**《软件测试》**

**软件测试实验三：等价类分析方法**

**姓 名： 胡国昌 学 号： 202210120518**

**院 系： 计算机与信息学院 专 业：计算机与科学**

**实 验 室： B1-119 实验日期： 2025.3.9**

**总评成绩： 审阅教师： 杨青**

# 一、实验目的：

1. 巩固 git 协作式管理工具的使用；
2. 巩固 gradle 项目构建 Java 应用的基本操作，使用Maven构建[参考案例](https://gitee.com/izxj/stassignment" \o "参考案例)
3. 掌握等价类分析方法设计测试用例的方法

# **实验要求**

# 二、课程目标：

目标2： 能够运用功能测试、基于控制流和数据流的测试等软件测试的核心技术和原理，结合相关文献，对测试问题及其影响因素进行分析或计算，得出有效结论

# 三、实验要求：

1. 实现电话账单收费功能
2. 采用边等价类分析方法设计测试用例，并采用Junit5 编写电话账单收费程序测试用例，并提交到代码仓库
3. 按照[实验报告模板](https://star.jmhui.com.cn/u/cms/www/202203/06160105hi3m.docx) 编写实验报告，以“学号-姓名-软件测试实验三”命名，提交到雨课堂“软件测试实验三”

# 四、实验步骤与内容

## **需求：电话账单计费**

对于电话账单来说，当春季和秋季标准时间与夏令时时间进行转换时会产生一个很有意思的问题: 春季，这种转换发生在(3月某个)星期日凌晨2:00点，这时要将时钟设置为凌晨3:00点；秋季， 转换通常在11月的第一个星期日，时钟要从2:59:59调回2:00:00。

请为长途电话服务函数开发计费类，使用等价类分析方法构建测试用例并使用Junit5测试。[java日期时间类详解](https://star.jmhui.com.cn/p1/796.html" \o "java日期时间类详解)

[夏令时参考](https://www.beijing-time.org/NewYork/" \o "参考)

### **采用如下计费规则计算通话费：**

1. 通话时间小于等于20分钟时，每分钟收费0.05美元，通话时间不足1分钟按1分钟计算。
2. 通话时间大于20分钟时，收费1.00美元，外加超过20分钟的部分每分钟0.10美元；
3. 不到1分钟按1分钟计算

### **假设：**

1. 通话计费时间从被叫方应答开始计算，到呼叫方挂机时结束；
2. 通话时间的秒数向上进位到分钟；
3. 没有超过30个小时的通话。

## 代码实现：

### Java部分

**BillingCalculator.java**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.time.ZonedDateTime;

import java.time.temporal.ChronoUnit;

/\*\*

\* 处理通话计费逻辑，包含时长计算和费用计算功能.

\* 该类提供基于调整后通话时长（考虑夏令时和取整规则）的费用计算功能，

\* 支持基础费率和分段费率两种计费模式。

\*/

public class BillingCalculator {

private static final Logger LOGGER = LoggerFactory.getLogger(BillingCalculator.class);

private static final int CEILING\_ADJUSTMENT = 59;

private static final int SECONDS\_PER\_MINUTE = 60;

private static final int BASE\_MINUTES\_LIMIT = 20;

private static final double BASE\_RATE = 0.05;

private static final double EXTRA\_RATE = 0.10;

private static final double BASE\_FEE = 1.00;

private static final int MINIMUM\_BILLABLE\_MINUTES = 1;

/\*\*

\* 计算调整后的通话时长（分钟），考虑夏令时和向上取整规则.

\* 实际计算时会进行以下处理：

\* 1. 计算精确到秒的时间差

\* 2. 向上取整到分钟（例如30秒→1分钟，61秒→2分钟）

\* 3. 保证至少计费1分钟

\*

\* @param start 通话开始时间（带时区）

\* @param end 通话结束时间（带时区）

\* @return 调整后的计费分钟数（至少1分钟）

\*/

public int calculateAdjustedDuration(ZonedDateTime start, ZonedDateTime end) {

long seconds = ChronoUnit.SECONDS.between(start, end);

LOGGER.debug("时间差计算: [{}] 到 [{}] -> {} 秒", start, end, seconds);

long totalMinutes = (seconds + CEILING\_ADJUSTMENT) / SECONDS\_PER\_MINUTE;

LOGGER.debug("向上取整后分钟数: {} 分钟", totalMinutes);

int adjustedMinutes = (int) Math.max(totalMinutes, MINIMUM\_BILLABLE\_MINUTES);

LOGGER.info("最终计费时长: {} 分钟", adjustedMinutes);

return adjustedMinutes;

}

/\*\*

\* 根据调整后的通话时长计算费用.

\* 使用分段计费策略：

\* 1. 20分钟及以下：0.05美元/分钟

\* 2. 超过20分钟：1美元基础费 + 超时部分0.10美元/分钟

\*

\* @param adjustedMinutes 调整后的通话分钟数

\* @return 计算出的费用（美元）

\*/

public double calculateCharge(int adjustedMinutes) {

LOGGER.debug("计算费用，时长: {} 分钟", adjustedMinutes);

double charge;

if (adjustedMinutes <= BASE\_MINUTES\_LIMIT) {

charge = BASE\_RATE \* adjustedMinutes;

LOGGER.trace("基础费率: {}$", charge);

} else {

double extraCharge = EXTRA\_RATE \* (adjustedMinutes - BASE\_MINUTES\_LIMIT);

charge = BASE\_FEE + extraCharge;

LOGGER.trace("分段费率: 基础1$ + 超时部分{}$ = {}$", extraCharge, charge);

}

LOGGER.info("最终费用: {}$", charge);

return charge;

}

}

BillingCalculatorTest.java

import org.junit.jupiter.api.DisplayName;

import org.junit.jupiter.api.Nested;

import org.junit.jupiter.api.Test;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.time.ZoneId;

import java.time.ZonedDateTime;

import static org.junit.jupiter.api.Assertions.assertEquals;

/\*\*

\* 测试BillingCalculator类的计费逻辑，包括基础费率、分段费率、夏令时转换和时间进位规则.

