交大交交

测试评估报告

版本 1.0

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| **日期** | **版本** | **说明** | **作者** |
| 17 /05/2020 | 1.0 | 白盒测试报告 | 林江浩 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

1. 简介 4

1.1 目的 4

1.2 范围 4

1.3 定义、首字母缩写词和缩略语 4

1.4 参考资料 4

1.5 概述 4

2. 测试过程明细 5

2.1 关于自主测试框架的搭建 5

2.2 Content微服务模块的测试 5

2.3 Message微服务模块的测试 6

2.4 SellInfo和File微服务模块的测试 8

3. 测试结果概述 11

4. 测试结果展示图 11

测试评估报告

# 简介

本测试报告是针对交大交交二手交易APP的测试文档。交大交交二手交易APP能让交大内的同学在平台上进行二手交易信息的发布。本次测试针对面向用户输出的四个模块进行测试。本次测试为白盒测试，采用了基于路径的测试方法和基于数据流的测试方法，所有测试用例按服务名存储在四个文件中，本次测试采用goconvey作为测试框架，gomod管理工程的依赖，由于测试仅包含四个模块，针对相应服务中存在的依赖，使用mock的方式进行处理，默认在测试该服务时，其他服务的正确性，使用VsCode编写和运行测试用例。

## 目的

本测试报告覆盖了针对Message, Content, SellInfo, File四个模块的白盒测试，在测试方法上我们实践了基于DD路径、McCabe路径和基于数据流的测试方法，说明了这些测试用例设计的过程、原因和得出的结论。

## 范围

测试范围为Content，Message，SellInfo，File四个微服务，涉及服务之间的调用以及服务内部的处理，选择这四个微服务的理由在于这四个微服务基本涵盖了较为复杂、并且。其他的微服务，BuyInfo和SellInfo本质上是等价关系，因此仅挑选SellInfo即可；Tag、Avatar微服务代码过于简单，在本次课程实验中不具备复杂性的要求，故省去；User、Auth微服务则涉及交大Jaccount账号权限，因此在本次测试中不予以测试。

而针对上述四个微服务模块，我们对Content模块采用基于DD路径的测试方法，对Message模块采用基于McCabe模块的测试方法，对SellInfo和File模块采用基于数据流的测试方法。

## 定义、首字母缩写词和缩略语

|  |  |
| --- | --- |
| 缩写词 | 含义 |
| 交大交交 | 本次测试所针对的接口类和软件的名称 |
| 白盒测试 | 基于代码及其内在逻辑的测试方法 |
| goconvey | 本次测试使用go语言的测试框架 |

## 参考资料

《Software Testing - A Craftsman’s Approach》

## 概述

本次测试报告包含针对测试用例设计的说明，测试结果和覆盖率展示，以及对被测试软件改进的建议

# 测试过程明细

## 关于自主测试框架的搭建

本次白盒测试的代码和上次黑盒测试的对象是完全一致的，可以理解为是具有连续性的测试过程。但是在上一次黑盒测试的过程之中，我们发现，在完成测试用例的设计后，将其编码为测试代码并运行反而占去了大半时间，而这个测试代码编写的过程本身具有冗余性和不灵活性（测试用例内容是嵌入在测试代码之中，后期难以调整和维护）。

因此，在本次白盒测试中，我们基于GoConvey框架，在目标测试代码上自行设计了一层测试框架，并单独部署在一个测试服务器上。这层框架允许我们通过SSH远程连接，将测试用例以JSON文件的形式给出，并进行自动测试。

如此的框架，减少了我们编写测试代码的冗余性、提升了后期调试的可维护性，极大地方便了我们的测试实践过程，将更多的精力放在测试用例的设计上。

具体的框架代码，我们已经开源至Github：<https://github.com/sjtu-jiaojiao/SJTU-JiaoJiao/tree/fea_tmp>

## Content微服务模块的测试

针对Content微服务模块，我们采用基于DD路径的测试方法。Content模块共有6个待测函数，分别为Create，CreateTag，Update，Query，Delete，Check。因为基于DD路径的测试方法本身存在多种覆盖评价标准，因此我们对它们尽可能采用不同的覆盖方法进行测试，包括语句覆盖、条件覆盖、多条件覆盖、分支+循环覆盖、包含最多2次循环的所有路径、路径测试，力求实践尽可能多的测试方法和指标。

