软件设计实践

单元测试

谢涛　　马郓

软件测试概述

单元测试背景

Boost单元测试框架

- https://www.jetbrains.com/help/clion/unit-testing-tutorial.html#boost-test-framework
- https://www.boost.org/doc/libs/1_82_0/more/getting_started/index.html
- https://www.boost.org/doc/libs/1_82_0/libs/test/doc/html/boost_test/intro.html

什么是单元测试用例?

单元测试用例：测试输入+测试断言

```
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_CASE(testHashSetAddTwoElems)
void testHashSetAddTwoElems()
{
    HashSet<int> set;
    set.add(3);
    set.add(14);
    BOOST_TEST(set.size() == 2);
}
```

很多开发者手写这些测试用例，包含

- 确定下有意义的函数调用序列,
- 选择有代表性的参数值 (测试输入),
- 表达断言 (测试预言)

单元测试: 好处

- 设计和规约
  - 样例行为
- 代码覆盖和回归测试
  - 正确性的信心
  - 维持同样的行为
- 短反馈环
  - *单元*测试用例测试*少量代码*
  - 失败更容易调试
- 文档

作为规约的测试用例

- 测试用例展示如何使用系统

- 测试用例需要可读性强
  - 需要注释/命名来描述其目的
  - 保持简短，删除重复或冗余的测试用例

```
BOOST_AUTO_TEST_CASE(testHashSetAddTwoElems)
void testHashSetAddTwoElems()
{
    HashSet<int> set;
    set.add(3);
    set.add(14);
    BOOST_TEST(set.size() == 2);
}
```

单元测试: 度量质量

- 覆盖率: 所有的程序部分都测到了吗?
  - 语句行
  - 基本块
  - 分支
  - ……
- 断言: 程序做对的事情?
  - 测试预言

经验:

- 只是高覆盖率或大量断言并不是高质量的indicator。
- 只有两者兼有才是!

单元测试框架

- 开发者测试框架/工具比如Boost, JUnit
- 每个代码单元需要几个测试用例

- 自动化测试!
- 用于单元测试, 也可用于集成测试和功能测试
- 回归测试

软件测试概述

数据驱动测试

https://www.boost.org/doc/libs/1_82_0/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation.html

带参数单元测试用例

例子：

```
#include <boost/test/unit_test.hpp>
#include <vector>

BOOST_DATA_TEST_CASE(my_test_case, std::vector<int>{1, 2, 3}, data)
{
    BOOST_TEST(data > 0);
}
```

不带参数单元测试用例例子：

```
BOOST_AUTO_TEST_CASE(testHashSetAddTwoElems)
void testHashSetAddTwoElems()
{
    HashSet<int> set;
    set.add(3);
    set.add(14);
    BOOST_TEST(set.size() == 2);
}
```

带参数单元测试用例

例子:

```
#include <boost/test/unit_test.hpp>
#include <vector>

BOOST_DATA_TEST_CASE(my_test_case, std::vector<int>{1, 2, 3}, data)
{
    BOOST_TEST(data > 0);
}
```

等同于JUnit的（之后胶片将采用如下简练易读方式）：

```
@ParameterizedTest
@ValueSource(1， 2， 3)
void my_test_case(int data) {
{
    BOOST_TEST(data > 0);
}
```

为带参数单元测试用例提供数据

例子：

```
#include <boost/test/unit_test.hpp>
#include <vector>

BOOST_DATA_TEST_CASE(my_test_case, std::vector<int>{1, 2, 3}, data)
{
    BOOST_TEST(data > 0);
}
```

1. the notion of **dataset** and **sample** is introduced
2. the declaration and registration of the data-driven test cases are explained,
3. the *operations* on datasets are detailed
4. and finally the built-in dataset generators are introduced.

See https://www.boost.org/doc/libs/1_82_0/libs/test/doc/html/boost_test/tests_organization/test_cases/test_case_generation.html

带参数单元测试用例 - Assumption

例子：

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<Integer> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST((oldSize + 1) == stk.size());
}
```

assumeTrue（JUnit）：如果假值，跳过测试用例的剩余部分

在Boost得写成如下：

```
if !(stk != null) return;
```

Separation of Concerns

Parameterized tests enable to separate two concerns:
(1) The specification of external behavior (i.e., assertions)
(2) The generation/selection of internal test inputs (i.e., coverage)

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<Integer> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST((oldSize + 1) == stk.size());
}
```

Besides manually writing test inputs, one can use an automatic test generation tool to construct *a small test suite* with *high coverage*!