\*/

class BillingCalculatorTest {

private static final Logger LOGGER = LoggerFactory.getLogger(BillingCalculatorTest.class);

private final BillingCalculator calculator = new BillingCalculator();

private static final ZoneId ZONE = ZoneId.of("America/New\_York");

private static final int MONTH\_JUNE = 6;

private static final int HOUR\_29 = 29;

private static final int MINUTE\_20 = 20;

private static final int SECONDS\_59 = 59;

private static final double DELTA = 0.001;

/\*\*

\* 测试基础费率的计费逻辑（通话时长≤20分钟）.

\*/

@Nested

@DisplayName("基础费率（≤20分钟）")

class BaseRateTests {

private static final int YEAR = 2025;

private static final int MONTH\_MAY = 5;

private static final int DAY = 12;

private static final int HOUR = 12;

private static final int MINUTE = 59;

private static final double BASE\_RATE = 0.05;

/\*\*

\* 验证1分钟通话按基础费率计费.

\*/

@Test

@DisplayName("1分钟通话")

void test1MinuteCall() {

LOGGER.info("1分钟通话");

ZonedDateTime start = ZonedDateTime.of(YEAR, MONTH\_JUNE, DAY, HOUR, MINUTE, 0, 0, ZONE);

ZonedDateTime end = start.plusSeconds(SECONDS\_59);

LOGGER.debug("Start: {}, End: {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(1, duration, "应向上取整到1分钟");

assertEquals(BASE\_RATE, calculator.calculateCharge(duration), "费用应为0.05美元");

}

/\*\*

\* 验证20分钟通话按基础费率计费.

\*/

@Test

@DisplayName("精确20分钟通话")

void testExactly20Minutes() {

LOGGER.info("测试精确20分钟通话");

ZonedDateTime start = ZonedDateTime.of(YEAR, MONTH\_MAY, DAY, HOUR, 0, 0, 0, ZONE);

ZonedDateTime end = start.plusMinutes(MINUTE\_20);

LOGGER.debug("Start: {}, End: {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(MINUTE\_20, duration, "应为精确20分钟");

assertEquals(1, calculator.calculateCharge(duration), DELTA, "费用应为1.00美元");

}

}

/\*\*

\* 测试分段费率的计费逻辑（通话时长>20分钟）.

\*/

@Nested

@DisplayName("分段费率（>20分钟）")

class TieredRateTests {

private static final double TIERED\_RATE = 0.10;

private static final double TIERED\_RATE\_1 = 1.10;

private static final int YEAR = 2025;

private static final int MONTH\_6 = 6;

private static final int MONTH\_7 = 7;

private static final int MINUTE\_21 = 21;

private static final int MINUTE\_59 = 59;

private static final int HOUR = 12;

private static final int HOUR\_20 = 20;

private static final int MINUTES\_1800 = 1800;

private static final int DURATION\_21\_MINUTES = 21;

/\*\*

\* 验证21分钟通话按分段费率计费.

\*/

@Test

@DisplayName("21分钟通话（费率切换边界）")

void test21MinutesCall() {

LOGGER.info("21分钟通话（费率切换边界）");

ZonedDateTime start = ZonedDateTime.of(YEAR, MONTH\_6, 1, HOUR, 0, 0, 0, ZONE);

ZonedDateTime end = start.plusMinutes(MINUTE\_21);

LOGGER.debug("Start (EST): {}, End (EDT): {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

assertEquals(DURATION\_21\_MINUTES, duration, "21分钟通话（费率切换边界）");

assertEquals(TIERED\_RATE\_1, calculator.calculateCharge(duration), DELTA, "费用应为1.10美元");

}

/\*\*

\* 验证30小时通话按分段费率计费.

\*/

@Test

@DisplayName("30小时通话（最大允许时长）")

void testMaxDurationCall() {

LOGGER.info("测试超长通话（接近30小时）");

ZonedDateTime start = ZonedDateTime.of(YEAR, MONTH\_7, 1, HOUR, 0, 0, 0, ZONE);

ZonedDateTime end = start.plusHours(HOUR\_29).plusMinutes(MINUTE\_59).plusSeconds(SECONDS\_59);

LOGGER.debug("Start: {}, End: {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(MINUTES\_1800, duration, "29小时59分59秒应进位到30小时");

double expectedCharge = 1.00 + TIERED\_RATE \* (duration - HOUR\_20);

assertEquals(expectedCharge, calculator.calculateCharge(duration), DELTA);

}

}

/\*\*

\* 测试夏令时转换期间的计费处理.

\*/

@Nested

@DisplayName("夏令时转换场景")

class DSTTests {

/\*\*

\* 验证春季夏令时转换期间的计费.

\*/

private static final int MINUTES\_30 = 30;

private static final int MINUTES\_60 = 60;

private static final int MINUTES\_150 = 150;

private static final double TIERED\_RATE\_5 = 5.00;

private static final double TIERED\_RATE\_14 = 14.00;

/\*\*

\* 验证春季夏令时转换期间的计费.

\*/

@Test

@DisplayName("春季转换")

void testSpringDSTWithCorrectDate() {

LOGGER.info("测试春季夏令时转换（丢失1小时）");

// 时间线：01:30 EST → 03:30 EDT（实际物理时间1小时）

ZonedDateTime start = ZonedDateTime.parse("2025-03-12T01:30:00-05:00");

ZonedDateTime end = ZonedDateTime.parse("2025-03-12T03:30:00-04:00");

LOGGER.debug("Start (EST): {}, End (EDT): {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(MINUTES\_60, duration, "应计算实际1小时");

assertEquals(TIERED\_RATE\_5, calculator.calculateCharge(duration), DELTA, "费用应为5.00美元");

}

/\*\*

\* 验证秋季转换内重复时段通话时转换期间的计费.

