软件测试与修复——任务单 app

1 单元测试

1.1 测试环境

在 Intellij IDEA 中使用 JUnit（测试框架）和 Mockito（构造桩模块）进行测试。
版本：

- IDEA 2024.2.3
- JUnit 5.10.2
- Mockito 5.14.2

1.2 测试流程

1. 在项目结构中导入库 Mavern: org.junit.jupiter:junit-jupiter:5.10.2
2. 在项目结构中导入库 Mavern: org.mockito:mockito-core:5.14.2
3. 构造测试源文件夹，当前文件结构如下：

```
.
|
|- src
    |
    |- main - java (源文件夹)
    |
    |- test - java (测试源文件夹)
```

源文件夹和测试源文件夹中的包一一对应

4. 在源文件夹中进行操作：

选择要测试的类 > 显示上下文操作 > 创建测试 > 勾选要测试的方法

IDE 自动在测试源文件夹中的相应位置构造测试类

5. 实现测试函数
6. 运行后没有报错，选择“使用覆盖率运行”来得到覆盖率报告

> 使用覆盖率运行时，如果 Windows 用户名不是 ASCII 码会报错。
>
> 原因：IDEA 会在默认路径 C:/Users/<user-name>/AppData 中生成覆盖率报告，而相关代码仅支持 ASCII 码。
>
> 解决方式：（参考自Errors occur when run coverage test）
>
> 编辑自定义虚拟机选项 > 插入“-Djava.io.tmpdir=<path-only-with-ASCII-code>”

1.3 测试步骤

1.3.1 OnetimeTask

针对类 OnetimeTask 混合了白盒测试和黑盒测试，以下是其中一个主要函数的测试说明：

- 白盒测试：判定覆盖

  源代码：

```
@Override
public List<LocalDateTime> getAlarmTimes() {
    if (getState() == TaskState.COMPLETED) {
        alarmTimes.clear();
    }
    else {
        alarmTimes.removeIf(new Predicate<LocalDateTime>() {
            @Override
            public boolean test(LocalDateTime localDateTime) {
                return !localDateTime.isAfter(LocalDateTime.now());
            }
        });
    }
    return new ArrayList<>(alarmTimes);
}
```

  判定覆盖测试

```
// test branches
task.setState(TaskState.ONGOING);
assertEquals(5, task.getAlarmTimes().size());
for (int i = 0; i < 5; i ++)
    assertEquals(now.plusDays(1).minusMinutes(5 - i), task.
       getAlarmTimes().get(i));

task.setState(TaskState.COMPLETED);
assertEquals(0, task.getAlarmTimes().size());

task.setState(TaskState.TODO);
assertEquals(0, task.getAlarmTimes().size());

// test removeIf
for (int i = 1; i <= 5; i ++) {
    task.addAlarmTime(now.minusMinutes(i));
    task.addAlarmTime(now.plusMinutes(i));
    assertEquals(i, task.getAlarmTimes().size());
    for (int j = 0; j < i; j ++)
        assertEquals(now.plusMinutes(j + 1), task.getAlarmTimes().
           get(j));
```

```
}
```

- 黑盒测试

  需求：

  – getAlarmTimes() 的输出按时间先后排序

  测试：

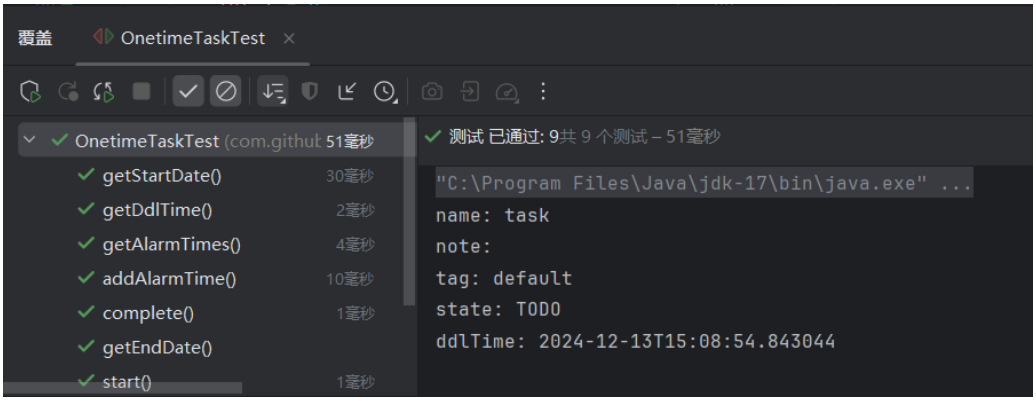
```
private static boolean checkOrderliness(List<LocalDateTime> times) {
    for (int i = 1; i < times.size(); i++) {
        if (!times.get(i).isAfter(times.get(i - 1))) {
            return false;
        }
    }
    return true;
}

// test orderliness
task.addAlarmTime(Duration.ofMinutes(4)); assert(checkOrderliness(
   task.getAlarmTimes()));
task.addAlarmTime(Duration.ofMinutes(1)); assert(checkOrderliness(
   task.getAlarmTimes()));
task.addAlarmTime(Duration.ofMinutes(3)); assert(checkOrderliness(
   task.getAlarmTimes()));
task.addAlarmTime(Duration.ofMinutes(5)); assert(checkOrderliness(
   task.getAlarmTimes()));
task.addAlarmTime(Duration.ofMinutes(2)); assert(checkOrderliness(
   task.getAlarmTimes()));

// test output
assertEquals(5, task.getAlarmTimes().size());
for (int i = 0; i < 5; i ++)
    assertEquals(now.plusDays(1).minusMinutes(5 - i), task.
       getAlarmTimes().get(i));
```