关于基于DD路径的测试方法的覆盖指标的描述如下：

* 语句覆盖，即测试用例覆盖所有执行语句；
* 条件覆盖，即测试用例覆盖每一个条件；
* 多条件覆盖，即测试用例覆盖每种条件组合；
* 循环覆盖，即测试用例覆盖不循环、1次循环、2次循环、多次循环等情况；
* 包含最多2次循环的所有路径，意义显然；
* 路径测试，即测试用例要覆盖所有可能执行的路径。

下面我们将以Content.Check和多条件覆盖指标为例，介绍基于DD路径的测试用例生成的过程，其他函数和用例生成的细节可见附件。以下为Content.Check的设计过程：

1. 470行检验两个参数contentID和contentToken是否为空，因此需要有4种组合来遍历。
2. 474行跳转入内部函数validCheck。
3. 484行结果总为true，也即该检查事实上是不必要的代码，因为在外部已经检查。所以会导致485行是无论如何不可达的代码，无法覆盖测试。
4. 492行检查contentID是否有效，需要2种组合，且之前的用例中已经包含了有效的那一种。
5. 500行检查contentToken是否有效，需要2种组合，且之前的用例中已经包含了有效的那一种。
6. 总计6个测试用例，其DD路径图如下图所示，而具体的测试用例可见附件。

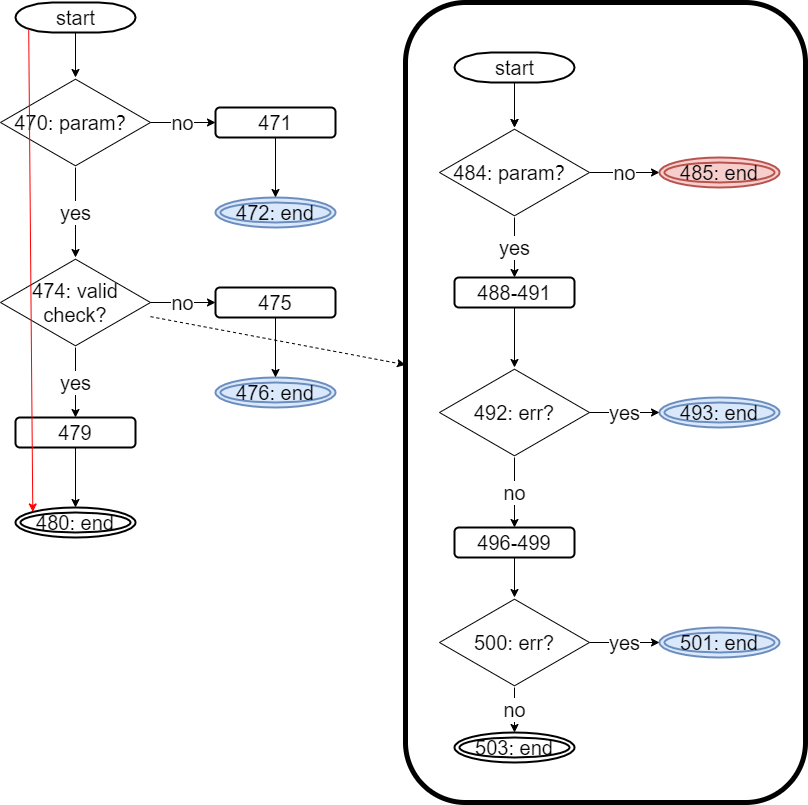


图2.2.1 Content.Check的DD路径图

## Message微服务模块的测试

对于Message模块，我们采用基于计算McCabe复杂度+DFS的方法进行测试。Message模块一共有三个函数，我们选取其中的Query函数作为代表，详细介绍基于McCabe复杂度+DFS的测试用例的生成过程。而其他函数的测试用例生成过程略去，具体材料我们放在附件之中。以下具体步骤：