\*/

@Test

@DisplayName("秋季转换内重复时段通话")

void testCallWithinFallDSTOverlap() {

LOGGER.info("测试秋季夏令时转换（重复1小时）");

// 时间线：01:30 EDT → 本地时间02:30（实际物理时间2.5小时）

ZonedDateTime start = ZonedDateTime.parse("2025-11-05T01:30:00-04:00");

// 保持本地时间连续性

ZonedDateTime end = start.plusHours(1).plusMinutes(MINUTES\_30).withZoneSameLocal(ZONE);

LOGGER.debug("Start (EDT): {}, End (EST): {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(MINUTES\_150, duration, "应计算实际2.5小时");

assertEquals(TIERED\_RATE\_14, calculator.calculateCharge(duration), DELTA, "费用应为14.00美元");

}

}

/\*\*

\* 时间进位规则测试组.

\*/

@Nested

@DisplayName("时间进位测试")

class RoundingTests {

private static final int YEAR = 2025;

private static final int HOUR = 12;

private static final int MINUTES = 10;

private static final int SECONDS = 30;

private static final double TIERED\_RATE = 0.05;

private static final double TIERED\_RATE\_1 = 0.5;

/\*\*

\* 测试短时通话秒数进位规则.

\*/

@Test

@DisplayName("59秒进位到1分钟")

void test59SecondsRounding() {

LOGGER.info("测试短时通话秒数进位规则");

ZonedDateTime start = ZonedDateTime.of(YEAR, MONTH\_JUNE, 1, HOUR, 0, 0, 0, ZONE);

ZonedDateTime end = start.plusSeconds(SECONDS); // 30秒通话

LOGGER.debug("Start: {}, End: {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(1, duration, "30秒应进位到1分钟");

assertEquals(TIERED\_RATE, calculator.calculateCharge(duration), DELTA, "费用应为0.05美元");

}

/\*\*

\* "0秒不进位.

\*/

@Test

@DisplayName("0秒不进位")

void testExactMinute() {

LOGGER.info("0秒不进位");

ZonedDateTime start = ZonedDateTime.of(YEAR, MONTH\_JUNE, 1, HOUR, 0, 0, 0, ZONE);

ZonedDateTime end = start.plusMinutes(MINUTES);

LOGGER.debug("Start: {}, End: {}", start, end);

int duration = calculator.calculateAdjustedDuration(start, end);

LOGGER.debug("Adjusted duration: {} minutes", duration);

assertEquals(MINUTES, duration);

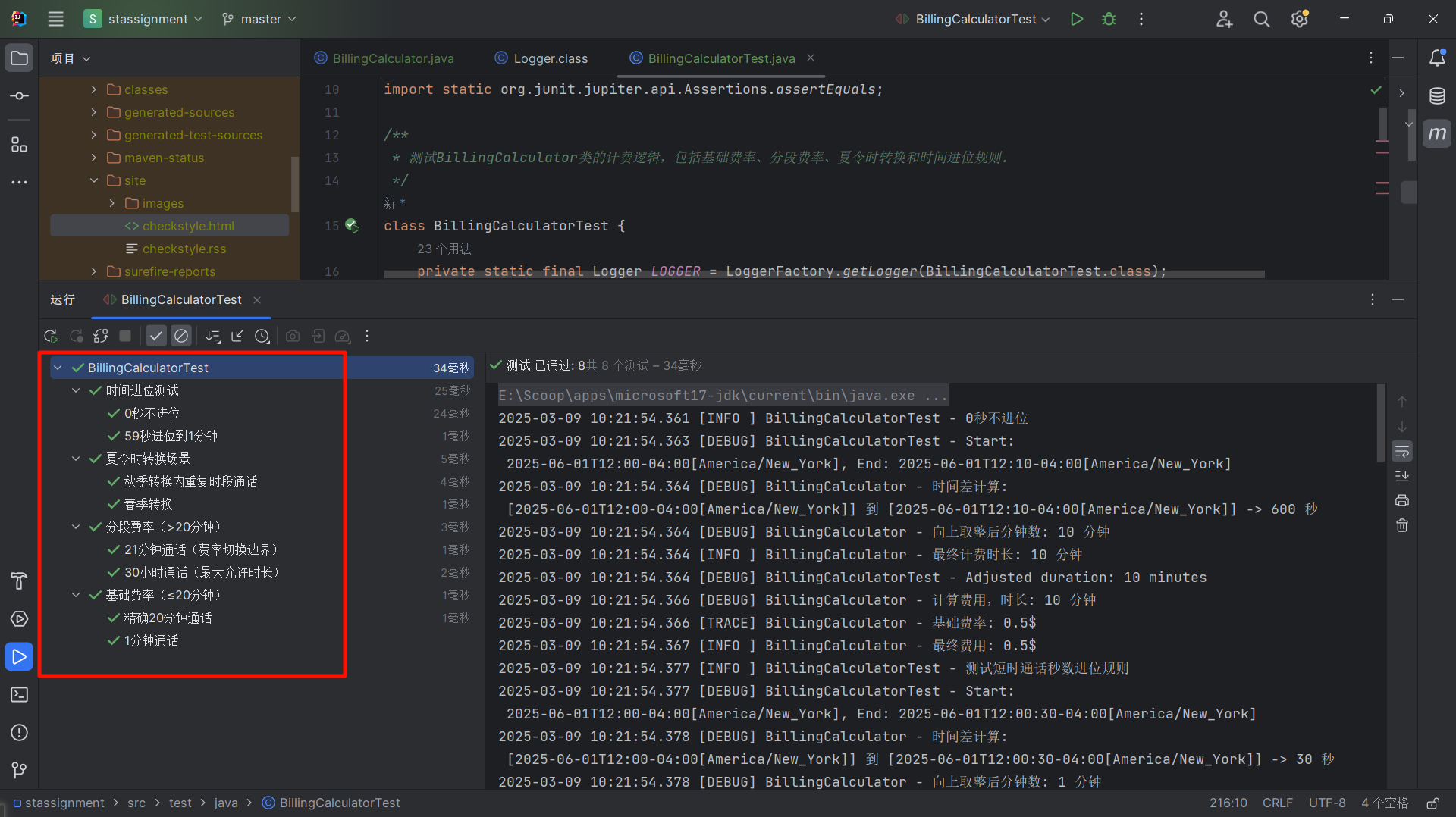
assertEquals(TIERED\_RATE\_1, calculator.calculateCharge(duration), DELTA, "费用应为0.5美元");