- **E.g., AgitarOne JUnit Generator for Java programs**
- **E.g., Parasoft Jtest for Java programs**
- **E.g., Microsoft IntelliTest in Visual Studio for .NET programs**

General 4A Pattern: Assume, Arrange, Act, Assert

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<Integer> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST((oldSize + 1) == stk.size());
}
```

(Buggy) Implementation for IntSet

```
class IntSet {
public:
    IntSet(int max_value) { … }
    ~IntSet() { … }
    void insert(int value) { … }
    void remove(int value) { … }
    bool member(int value) const { … }
    bool equals(IntSet o) {…}
};
```

Parameterized Tests as Algebraic Specifications

- **Universally quantified, conditional property**

```
∀ int i, int j, IntSet s:
s != null && i == j  ⇒
remove(insert(s, j).state, i).state = remove(s, i).state
```

```
∀ int i, int j, IntSet s1, IntSet s2:
s != null && i == j && s1.equals(s2) ⇒
remove(insert(s1, j).state, i).state.equals(remove(s2, i).state)
```

First argument is the receiver object
The resulting receiver object state of m as m.state
The return of m as m.ret

Parameterized Tests as Algebraic Specifications

- **Universally quantified, conditional property**

```
∀ int i, int j, IntSet s:
s != null && i == j  ⇒
remove(insert(s, j).state, i).state = remove(s, i).state
```

```
∀ int i, int j, IntSet s1, IntSet s2:
s != null && i == j && s1.equals(s2) ⇒
remove(insert(s1, j).state, i).state.equals(remove(s2, i).state)
```

```
void PUTSet1(int i, int j, IntSet s1, IntSet s2) {
  assumeTrue(s!= null && i == j && s1.equals(s2));
  s1.insert(j);
  s1.remove(i);
  s2.remove(i);
  BOOST_TEST(s1.equals(s2);
}
```

Some Properties for IntSet

(Stotts et al., 2002)

Properties (axioms)

```
∀  int i, int j, IntSet s:
s != null ⇒
member(IntSet().ret, i).ret = false &&
member(insert(s, j).state, i).ret =
      if i = j then true
      else member(s, i).ret
```

Guidelines for Writing Properties

- Property First (Assumption Second)
- Assumption First (Property Second)

Property-First Guideline

- **Step 1. Classify methods to be**
  - modifiers: IntSet constructor, insert, remove while modifying/constructing states
  - observers: returning some other value w/o modifying states
- **Step 2. Pairwise-combine them to formulate properties**
  - Every observer o is paired with a modifier
  - Every modifier m is paired with another modifier (including m itself)
- **Step 3. Add proper assumptions to make the properties (e.g., equality, inequality) true**

Add More Properties

(Stotts et al., 2002)

```
∀  int i, int j, IntSet s:
s != null ⇒
member(IntSet().ret, i).ret = false &&
member(insert(s, j).state, i).ret =
      if i = j then true
      else member(s, i).ret  &&
remove(IntSet().ret, i).state = IntSet().ret &&
remove(insert(s, j), i).state =
       if i = j then remove(s, i).state
       else insert(remove(s, i), j).state  &&
insert(insert(s, i), j) = insert(insert(s, j), i) &&
insert(insert(s, i), i) = insert(s, i)
...
```

Assumption-First Guideline

- Step 1: All-combine possible returns of observers, e.g.,
  - !member(i)
  - member(i)
  - isEmpty()
  - !isEmpty()
  - !member(i) && isEmpty()
  - member(i) && isEmpty()
  - !member(i) && !isEmpty()
  - member(i) && !IsEmpty()
  - true

```
class IntSet {
public:
    IntSet(int max_value) { … }
    ~IntSet() { … }
    void insert(int value) { … }
    void remove(int value) { … }
    bool member(int value) const { … }
    bool isEmpty() { … }
    bool equals(IntSet o) {…}
};
```

- Step 2: Formulate properties ***specific*** under each feasible combination as assumption
  - then follow the property-first guideline under such assumption

References

- Stotts, D., Lindsey, M., & Antley, A. (2002). An informal formal method for systematic JUnit test case generation. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods (XP/Agile Universe '02)*, 131–143. DOI:https://doi.org/10.1007/3-540-45672-4_13

软件测试概述

测试用例泛化

Example Conventional Tests (CTs) for Stack