Mockito

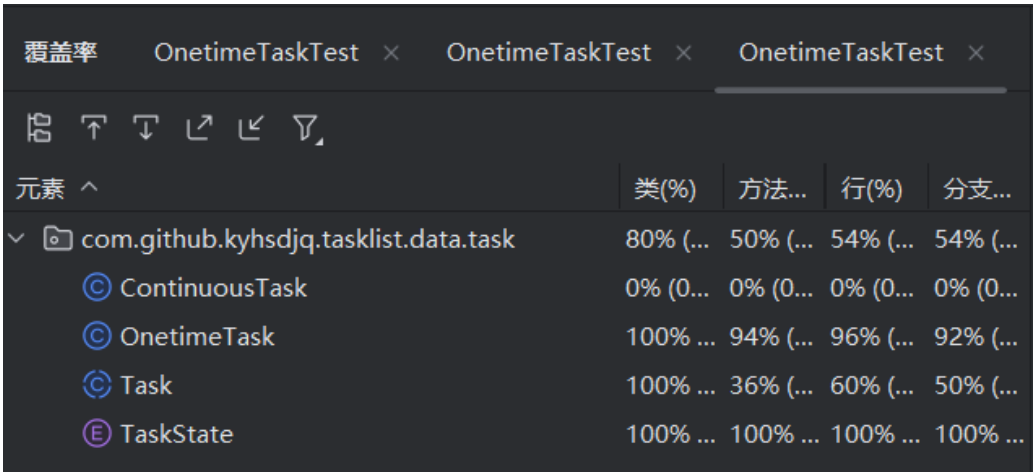
特别地，当 alarmTime 改变时，Task 需要向 TaskPond 同步信息。
为了测试绑定了 TaskPond 的情况，我们需要构建桩模块来模拟 TaskPond。
通过Mockito来模拟 TaskPond 并检测同步函数是否被调用。
例如在 complete 函数的测试中，setState 和 complete 都会调用一次同步函数，通过 verify 语句来确认调用了两次：

```
TaskPond taskPond = mock(TaskPond.class);
task.setTaskPond(taskPond);
task.setState(TaskState.ONGOING);
task.complete();
assertEquals(TaskState.COMPLETED, task.state);
assertFalse(task.complete());
verify(taskPond, times(2)).sycTask(task);
```

测试结果：



测试的对象是 OnetimeTask 类，其中方法覆盖率 94%，行覆盖率 96%，分支覆盖率 92%。



1.3.2 ContinuousTask

对 ContinuousTask 应用条件覆盖测试，以方法 getState 为例：
源代码：

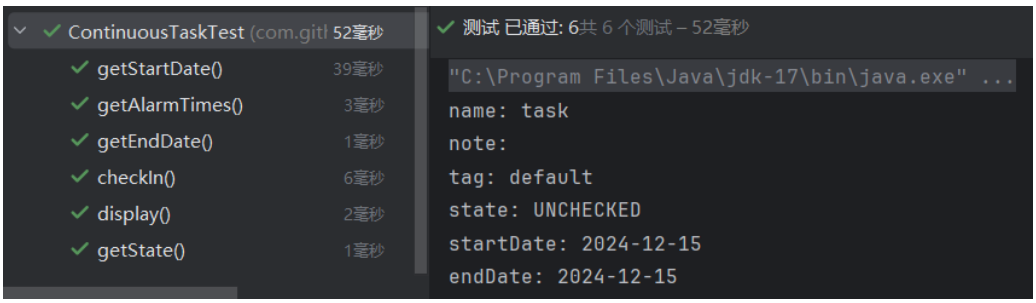
```
private boolean updateState() {
```

```
        boolean result = updateNextAlarmTime();
        TaskState prevState = state;
        if (nextAlarmTime.toLocalDate().isAfter(endDate))
            state = TaskState.COMPLETED;
        else if (nextAlarmTime.toLocalDate().isAfter(LocalDate.now()))
            state = TaskState.CHECKED;
        else
            state = TaskState.UNCHECKED;
        return !prevState.equals(state) || result;
    }

    @Override
    public TaskState getState() {
        updateState();
        return super.getState();
    }
```