}

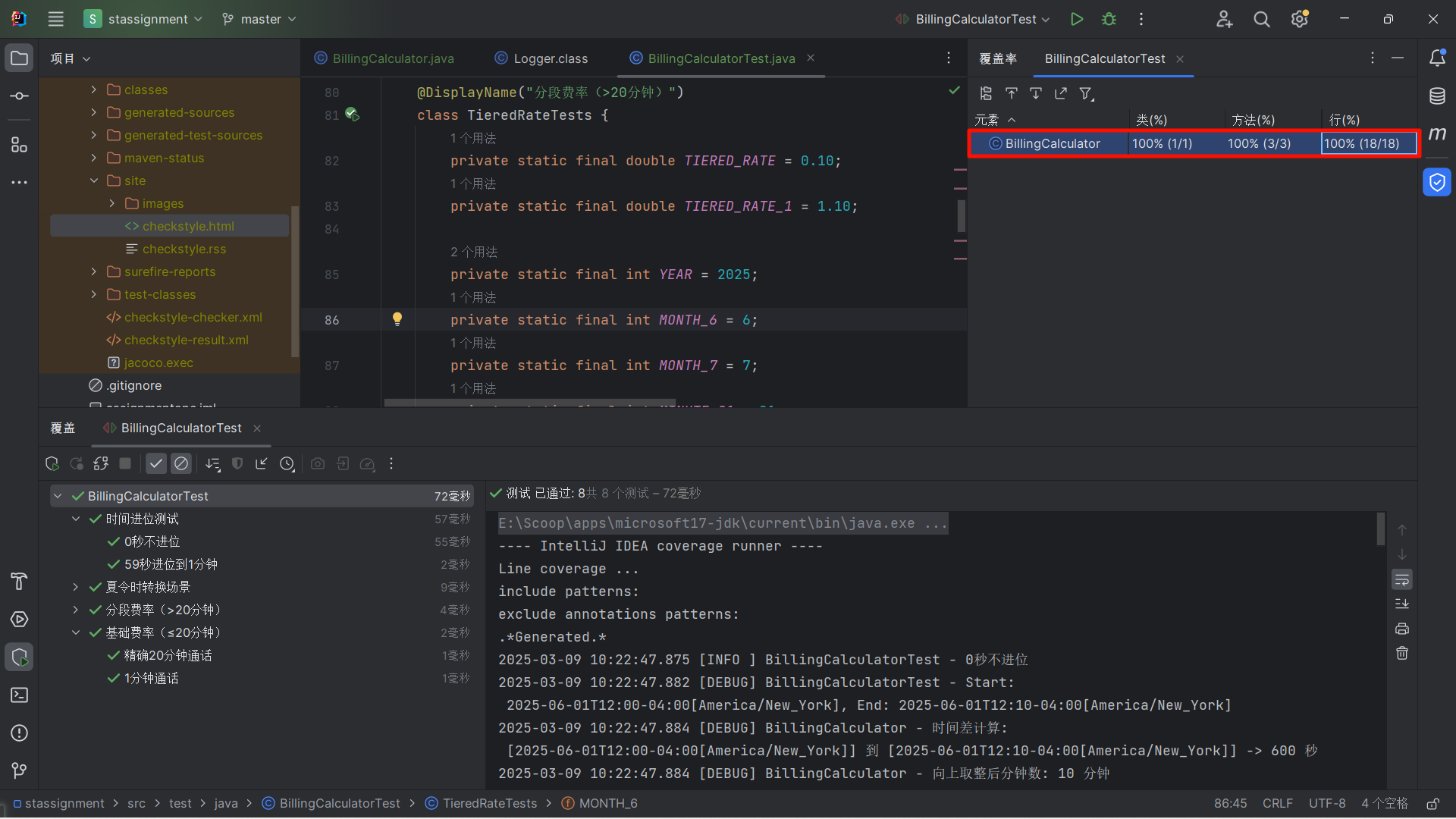
}

}

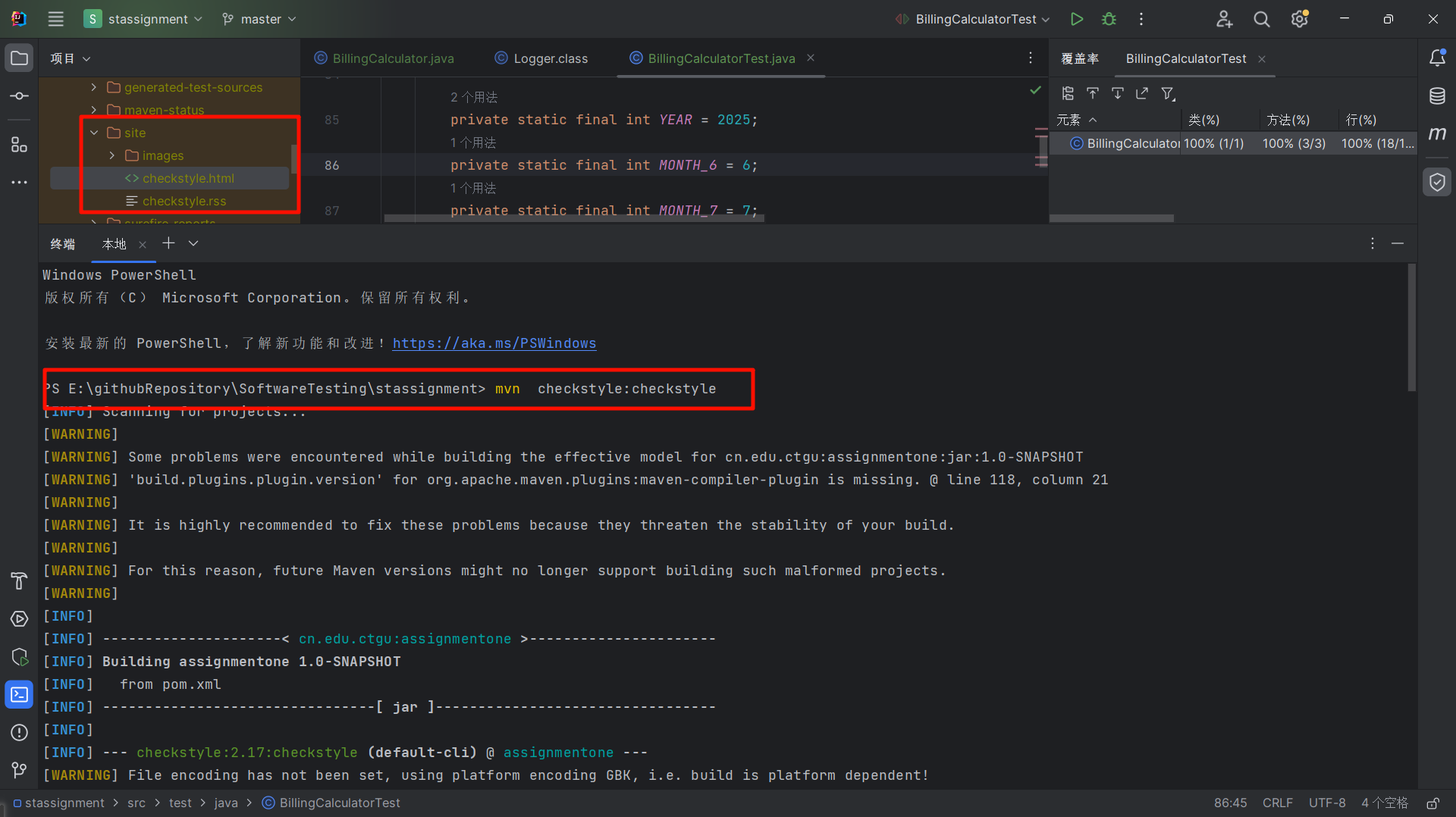
#### Idea工具测试用例 覆盖率结果 checkstyle jacoco

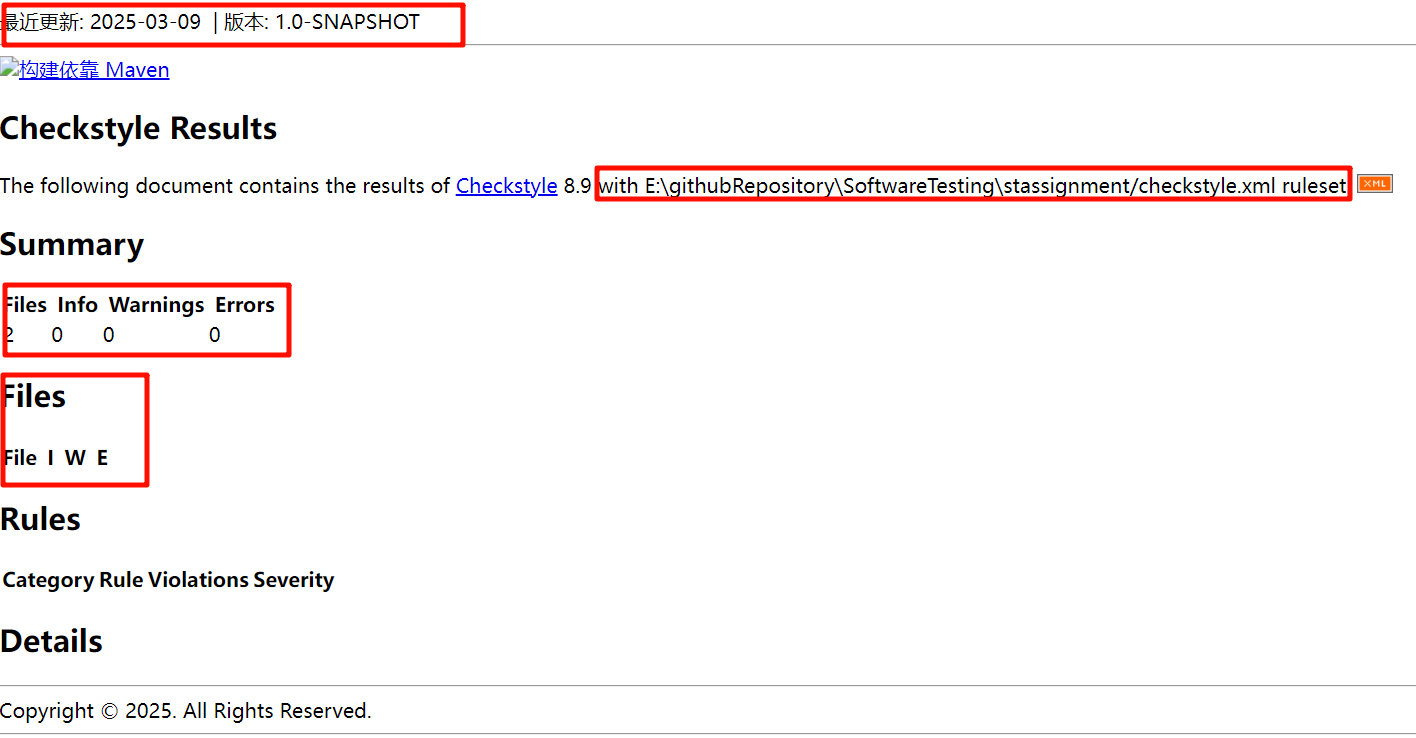


**覆盖率**

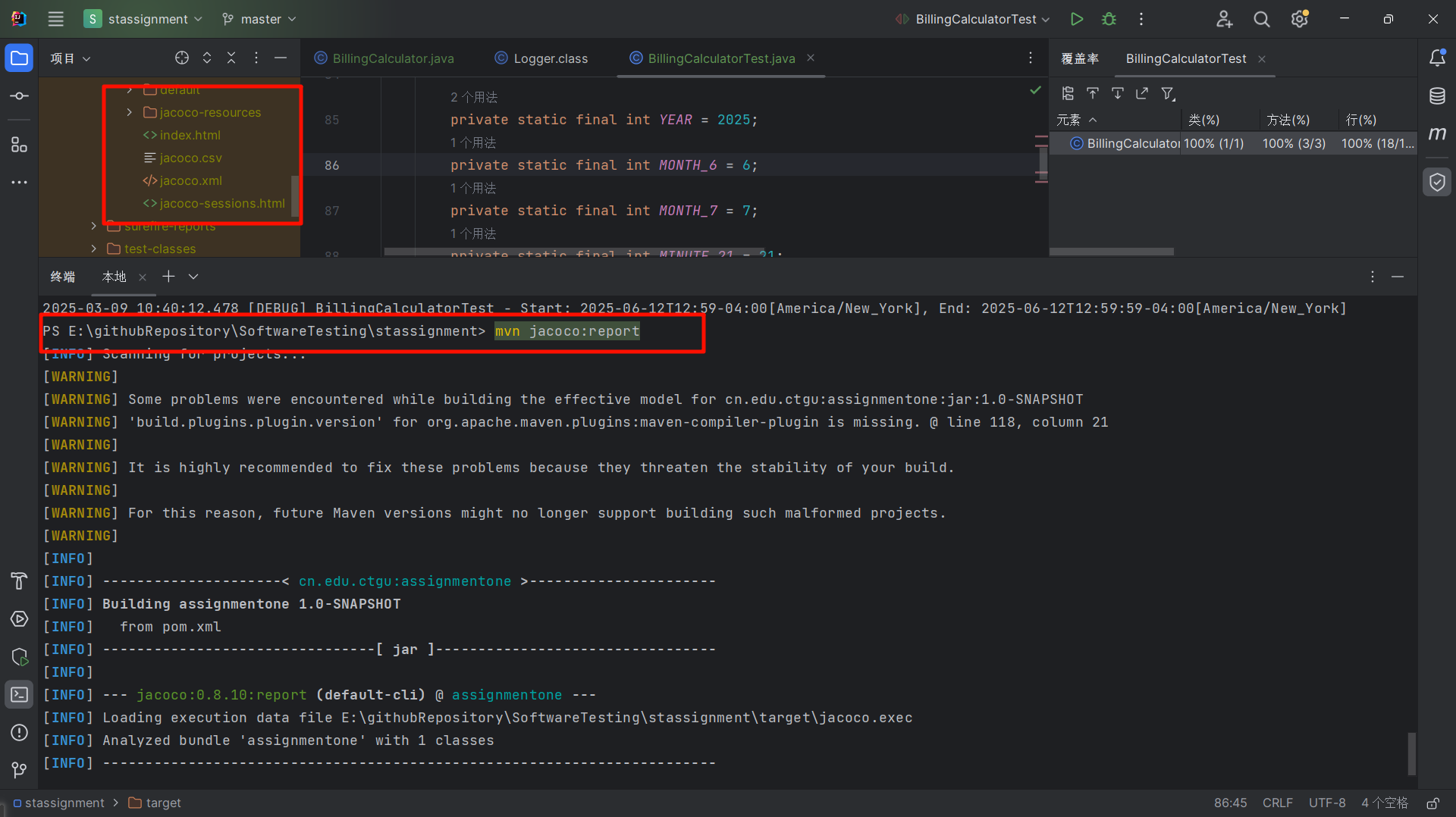


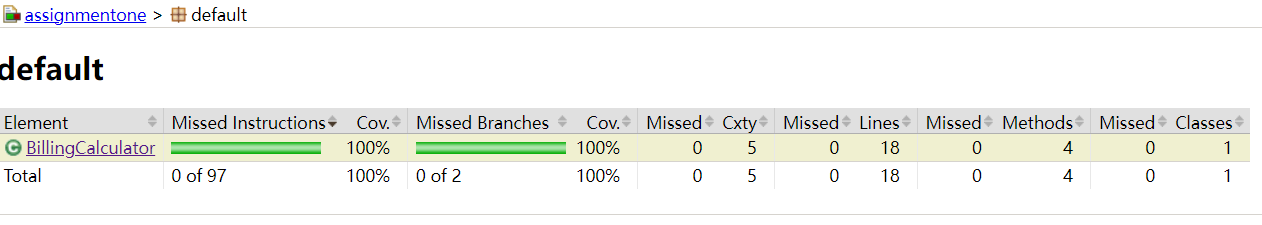
**Checkstyle**





**Jacoco**





### Golang部分

BillingCalculator.go

package Billing

import (

"log"

"time"

)

const (

CeilingAdjustment = 59 SecondsPerMinute = 60 BaseMinutesLimit = 20 BaseRate = 0.05 ExtraRate = 0.10 BaseFee = 1.00 MinimumBillableMinutes = 1

)

// BillingCalculator 处理通话计费逻辑，包含时长计算和费用计算功能