```
BOOST_AUTO_TEST_CASE(CT1)
void CT1() {
   int elem = 1;
   Stack<int> stk;
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

```
BOOST_AUTO_TEST_CASE(CT2)
void CT2() {
   int elem = 30;
   Stack<int> stk;
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

```
BOOST_AUTO_TEST_CASE(CT3)
void CT3() {
   int elem1 = 1;
   int elem2 = 30;
   Stack<int> stk;
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == 2);
}
```

- CT1 and CT2 exercise ***push*** with different test data
- CT3 exercises ***push*** when stack is not empty

Two main issues with CTs:

- Fault-detection capability issue: undetected fault where things go wrong when passing a negative value to ***push***
- Redundant test issue: CT2 is redundant with respect to CT1

Test Generalization: CTs → Parameterized Test (PT)

(Thummalapenta et al., 2011)

- **Step 1: Parameterize**
- **Step 2: Generalize Test Oracle**
- **Step 3: Add Assumptions**
- **Step 4: Cross-test generalize**

Step 1 - Parameterize

- Promote primitive values as arguments
  - *elem* as a parameter of type int
- Promote non-primitive objects such as receiver objects as arguments
  - *stk* as a parameter of type Stack<Integer>

```
BOOST_AUTO_TEST_CASE(CT1)
void CT1() {
   int elem = 1;
   Stack<int> stk;
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<Integer> stk) {
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

```
BOOST_AUTO_TEST_CASE(CT2)
void CT2() {
   int elem = 30;
   Stack<int> stk;
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT2(int elem, Stack<Integer> stk) {
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

Step 1 - Parameterize

Cont.

- **Promote all primitive values as arguments**
  - ***elem*** **as a parameter of type int**
- **Promote non-primitive objects such as receiver objects as arguments**
  - ***stk*** **as a parameter of type Stack<Integer>**

```
BOOST_AUTO_TEST_CASE(CT3)
void CT3() {
   int elem1 = 1;
   int elem2 = 30;
   Stack<int> stk;
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == 2);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT3(int elem1, int elem2, Stack<Integer> stk) {
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == 2);
}
```

Step 2 – Generalize Test Oracle

- Replace **constant** values in assertions with some generalized expressions
  - e.g., 1 and 2

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<int> stk) {
   stk.push(elem);
   BOOST_TEST(stk.size() == 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<int> stk) {
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST(stk.size() == oldSize + 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT3(int elem1, int elem2, Stack<int> stk) {
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == 2);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT3(int elem1, int elem2, Stack<int> stk) {
   int oldSize = stk.size();
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == oldSize + 2);
}
```

Step 3 – Add Assumptions

- **Add assumptions to prevent illegal values for the parameters or enable the assertions to be valid**
  - E.g., Add `assumeTrue(stk != null)`, i.e., generated value should not be *null*

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<int> stk) {
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST(stk.size() == oldSize + 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<int> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST(stk.size() == oldSize + 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT3(int elem1, int elem2, Stack<int> stk) {
   int oldSize = stk.size();
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == oldSize + 2);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT3(int elem1, int elem2, Stack<int> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == oldSize + 2);
}
```

Step 4 – Cross-test Generalize

- **Generalize further across multiple tests**
  - **E.g., further generalize PT1 and PT3**

```
@ParameterizedTest
@ValueSource(…)
void PT1(int elem, Stack<int> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem);
   BOOST_TEST(stk.size() == oldSize + 1);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT3(int elem1, int elem2, Stack<int> stk) {
   assumeTrue(stk != null);
   int oldSize = stk.size();
   stk.push(elem1);
   stk.push(elem2);
   BOOST_TEST(stk.size() == oldSize + 2);
}
```

```
@ParameterizedTest
@ValueSource(…)
void PT(int[] elem) {
  assumeTrue(elem != null);
  Stack<int> stk;
  for(int i=0; i< sizeof(elem); i++)
      stk.push(elem[i]);
  BOOST_TEST(stk.size() == elem.length);
}
```

References

- Thummalapenta, S., Marri, M., Xie, T., Tillmann, N., de Halleux, J. (2011). Retrofitting unit tests for parameterized unit testing. *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE 2011),* 294-309. DOI: 10.1007/978-3-642-19811-3_21. Retrieved from https://link.springer.com/chapter/10.1007/978-3-642-19811-3_21