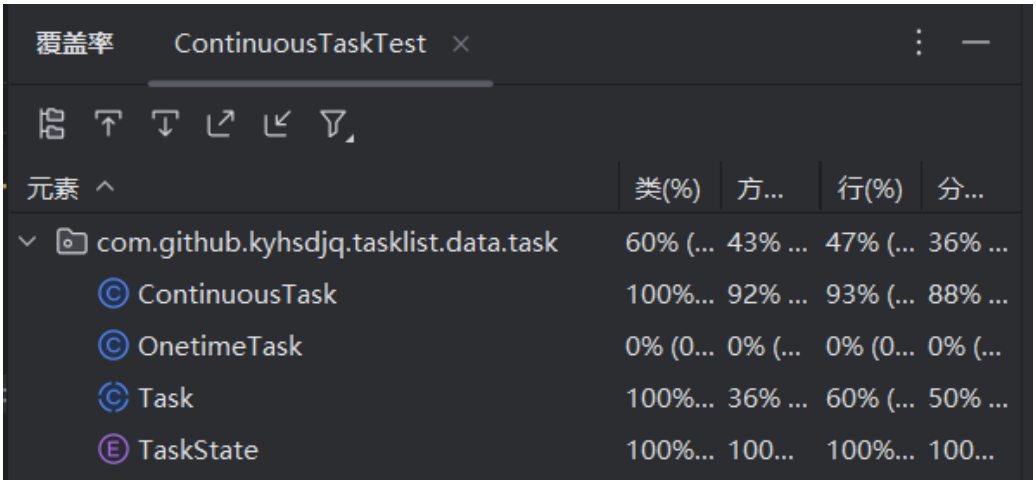
条件覆盖测试：

```
    @Test
    void getState() {
        LocalDateTime now = LocalDateTime.now();
        task.setEndDate(now.toLocalDate().plusDays(5));

        // COMPLETED
        task.setNextAlarmTime(now.plusDays(6));
        assertEquals(TaskState.COMPLETED, task.getState());

        // CHECKED
        task.setNextAlarmTime(now.minusMinutes(1));
        assertEquals(TaskState.CHECKED, task.getState());
        task.setNextAlarmTime(now.plusDays(3));
        assertEquals(TaskState.CHECKED, task.getState());

        // UNCHECKED
        task.setNextAlarmTime(now.plusMinutes(1));
        assertEquals(TaskState.UNCHECKED, task.getState());
    }
```

测试结果：



测试对象为 ContinuousTask 类，其中方法覆盖率 92%，行覆盖率 93%，分支覆盖率 88%。



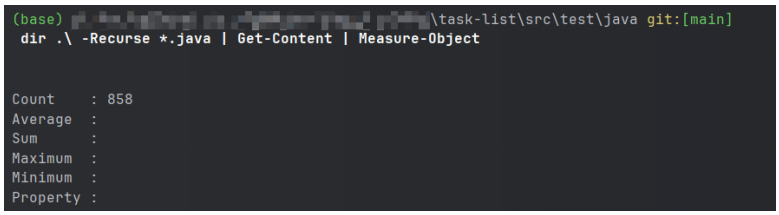
2 集成测试

2.1 测试环境

使用 JUnit 测试

2.2 测试规模

共 8 个类 858 行



2.3 测试步骤

自底向上进行集成测试，分为两组：

- 模块一：data
- 模块二：ui

2.4 模块一：data

data 中包含以下类：

- OnetimeTask
- ContinuousTask
- Task
- TaskState
- AlarmSystem
- TaskPond

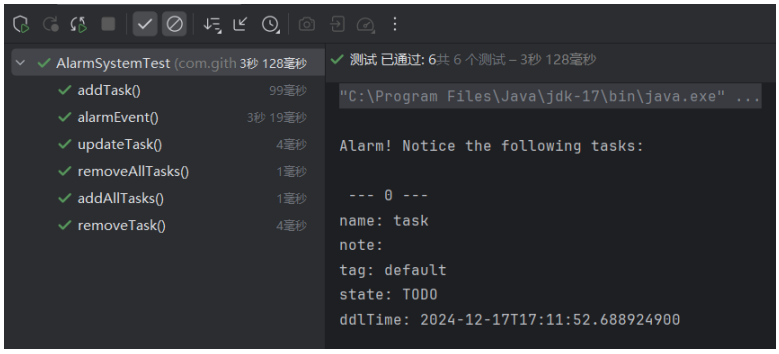
依赖路径如下：

```
TaskPond
    |
    |- AlarmSystem
            |
            |- Task
                |
                |- OnetimeTask
                |
                |- ContinuousTask
                |
                |- TaskState
```

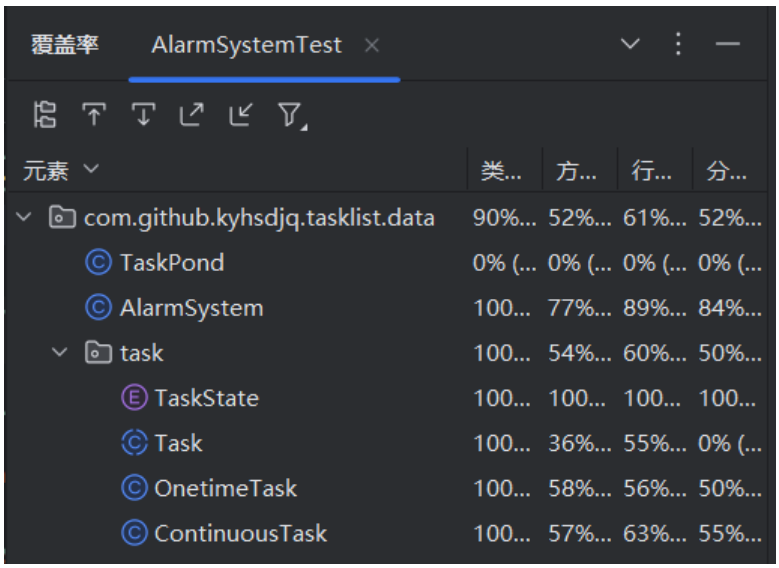
其中 OnetimeTask 和 ContinuousTask 已通过单元测试，将它们作为桩模块，接下来按序测试 AlarmSystem 和 TaskPond。