type BillingCalculator struct {

logger \* log.Logger

}

// NewBillingCalculator 创建一个新的BillingCalculator实例

func NewBillingCalculator() \* BillingCalculator {

return &BillingCalculator {

logger: log.New(log.Writer(), "BillingCalculator: ", log.Flags()),

}

}

// CalculateAdjustedDuration 计算调整后的通话时长（分钟），考虑夏令时和向上取整规则

func(b \* BillingCalculator) CalculateAdjustedDuration(start, end time.Time) int {

duration: = end.Sub(start)

seconds: = int(duration.Seconds())

b.logger.Printf("时间差计算: [%.0d] 秒", seconds)

totalMinutes: = (seconds + CeilingAdjustment) / SecondsPerMinute

b.logger.Printf("向上取整后分钟数: %d 分钟", totalMinutes)

adjustedMinutes: = totalMinutes

if adjustedMinutes <= MinimumBillableMinutes {

adjustedMinutes = MinimumBillableMinutes

}

b.logger.Printf("最终计费时长: %d 分钟", adjustedMinutes)

return adjustedMinutes

}

// CalculateCharge 根据调整后的通话时长计算费用

func(b \* BillingCalculator) CalculateCharge(adjustedMinutes int) float64 {

b.logger.Printf("计算费用，时长: %d 分钟", adjustedMinutes)

var charge float64

if adjustedMinutes <= BaseMinutesLimit {

charge = BaseRate \* float64(adjustedMinutes)

b.logger.Printf("基础费率: %.2f 美元", charge)