1. 绘制目标函数的程序流图

Query函数的程序流图如下，图中蓝色节点和红色节点均代表函数出口，蓝色节点代表下层模块的故障，而红色节点的代表输入与程序的逻辑故障。

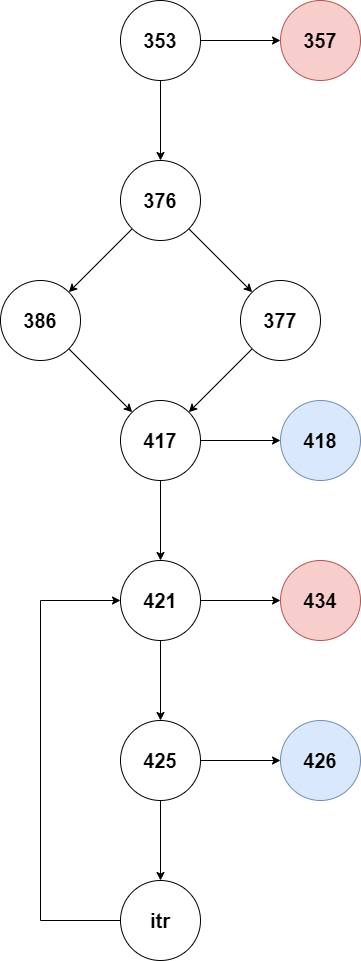


图2.3.1 Query函数的程序流图

1. 根据路径计算McCabe复杂度

由于我们的函数都是多出口的函数，为了使图成为强连通图，从图的出口点到入口点加一条用虚线表示的有向边，使图成为强连通图。之后使用公式



进行计算。下层模块错误退出在图例中将以蓝色体现，逻辑错误退出在图例中以红色体现。在此种情况下我们再进行McCabe复杂度的计算。而Message 模块一共有三个函数，计算结果如下：

(a \*srv) Create(ctx context.Context, req \*message.MessageCreateRequest, rsp \*message.MessageCreateResponse) error

其节点数量为16，边数量为17。最终得出V(G)=7



(a \*srv) Find(ctx context.Context, req \*message.MessageFindRequest, rsp \*message.MessageFindResponse) error

其节点数量为19，边数量为22，最终得出V(G)=8

(a \*srv) Query(ctx context.Context, req \*message.MessageQueryRequest, rsp \*message.MessageQueryResponse) error

其节点数量为9，边数量为10，最终得出V(G)=4



1. 统计变量的定义/使用节点

根据程序流图，对其进行DFS，找出V(G)条基路径，并根据对应的路径生成测试用例。

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库 | | | |
| fromUser | toUser | badge | infos |
| 1 | 2 | 1 | [{unread:false,forward:false},{unread:false,forward:false}] |
| 2 | 1 | 1 | [{unread:false,forward:false},{unread:false,forward:false}] |
| 3 | 1 | 1 | [{unread:false,forward:true},{unread:false,forward:true}] |
| 1 | 3 | 1 | [{unread:true,forward:true},{unread:true,forward:true}] |

表2.3.1 Query函数测试的数据库预插入数据

|  |  |
| --- | --- |
| 路径 | 请求数据 |
| 353-376-421-425-itr-421-434 | {userID:1, oldMsg:1} |
| 353-376-421-425-itr-421-434 | {userID:4, oldMsg:0} |
| 353-357 | {userID:null} |
| 353-376-386-421-425-itr-421-434 | {userID:3, oldMsg:0} |

表2.3.2 Query函数路径与测试用例

以上就是基于McCabe的测试方法的具体过程，从理论而言，通过计算McCabe复杂度并结合深度优先搜索所得出的测试用例一定能够完全覆盖所有路径，从而覆盖所有代码。而实测可得，Message模块的代码覆盖率为100%。

## SellInfo和File微服务模块的测试

对于SellInfo和File模块，我们采用基于数据流的测试用例生成方法。SellInfo模块一共有四个函数，File模块一共有三个函数，我们选取其中的SellInfo. Query函数作为代表，详细介绍基于数据流的测试用例的生成过程。而其他函数的测试用例生成过程略去，具体材料我们放在附件之中。

需要注意的是，为了方便测试和展示，在本小节中，我们对所有的目标测试函数进行行号重排，即行号自函数头从1开始重新计数。以下为具体步骤：

1. 绘制目标函数的程序图

在对SellInfo.Query函数进行行号重排后，我们可以画出对应的程序图。为了展示方便，且因为源代码中存在无意义的空行或括号单行，同时援引教材9.1.6中的方法，我们对原始的程序图进行节点合并与化简，得到最终可用的程序图。