2.4.1 AlarmSystem

对 AlarmSystem 进行条件覆盖测试。测试结果：

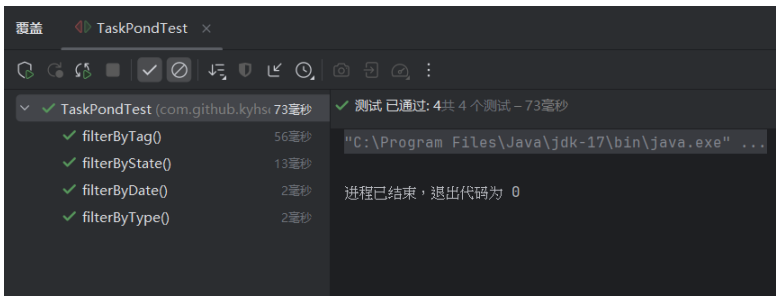


其中 AlarmSystem 方法覆盖率 77%，行覆盖率 89%，分支覆盖率 84%。

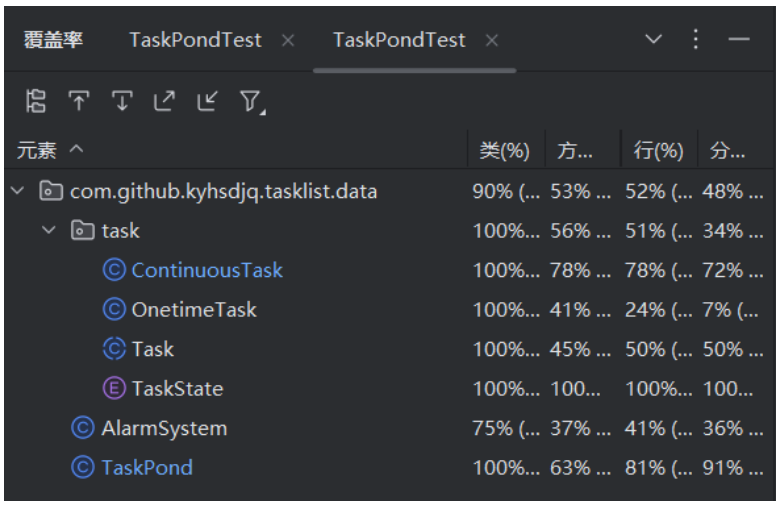


2.4.2 TaskPond

对类 TaskPond 进行条件覆盖测试。测试结果：

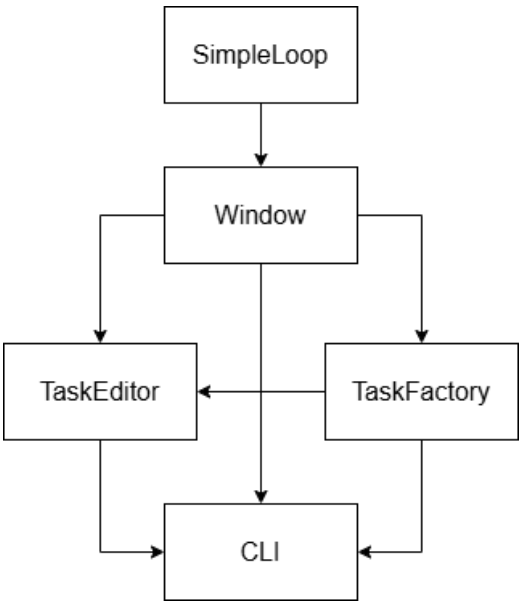


其中 TaskPond 方法覆盖率 63%，行覆盖率 81%，分支覆盖率 91%。



2.5 模块二：ui

依赖图如下：



测试顺序：

1. CLI
2. TaskEditor
3. TaskFactory
4. Window
5. SimpleLoop

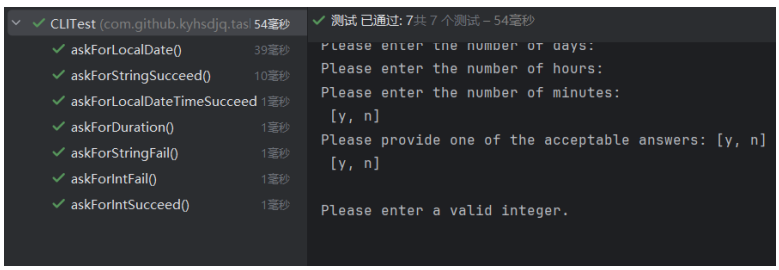
2.5.1 CLI

模拟用户对命令行的输入：

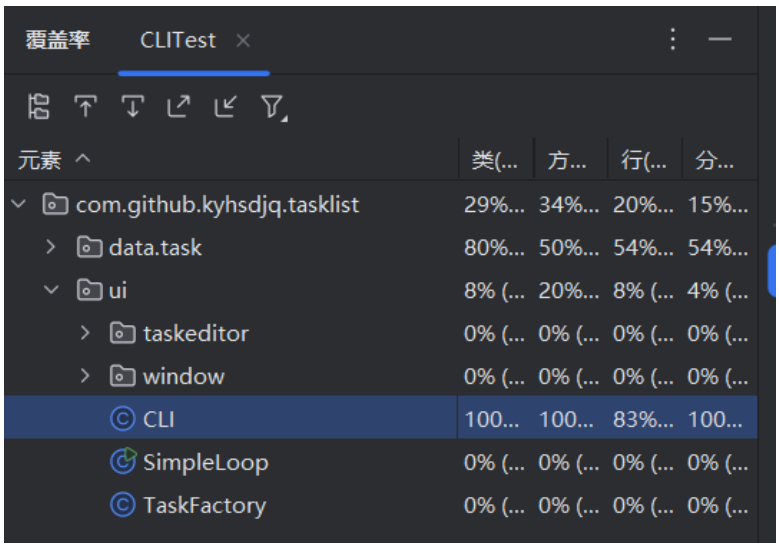
```
String simulatedInput = "Hello, World!\n";
ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(
    simulatedInput.getBytes());
System.setIn(byteArrayInputStream);
```