} else {

extraMinutes: = adjustedMinutes - BaseMinutesLimit

extraCharge: = ExtraRate \* float64(extraMinutes)

charge = BaseFee + extraCharge

b.logger.Printf("分段费率: 基础 %.2f 美元 + 超时部分 %.2f 美元 = %.2f 美元", BaseFee, extraCharge, charge)

}

b.logger.Printf("最终费用: %.2f 美元", charge)

return charge

}

BillingCalculator\_test.go

package Billing

import (

"log"

"testing"

"time"

)

// TestBillingCalculator 测试BillingCalculator的计费逻辑

func TestBillingCalculator(t \* testing.T) {

calculator: = NewBillingCalculator()

// 测试基础费率

t.Run("基础费率（≤20分钟）", func(t \* testing.T) {

test1MinuteCall(t, calculator)

testExactly20Minutes(t, calculator)

})

// 测试分段费率

t.Run("分段费率（>20分钟）", func(t \* testing.T) {

test21MinutesCall(t, calculator)

testMaxDurationCall(t, calculator)

})

// 测试夏令时转换

t.Run("夏令时转换场景", func(t \* testing.T) {

testSpringDSTWithCorrectDate(t, calculator)

testCallWithinFallDSTOverlap(t, calculator)

})

// 测试时间进位规则

t.Run("时间进位测试", func(t \* testing.T) {

test59SecondsRounding(t, calculator)

testExactMinute(t, calculator)

})

}

// 测试1分钟通话

func test1MinuteCall(t \* testing.T, calculator \* BillingCalculator) {

log.Println("1分钟通话")

start: = time.Date(2025, 6, 12, 12, 59, 0, 0, time.UTC)

end: = start.Add(59 \* time.Second)

log.Printf("Start: %v, End: %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 1 {

t.Errorf("应向上取整到1分钟，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

if charge != 0.05 {

t.Errorf("费用应为0.05美元，实际得到: %.2f", charge)

}

}

// 测试精确20分钟通话

func testExactly20Minutes(t \* testing.T, calculator \* BillingCalculator) {

log.Println("测试精确20分钟通话")

start: = time.Date(2025, 5, 12, 12, 0, 0, 0, time.UTC)

end: = start.Add(20 \* time.Minute)

log.Printf("Start: %v, End: %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 20 {

t.Errorf("应为精确20分钟，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

if charge != 1.00 {

t.Errorf("费用应为1.00美元，实际得到: %.2f", charge)

}

}

// 测试21分钟通话（费率切换边界）

func test21MinutesCall(t \* testing.T, calculator \* BillingCalculator) {

log.Println("21分钟通话（费率切换边界）")

start: = time.Date(2025, 6, 1, 12, 0, 0, 0, time.UTC)

end: = start.Add(21 \* time.Minute)

log.Printf("Start: %v, End: %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

if duration != 21 {

t.Errorf("应为21分钟，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

if charge != 1.10 {

t.Errorf("费用应为1.10美元，实际得到: %.2f", charge)

}

}

// 测试超长通话（接近30小时）

func testMaxDurationCall(t \* testing.T, calculator \* BillingCalculator) {

log.Println("测试超长通话（接近30小时）")

start: = time.Date(2025, 7, 1, 12, 0, 0, 0, time.UTC)

end: = start.Add(29 \* time.Hour + 59 \* time.Minute + 59 \* time.Second)

log.Printf("Start: %v, End: %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 1800 {

t.Errorf("29小时59分59秒应进位到30小时，实际得到: %d", duration)

}

expectedCharge: = 1.00 + 0.10 \* float64(duration - 20)

charge: = calculator.CalculateCharge(duration)

if charge != expectedCharge {

t.Errorf("费用应为%.2f美元，实际得到: %.2f", expectedCharge, charge)

}

}

// 测试春季夏令时转换（丢失1小时）

func testSpringDSTWithCorrectDate(t \* testing.T, calculator \* BillingCalculator) {

log.Println("测试春季夏令时转换（丢失1小时）")

start, \_: = time.Parse("2006-01-02T15:04:05-07:00", "2025-03-12T01:30:00-05:00")

end, \_: = time.Parse("2006-01-02T15:04:05-07:00", "2025-03-12T03:30:00-04:00")

log.Printf("Start (EST): %v, End (EDT): %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 60 {

t.Errorf("应计算实际1小时，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

if charge != 5.00 {

t.Errorf("费用应为5.00美元，实际得到: %.2f", charge)

}

}

// 测试秋季夏令时转换（重复1小时）

func testCallWithinFallDSTOverlap(t \* testing.T, calculator \* BillingCalculator) {

log.Println("测试秋季夏令时转换（重复1小时）")

start, \_: = time.Parse("2006-01-02T15:04:05-07:00", "2025-11-05T01:30:00-04:00")

end: = start.Add(1 \* time.Hour + 30 \* time.Minute).In(time.UTC)

log.Printf("Start (EDT): %v, End (EST): %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 90 {

t.Errorf("应计算实际1.5小时，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

if charge != 8.00 {

t.Errorf("费用应为8.00美元，实际得到: %.2f", charge)

}

}

// 测试59秒进位到1分钟

func test59SecondsRounding(t \* testing.T, calculator \* BillingCalculator) {

log.Println("测试短时通话秒数进位规则")

start: = time.Date(2025, 6, 1, 12, 0, 0, 0, time.UTC)

end: = start.Add(30 \* time.Second)

log.Printf("Start: %v, End: %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 1 {

t.Errorf("30秒应进位到1分钟，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

if charge != 0.05 {

t.Errorf("费用应为0.05美元，实际得到: %.2f", charge)