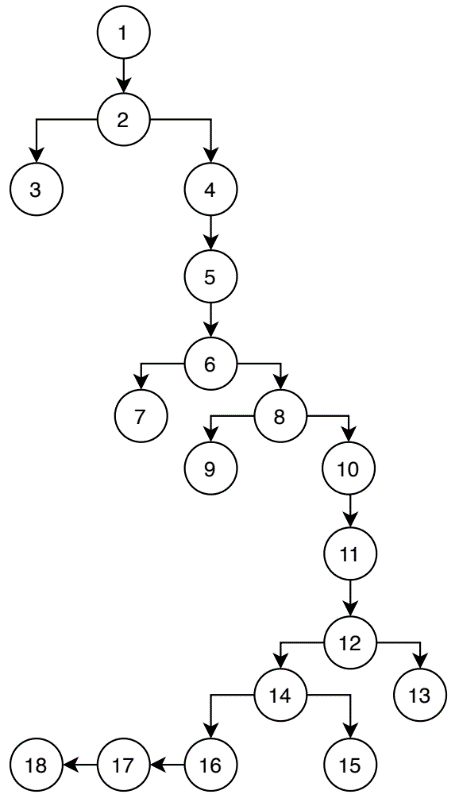
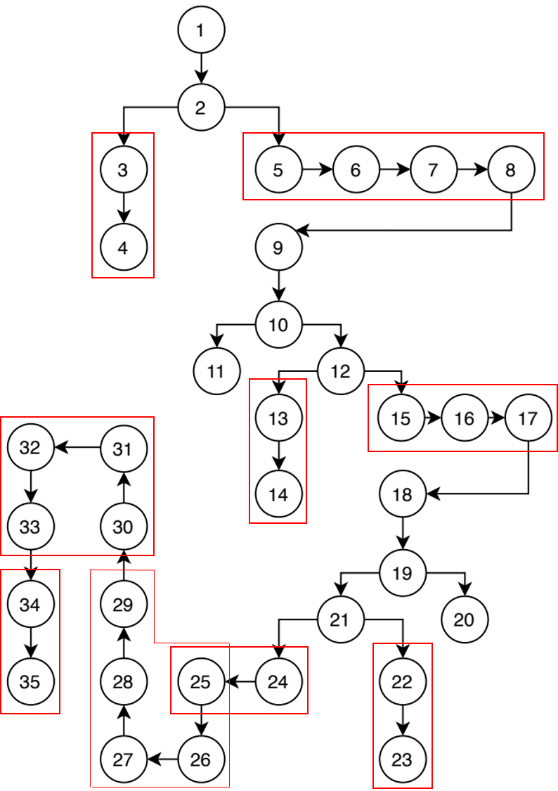


图2.4.1 SellInfo.Query的原始程序图和简化程序图

1. 统计变量的定义/使用节点

在对程序图进行简化和绘制后，我们可以对SellInfo.Query中的所有变量的定义/使用节点进行统计，具体可以见表2.4.1。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量 | 定义节点  原始程序图 | 定义节点  简化程序图 | 使用节点  原始程序图 | 使用节点  简化程序图 |
| ctx | 1 | 1 | - | - |
| req | 1 | 1 | 2, 7 | 2, 4 |
| rsp | 1, 25, 26, 27, 28, 29, 30, 31, 32, 33 | 1, 16, 17 | - | - |
| info | 6 | 4 | 9, 16, 25, 26, 27, 28, 29 | 5, 10, 16 |
| err | 9, 18 | 5, 11 | 10, 12, 13, 19, 21, 22 | 6, 8, 9, 12, 14, 15 |
| good | 15 | 10 | 18, 30, 31, 32, 33 | 11, 17 |