> 这样模拟的输入只有第一个 scanner 可以读到信息 (通过 scanner.nextLine().trim())
> 所以对 CLI 中的静态函数进行了重构：增加一个参数 scanner。这样可以保证每次输入的 scanner 相同。

对类 CLI 进行条件覆盖测试。测试结果：

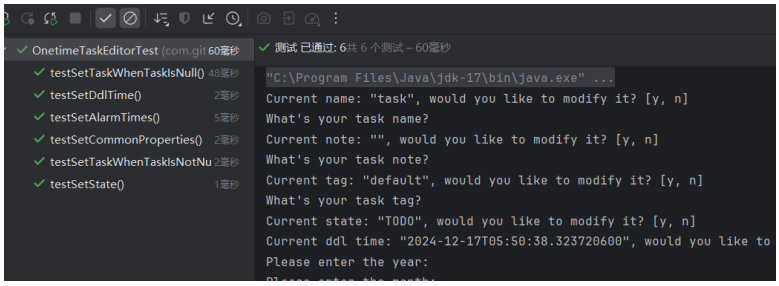


其中 CLI 方法覆盖率 100%，行覆盖率 83%，分支覆盖率 100%。

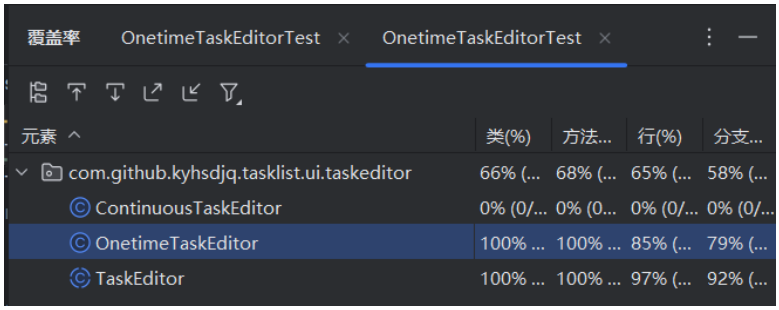


2.5.2 OnetimeTaskEditor

对类 OnetimeTaskEditor 进行路径覆盖测试。测试结果：

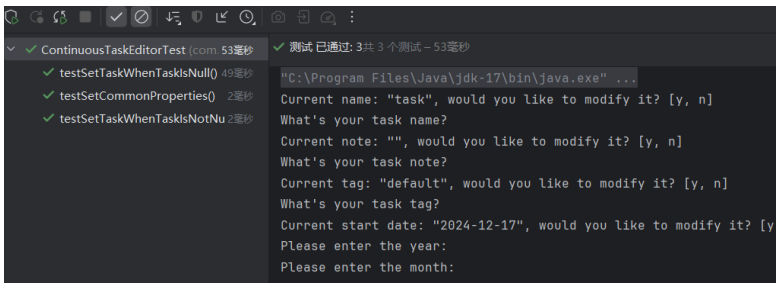


其中 OnetimeTaskEditor 方法覆盖率 100%，行覆盖率 85%，分支覆盖率 79%。

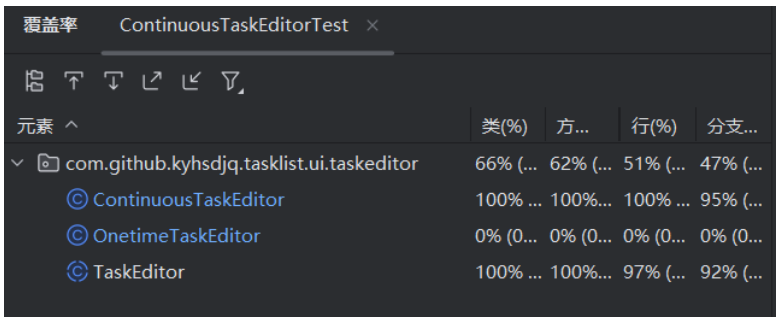


2.5.3 ContinuousTaskEditor

对类 ContinuousTaskEditor 进行路径覆盖测试。测试结果：

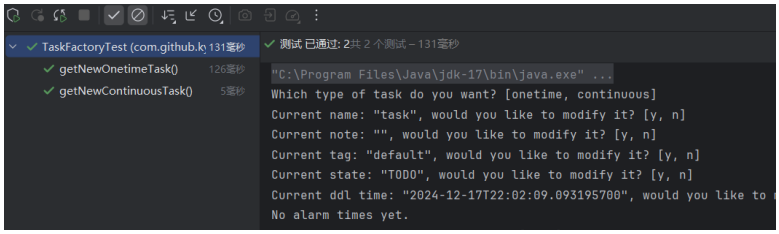


其中 ContinuousTaskEditor 方法覆盖率 100%，行覆盖率 100%，分支覆盖率 95%。

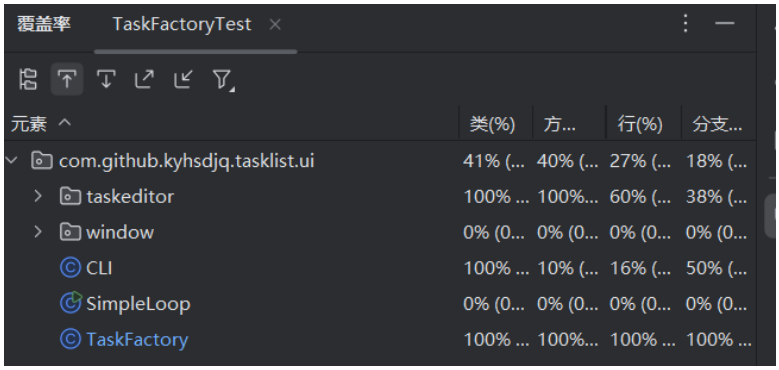


2.5.4 TaskFactory

对类 TaskFactory 进行条件覆盖测试。测试结果：



其中 TaskFactory 方法覆盖率 100%，行覆盖率 100%，分支覆盖率 100%。



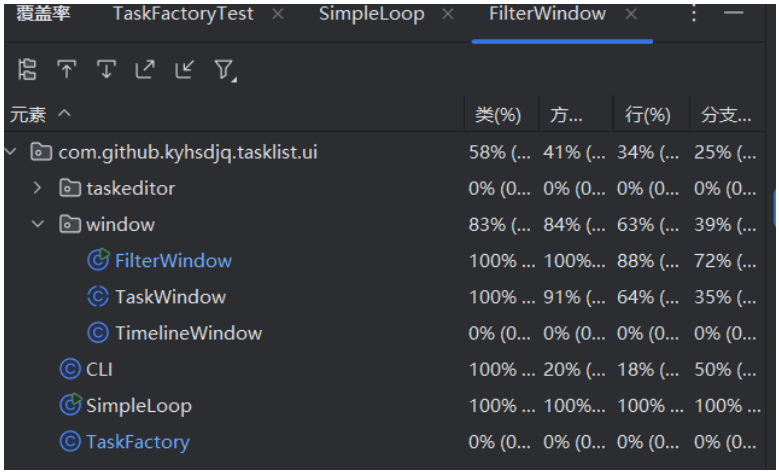
2.5.5 Window & SimpleLoop

在这两个类中，需要模拟的用户输入过于复杂，直接通过命令行对它们进行测试。
（因为 JUnit 测试时命令行是只读的，这些测试作为 main 函数插入在源代码中）

- 对类 FilterWindow 进行条件覆盖测试。测试结果（一共六页，仅展示其中最具代表性的）：

```
 --- 0 ---
name: task
note:
tag: default
state: TODO
ddlTime: 2024-12-17T22:23:44.267770

No continuous tasks yet.

Choose an action: [filter, add, remove, edit, exit]
filter
Which attribute would you like to filter by? [date, tag, type, state, none]