}

}

// 测试0秒不进位

func testExactMinute(t \* testing.T, calculator \* BillingCalculator) {

log.Println("0秒不进位")

start: = time.Date(2025, 6, 1, 12, 0, 0, 0, time.UTC)

end: = start.Add(10 \* time.Minute)

log.Printf("Start: %v, End: %v", start, end)

duration: = calculator.CalculateAdjustedDuration(start, end)

log.Printf("Adjusted duration: %d minutes", duration)

if duration != 10 {

t.Errorf("应为10分钟，实际得到: %d", duration)

}

charge: = calculator.CalculateCharge(duration)

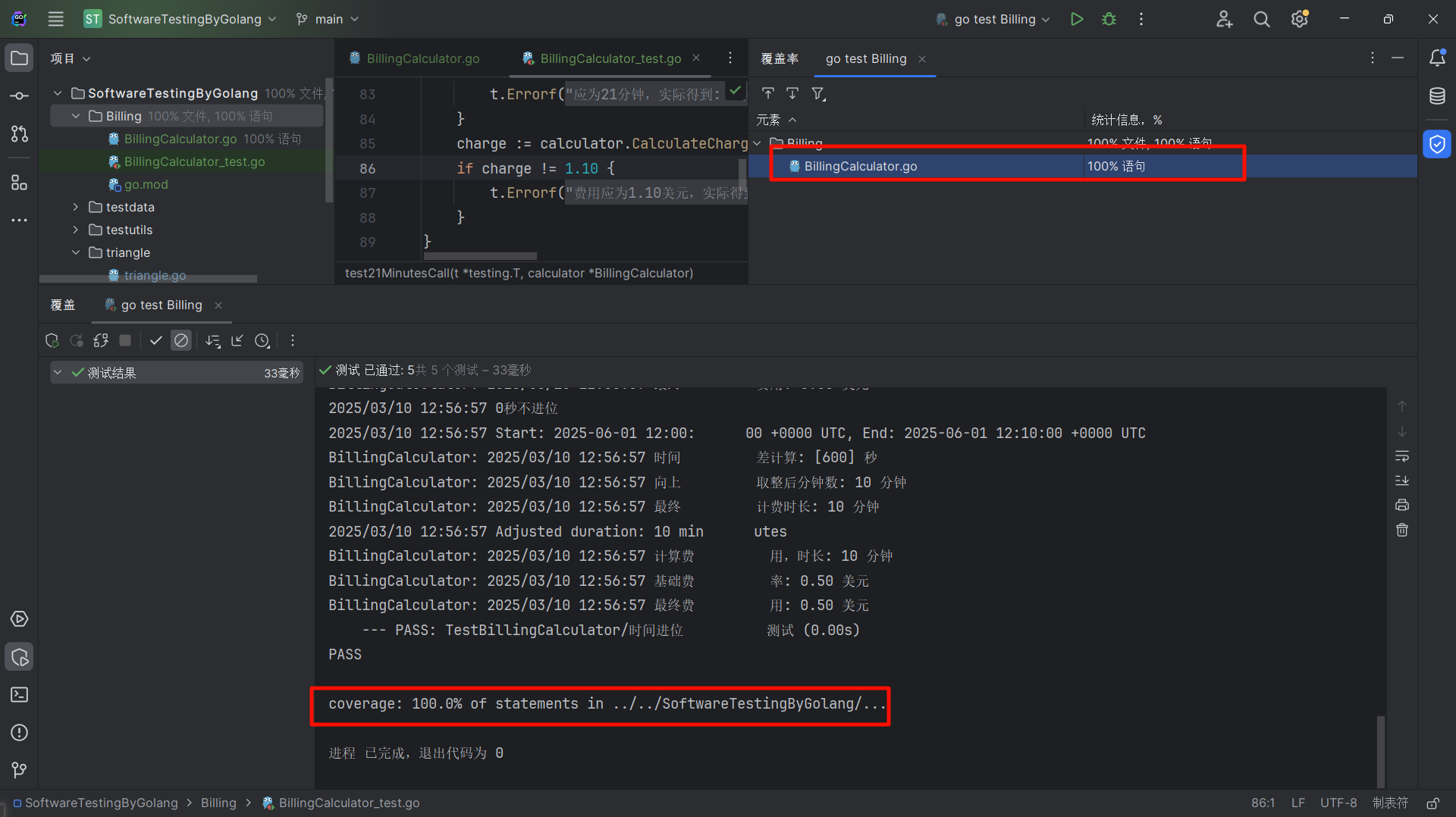
if charge != 0.50 {

t.Errorf("费用应为0.50美元，实际得到: %.2f", charge)

}

}

#### Idea工具测试用例 覆盖率结果



# 五、结论分析与体会

在测试代码中集成日志记录功能，能够帮助开发者更好地跟踪测试执行过程，尤其是在测试失败时，日志可以提供详细的上下文信息，便于快速定位问题。我们使用了 Go 标准库 log 来实现日志记录，并在测试通过或失败时分别记录了相关信息。

测试与报告的依赖关系：无论是 Go 还是 Java，生成报告的前提都是先运行测试。例如，单独执行 mvn jacoco:report 会失败，必须通过 mvn test jacoco:report 确保测试数据已生成。单独执行 mvn checkstyle:checkstyle会失败，必须通过 mvn test checkstyl:checkstyle确保测试数据已生成。

Go 语言通过内置的 go test 工具实现了开箱即用的测试能力，配合 -cover 和 -json 参数可以快速生成覆盖率报告和结构化测试结果。

Java 的生态复杂性：Java 依赖 Maven/Gradle 等构建工具，通过插件（如 Surefire、Jacoco）实现测试和覆盖率统计。使用chcekstyle进行静态调整代码规范

# 仓库地址

https://github.com/xieyangyuyue/SoftwareTesting

[xieyangyuyue/SoftwareTesting: 软件测试--java (github.com)](https://github.com/xieyangyuyue/SoftwareTesting)