiguration(classes = com.huawei.manageone.servicemgr.ServiceMgrApplication.class) |
| 3 | @WebMvcTest(controllers = NewPortalController.class) |
| 4 | public class PortalControllerUniqMockTest { |
| 5 | @Autowired |
| 6 | private MockMvc mockMvc; |
| 7 |  |
| 8 | @Test |
| 9 | public void should\_get\_default\_portal\_url\_when\_get\_portal() throws Exception { |
| 10 | mockMvc.perform(get("/portal")).andExpect(status().isOk()).andExpect(content().string(containsString("index.html"))); |
| 11 | } |
| 12 | } |

不启动容器，并且只加载要用的bean。执行时间216ms

**测试场景五：依赖MOCK**

|  |  |
| --- | --- |
| 1 | @RunWith(SpringRunner.class) |
| 2 | @ContextConfiguration(classes = com.huawei.manageone.servicemgr.ServiceMgrApplication.class) |
| 3 | @WebMvcTest(controllers = {NewPortalController.class, ForwardPortalController.class}) |
| 4 | public class ForwardPortalControllerUniqMockTest { |
| 5 | @Autowired |
| 6 | private MockMvc mockMvc; |
| 7 |  |
| 8 | @MockBean |
| 9 | private NewPortalController newPortalController; |
| 10 |  |
| 11 | @Test |
| 12 | public void should\_get\_default\_portal\_url\_when\_get\_portal() throws Exception { |
| 13 | when(newPortalController.index()).thenReturn("/haha.html"); |
| 14 | mockMvc.perform(get("/forward-portal")).andExpect(status().isOk()).andExpect(content().string(containsString("/forward/haha.html"))); |
| 15 | } |
| 16 | } |

对ForwardPortalController进行测试，ForwardPortalController依赖了NewPortalController的/portal接口，测试时将/portal接口mock。  
执行时间216ms。

**测试场景六：不用spring框架单独测试单元逻辑**

|  |  |
| --- | --- |
| 1 | public class NewPortalControllerInitTest { |
| 2 | @Test |
| 3 | public void should\_return\_index\_url\_when\_invoke\_index(){ |
| 4 | assertThat(new NewPortalController().index(), is("index.html")); |
| 5 | } |
| 6 | } |

执行时长6ms。 差距50多倍。虽然几百ms也还可以忍受，但是数量多了也架不住。  
所以，独立的单元逻辑就别用SPRING了，直接写UT，效率更高。尽量用spring框架测金字塔上端的拉通测试。

### JAVA UT实践分享 - 优秀案例赏析之SonarQube UT框架

  日期：2020-12-02 17:37浏览：90[评论：0](http://3ms.huawei.com/km/blogs/details/9454667?l=zh-cn#comment_pop)

今年在扩展ManageOne门禁规则，使用SonarQube插件机制开发时，被其UT自检框架惊艳到了。  
先看一段代码：

|  |  |
| --- | --- |
| 1 | public class MOForbiddenConfigPasswordInFile extends FileCheck { |
| 2 | private static final Logger LOG = LoggerFactory.getLogger(MOForbiddenConfigPasswordInFile.class); |
| 3 |  |
| 4 | private static final String[] PASSWORDS = new String[] { |
| 5 | "Huawei12#$", "Huawei@123", "Huawei123", "huawei@123", "huawei123", "Changeme\_123" |
| 6 | }; |
| 7 |  |
| 8 | @Override |
| 9 | public void scanFile(InputFile inputFile) { |
| 10 | try (BufferedReader reader = new BufferedReader(new InputStreamReader(inputFile.inputStream()))) { |
| 11 | AtomicInteger lineNum = new AtomicInteger(0); |
| 12 | reader.lines().forEach(lineContent -> { |
| 13 | // 执行检查 |
| 14 | int now = lineNum.incrementAndGet(); |
| 15 | if (Arrays.stream(PASSWORDS).anyMatch(lineContent::contains)) { |
| 16 | TextRange textRange = inputFile.selectLine(now); |
| 17 | reportIssueOnFile("Forbidden to config password in file.", textRange); |
| 18 | } |
| 19 | }); |
| 20 | } catch (IOException e) { |
| 21 | LOG.error("Failed to check file {}", inputFile.filename(), e); |
| 22 | } |
| 23 | } |
| 24 | } |

以上代码逻辑很简单，就是判断入参inputFile的文件内容中有没有PASSWORDS列表中定义的明文密码，有的话调用reportIssueOnFile接口上报检查异常。

测试代码如下：

|  |  |
| --- | --- |
| 1 | @Test |
| 2 | public void should\_verify\_issue\_when\_pom\_configuration\_without\_compile\_lint() { |
| 3 | XayPublicFileCheckVerifier.verifyIssues("nook.properties", new MOForbiddenConfigPasswordInFile()); |
| 4 | } |
| 5 |  |
| 6 | @Test |
| 7 | public void should\_not\_verify\_issue\_when\_pom\_configuration\_with\_compile\_lint() { |
| 8 | XayPublicFileCheckVerifier.verifyNoIssue("ok.cfg", new MOForbiddenConfigPasswordInFile()); |
| 9 | } |

简不简单，清不清爽！  
刚开始，很不解，用junit写UT的话，需要一个执行结果，再加一个预期结果，然后两个比较。  
而这个测试代码中，verifyNoIssue的两个参数，一个是检查规则实现类即业务逻辑，一个是要检查的文件，即入参，没有看到预期结果呀，神奇了！！！

仔细看了原测试用例，发现，它将预期结果预置在了入参文件中。  
看下文：

|  |  |
| --- | --- |
| 1 | Key=Value |
| 2 | Huawei@123 // Noncompliant |
| 3 | #test |

注意到了没，第2行后面有个// Noncompliant注解，这就是关键所在了。

原来SonarQube框架是一个高度抽象封装的一个框架，它卡住代码检测的入出口两端，入口是inputFile，出口是reportIssue，那么它在加载要测试文件转换成inputFile时，将注解标识的行，及使用的检测类都作为预期结果了。

这样UT写起来简直了！