date
Please enter the year:
2024
Please enter the month:
12
Please enter the day:
17

Onetime tasks:

 --- 0 ---
name: task
note:
tag: default
state: TODO
ddlTime: 2024-12-17T22:23:44.267770
```

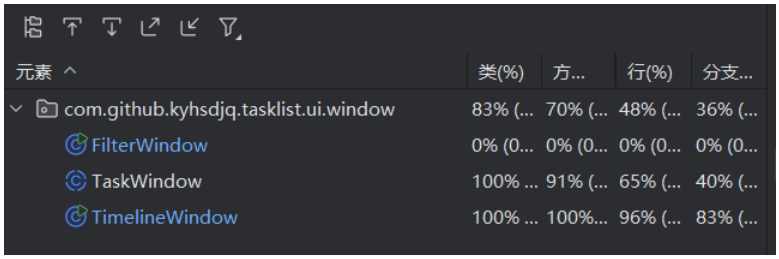
其中 FilterWindow 方法覆盖率 100%，行覆盖率 88%，分支覆盖率 72%。



- 对类 TimelineWindow 进行条件覆盖测试。测试结果（一共六页，仅展示其中最具代表性的）：

```
 --- 0 ---
name: task
note:
tag: default
state: TODO
ddlTime: 2024-12-17T22:34:04.649650500

No continuous tasks yet.

Choose an action: [date, add, remove, edit, exit]
date
Please enter the year:
2024
Please enter the month:
12
Please enter the day:
17

Date: 2024-12-17

Onetime tasks:

 --- 0 ---
name: task
```

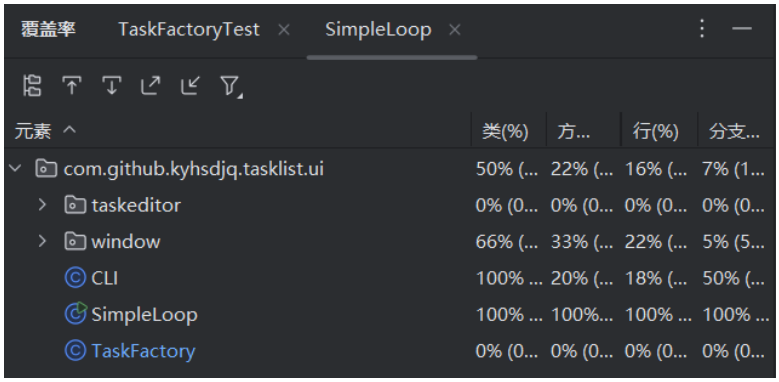
其中 TimelineWindow 方法覆盖率 100%，行覆盖率 96%，分支覆盖率 83%。



- 对类 SimpleLoop 进行条件覆盖测试。测试结果：

```
"C:\Program Files\Java\jdk-17\bin\java.exe" ...
Welcome to task list!
Choose a window: [filter, timeline, exit]
filter

No tasks yet.

Choose an action: [filter, add, remove, edit, exit]
exit
Choose a window: [filter, timeline, exit]
timeline

Date: 2024-12-17

No tasks yet.

Choose an action: [date, add, remove, edit, exit]
exit
Choose a window: [filter, timeline, exit]
exit
Good bye.

进程已结束，退出代码为 0
```