表2.4.1 SellInfo.Query的变量的定义/使用节点

1. 列出变量的定义-使用路径

在列出所有变量的定义/使用节点后，我们可以依据简化程序图的节点情况，依次列出所有变量的定义-使用路径，并且注明其是否为定义-清除路径，具体可见表2.4.2。同时，因为部分路径存在重合部分，因为我们对路径进行了整理，比如用<p1, 4>来代表p2的路径。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 变量 | 路径 | 完整路径 | 简化路径 | 是否为定义-清除路径 |
| req | p1 | 1, 2 | 1, 2 | Y |
| p2 | 1, 2, 4 | p1, 4 | Y |
| info | p3 | 4, 5 | 4, 5 | Y |
| p4 | 4, 5, 6, 8, 10 | p3, 6, 8, 10 | Y |
| p5 | 4, 5, 6, 8, 10, 11, 12, 14, 16 | p4, 11, 12, 14, 16 | Y |
| err | p6 | 5, 6 | 5, 6 | Y |
| p7 | 5, 6, 8 | P6, 8 | Y |
| p8 | 5, 6, 8, 9 | P7, 9 | Y |
| p9 | 5, 6, 8, 10, 11, 12 | P7, 10, p12 | N |
| p10 | 5, 6, 8, 10, 11, 12, 14 | P9, 14 | N |
| p11 | 5, 6, 8, 10, 11, 12, 14, 15 | P10, 15 | N |
| p12 | 11, 12 | 11, 12 | Y |
| p13 | 11, 12, 14 | p14, 14 | Y |
| p14 | 11, 12, 14, 15 | p15, 15 | Y |
| good | p15 | 10, 11 | 10, 11 | Y |
| p16 | 10, 11, 12, 14, 16, 17 | P15, 12, 14, 16, 17 | Y |

表2.4.2 SellInfo.Query的变量的定义-使用路径

1. 选取测试覆盖指标，生成测试用例

在这一步，我们选用**全使用准则**作为测试覆盖指标，即选择每一个定义节点到所有使用节点以及其后续节点的定义-清除路径。而对于所有的定义-使用路径，依照表2.4.2中的简化路径可知，我们可以用尽可能少的测试用例去覆盖尽可能多的目标路径，比如对于路径p8, p9, p10，我们只需要生成一个可以覆盖p10的测试用例即可。而具体的生成的测试用例，我们放在附件中予以展示，此处略去。

1. 编写代码，进行测试

步骤1~4就是基于数据流的测试用例生成过程，接下来我们可以编写代码，进行测试。值得一提的是，因为对于数据流测试的测试覆盖指标本身就是我们测试用例生成的依据，因此按照全使用准则，我们的覆盖率必然是100%，但是在其他覆盖率（比如条件覆盖率、分支覆盖率等）语境下，覆盖率可能会有所下降。

结束了对数据流测试的用例生成过程的介绍，接下来是实践过程中，我们对于数据流测试的感想：

相比于基于路径的测试方法，数据流测试方法的实践过程更加繁琐，一旦代码量增加，或者是出现循环、变量多处定义/使用，我们就很难用手工的方式来实现。而书上也提到，目前很少有针对数据流测试的商业软件，因此这种白盒测试方法似乎显得有点鸡肋。我们认为，基于数据流的测试方法应该是作为辅助的角色，帮助基于路径的测试方法，提供多样化的测试用例，提高测试覆盖率和效果。

最明显的例子就是SellInfo.Find，这个函数中针对变量req有连续10个独立的分支判断，导致总的定义-使用路径达到了条！虽然可以用归纳的方法和少量的用例将这些高度重合的路径全部覆盖，但是的确很难直接手写出所有的定义-使用路径。



# 测试结果概述

本次测试用例的通过率为100.0%，因为本次白盒测试与上一轮的黑盒测试的测试目标是一致的，两次测试过程具备连续性。而上次黑盒测试发现的三个bug均已进行了修复（具体可见之前黑盒测试的测试报告），因此这一次白盒测试的用例通过率达到100.0%是一件可以预料的事情。

本次测试的类覆盖率为100%，总体代码覆盖率为100%，每一个模块各自的代码覆盖率可见表3.1。

本次白盒测试共采纳样例94个，每一个模块各自生成的测试用例数量可见表3.1。相比于黑盒测试的115个测试用例，白盒测试以相对较少的用例数量，达到相同的测试覆盖率。

黑盒与白盒方法生成的用例数量有些许差异，但究竟哪一种方法存在冗余主要取决于具体的测试方法。比如基于决策表的黑盒测试方法可以较好的降低用例的冗余性，基于等价类或边界值的黑盒测试方法则存在大量冗余用例，而白盒测试方法则相对平衡。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模块 | 黑盒/白盒 | 测试方法 | 用例数量 | 代码覆盖率 |
| Content | 黑盒 | 强健壮等价类 | 46 | 100% |
| 白盒 | 基于DD路径 | 44 | 100% |
| Message | 黑盒 | 基于决策表 | 19 | 100% |
| 白盒 | 基于McCabe | 21 | 100% |
| SellInfo | 黑盒 | 强健壮等价类+边界值 | 42 | 100% |
| 白盒 | 基于数据流 | 18 | 100% |
| File | 黑盒 | 强健壮等价类 | 8 | 100% |
| 白盒 | 基于数据流 | 11 | 100% |