其中 SimpleLoop 方法覆盖率 100%，行覆盖率 100%，分支覆盖率 100%。



3 模糊测试

3.1 测试环境

选择Jazzer作为模糊测试工具，原因如下：

- Jazzer 专门针对 Java 项目进行模糊分析
  （而 AFL++ 本身只针对 C/C++ 项目，需要添加中间层才能分析 Java 项目）
- Jazzer 提供的平台多样，包含 Junit, GitHub releases, Bazel, OSS-Fuzz。
- 恰好单元测试的时候使用了 JUnit，现在使用 Jazzer 建立在 JUnit 上的平台。

安装过程：

- 在项目结构中导入库 Mavern:org.mockito:mockito-core:RELEASE 即可。

3.2 测试流程

1. 和单元测试相似，设计模糊测试方法。一个对照的例子：

```
class ParserTests {
    @Test
    void unitTest() {
        assertEquals("foobar", SomeScheme.decode(SomeScheme.encode("
            foobar")));
    }

    @FuzzTest
    void fuzzTest(FuzzedDataProvider data) {
        String input = data.consumeRemainingAsString();
        assertEquals(input, SomeScheme.decode(SomeScheme.encode(input)))
            ;
    }
}
```

2. 将环境变量 JAZZER_FUZZ 设为 1，进行模糊测试

3. 出现错误后，将环境变量 JAZZER 删除，进行回归测试（通过.cifuzz-corpus 中的崩溃结果）

项目路径

Jazzer 需要确保项目路径不包含中文，可以通过移动项目本身修改。

临时文件路径

Jazzer 需要确保临时文件路径不包含中文。临时文件放在 AppData 文件夹，如果 Windows 用户名为中文会出问题。

查看 Jazzer 源代码，发现它通过 Files.createTempDirectory("jazzer-java-seeds") 建立临时文件。

解决方式为设置全局变量：

3.3 测试步骤

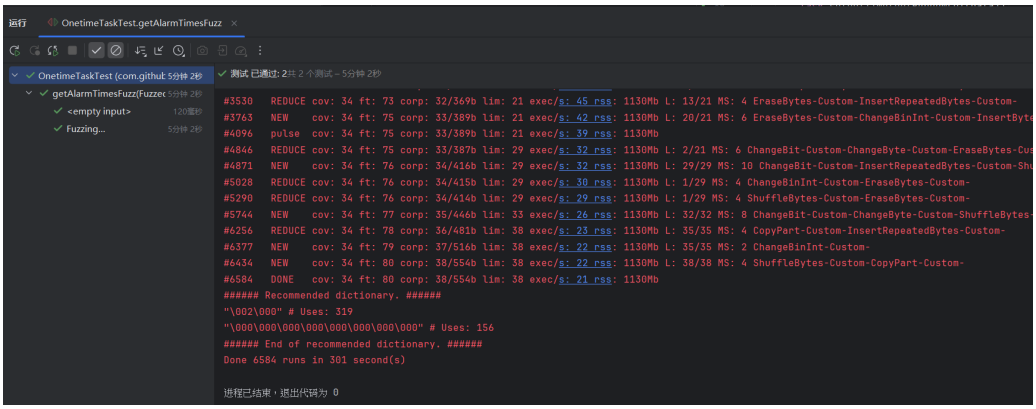
3.3.1 getAlarmTimesFuzz

以其中检测得到 alarm times 是否有序的模糊测试为例

```
private static LocalDateTime getRandomLDT(FuzzedDataProvider data) {
    while (true) {
        try {
            int year = data.consumeInt(-999999999, 999999999);
            int month = data.consumeInt(1, 12);
            int day = data.consumeInt(1, 31);
            int hour = data.consumeInt(0, 23);
            int minute = data.consumeInt(0, 59);

            return LocalDateTime.of(year, month, day, hour, minute);
        } catch (DateTimeException e) {
            continue;
        }
    }
}

@FuzzTest
void getAlarmTimesFuzz(FuzzedDataProvider data) {
    task.setDdlTime(getRandomLDT(data));
    for (int i = 0; i < 1000; i ++) {
        task.addAlarmTime(getRandomLDT(data));
```

```
            assert(checkOrderliness(task.getAlarmTimes()));
        }
    }
```

测试结果：



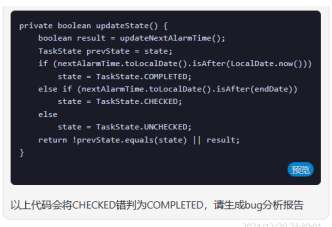
4 软件修复

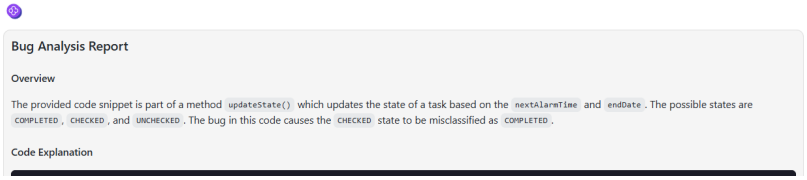
4.1 ContinuousTask 的 state

该问题在单元测试（黑盒测试）中发现，报错为 ContinuousTask 的 stateType 与期待的不同。问题定位到 updateState 方法，错误代码：

```
private boolean updateState() {
    boolean result = updateNextAlarmTime();
    TaskState prevState = state;
    if (nextAlarmTime.toLocalDate().isAfter(LocalDate.now()))
        state = TaskState.COMPLETED;
    else if (nextAlarmTime.toLocalDate().isAfter(endDate))
        state = TaskState.CHECKED;
    else
        state = TaskState.UNCHECKED;
    return !prevState.equals(state) || result;
}
```

询问 gpt 结果：





Suggested Fix

To fix this bug, the conditions should be reordered to ensure that `CHECKED` is evaluated before `COMPLETED`. This ensures that if `nextAlarmTime` is after `endDate`, it is marked as `CHECKED`, regardless of whether it is also after the current date.