表3.1 黑盒/白盒测试的结果对比

# 测试结果展示图

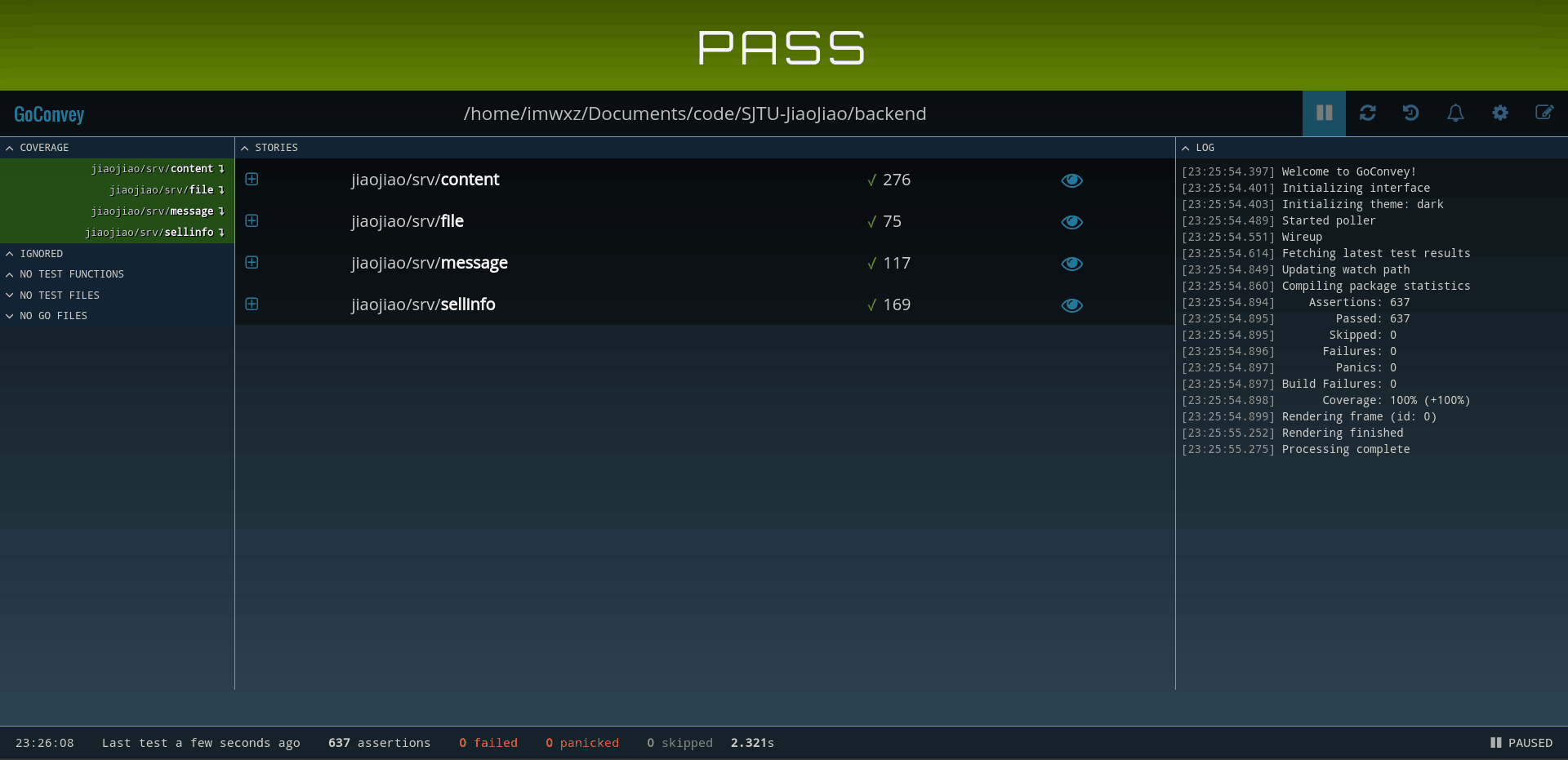


图4.1 白盒测试结果展示图