Corrected Code

```
private boolean updateState() {
    boolean result = updateNextAlarmTime();
    TaskState prevState = state;
    if (nextAlarmTime.toLocalDate().isAfter(endDate))
        state = TaskState.CHECKED;
    else if (nextAlarmTime.toLocalDate().isAfter(LocalDate.now()))
        state = TaskState.COMPLETED;
    else
        state = TaskState.UNCHECKED;
    return !prevState.equals(state) || result;
}
```

Explanation of Fix

- **Reordered Conditions:** The condition to check if `nextAlarmTime` is after `endDate` is now placed before the condition that checks if `nextAlarmTime` is after the current date. This ensures that tasks are correctly marked as `CHECKED` before considering them for `COMPLETED`.

Conclusion

By reordering the conditions, the logic now correctly prioritizes marking tasks as `CHECKED` when `nextAlarmTime` is after `endDate`, preventing misclassification as `COMPLETED`. This fix ensures the states are updated accurately based on the intended logic.

2024/12/20 23:30:01

bug 在于条件判断顺序反了，修改后代码：

```
private boolean updateState() {
    boolean result = updateNextAlarmTime();
    TaskState prevState = state;
    if (nextAlarmTime.toLocalDate().isAfter(endDate))
        state = TaskState.COMPLETED;
    else if (nextAlarmTime.toLocalDate().isAfter(LocalDate.now()))
        state = TaskState.CHECKED;
    else
        state = TaskState.UNCHECKED;
    return !prevState.equals(state) || result;
}
```

4.2 TaskPond 和 Task 死循环

问题出现在集成测试中，报错为栈溢出，可以观察到 removeTask 方法和 setTaskPond 方法互相调用，产生死循环。

```
public boolean removeTask(Task task) {
    boolean result = tasks.remove(task);
    alarmSystem.removeTask(task);
    task.setTaskPond(null);
    return result;
}

public void setTaskPond(TaskPond taskPond) {
    if (this.taskPond != null) {
        this.taskPond.removeTask(this);
    }
    this.taskPond = taskPond;
}
```

gpt 解答：

Corrected Code

```
public boolean removeTask(Task task) {
    boolean result = tasks.remove(task);
    alarmSystem.removeTask(task);
    // Avoid calling setTaskPond if it's already null to prevent recursion
    if (task.getTaskPond() != null) {
        task.setTaskPond(null);
    }
    return result;
}

public void setTaskPond(TaskPond taskPond) {
    if (this.taskPond != null && this.taskPond != taskPond) {
        this.taskPond.removeTask(this);
    }
    this.taskPond = taskPond;
}
```

Explanation of Fix

- **Conditional Invocation:** In `removeTask`, the call to `setTaskPond(null)` is guarded by a check to ensure that it is only called if the `taskPond` is not already `null`. This prevents unnecessary recursion.
- **Avoid Redundant Calls:** In `setTaskPond`, an additional check ensures that `removeTask` is only called if the current `taskPond` is different from the new `taskPond`, thus avoiding redundant calls and potential recursion.

Conclusion

By introducing checks to prevent unnecessary and redundant method calls, the infinite recursion is eliminated, resolving the stack overflow issue. This ensures that tasks can be removed and reassigned without causing the application to crash.

事实上只要在 setTaskPond 中删除原先 TaskPond 的时候确认要设置的 TaskPond 不为空（虽然 gpt 给的解决方式更严密一些，但我们认为 TaskPond 包含的 Task 肯定已经被设置好了 TaskPond）。

```
public boolean removeTask(Task task) {
    boolean result = tasks.remove(task);
    alarmSystem.removeTask(task);
    task.setTaskPond(null);
    return result;
}

public void setTaskPond(TaskPond taskPond) {
    if (this.taskPond != null && taskPond != null) {
        this.taskPond.removeTask(this);
    }
    this.taskPond = taskPond;
}
```

4.3 CLI

对于 CLI 的所有输入在命令行交互中可以正常表现，但是通过程序注入输入时必须使用同一个 scanner。

经过 copilot 提醒后对所有有关 CLI 的类进行了重构，在对象中存储要使用的同一个 scanner。

5 软件运行方式

运行环境：命令行
运行方式：
软件会不断在命令行中抛出询问，并且在中括号中提示用户可以回答的结果。
如果结果不符合期待的输入，软件会再次提醒输入要求，然后重新要求用户输入。
一开始可以选择 filterWindow 或 timelineWindow，或是退出软件。
在两个窗口中每步操作后都会显示当前界面上的任务，可选的操作：

- 在所有任务中做出筛选

- 在已筛选的任务中根据下标选择任务进行编辑或删除
- 创建新的任务（软件会一项项询问想要创建的任务的属性）
- 退出该窗口