

JUnit使用说明

简介

JUnit是一款在单元测试领域很著名的测试工具，为了倡导同学们学习单元测试，我们的oopre课程会增加一些JUnit的内容和要求。

以后的测评中，我们会专门在中测中设置JUnit的测试点，所以希望同学们能尽量掌握它的基础使用方式，在JUnit的帮助下更好地进行oopre的代码测试。

重要说明 本教程仅作为补充资料辅助同学们使用和理解JUnit，一切作业要求和测评要求以指导书为准。

使用

本使用说明以第一次作业的代码为例，简单说明如何写出一个比较全面的单元测试。

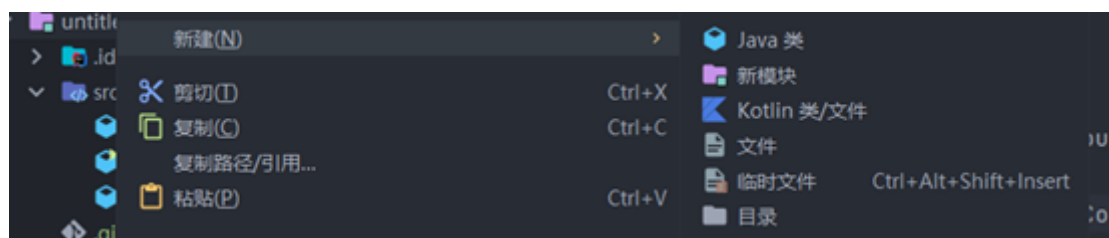
安装与建立

IDEA是自带JUnit的。只需要通过点击IDEA的相应部分，就可以完成对当前代码的测试构建。

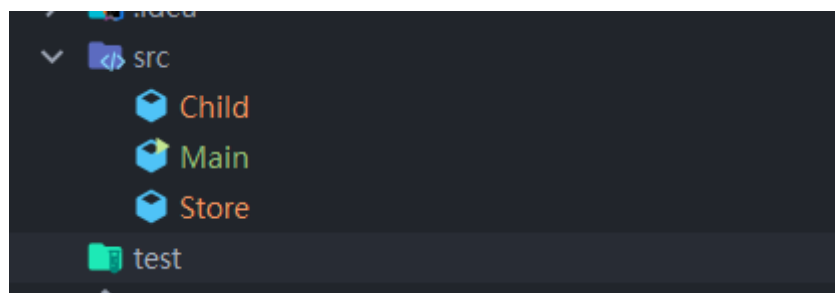
现在进入我们第一次作业改正后的代码项目：

为了将代码实现与测试分离，我们希望将测试的代码专门放在一个文件夹里面。

把鼠标移动到项目顶层模块上，右键新建一个名为test的文件夹。



点击目录，名字叫test（这里，我们约定，以后的作业中放置JUnit的测试代码的文件夹都叫做test）



然后将鼠标移动到test文件夹上，右键，找到下面的将源代码标记为、选择测试源代码根目录（就是第二个）

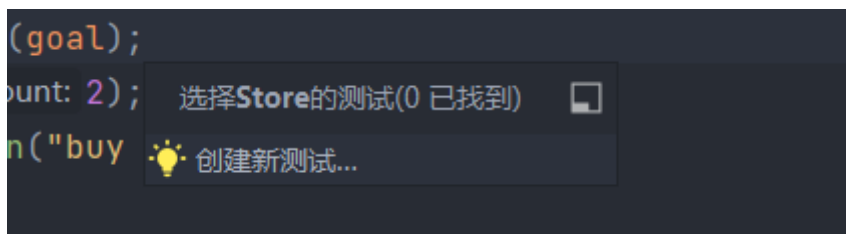


以上，我们就设置成功了测试文件的文件夹，以后的测试文件需要被放在这个文件夹内

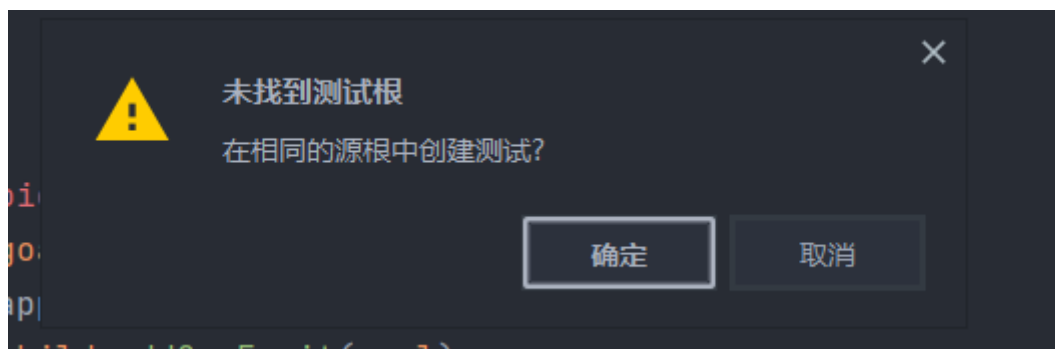
接下来，我们开始生成对应的测试代码。在Child或者Store文件里点击一下右键：



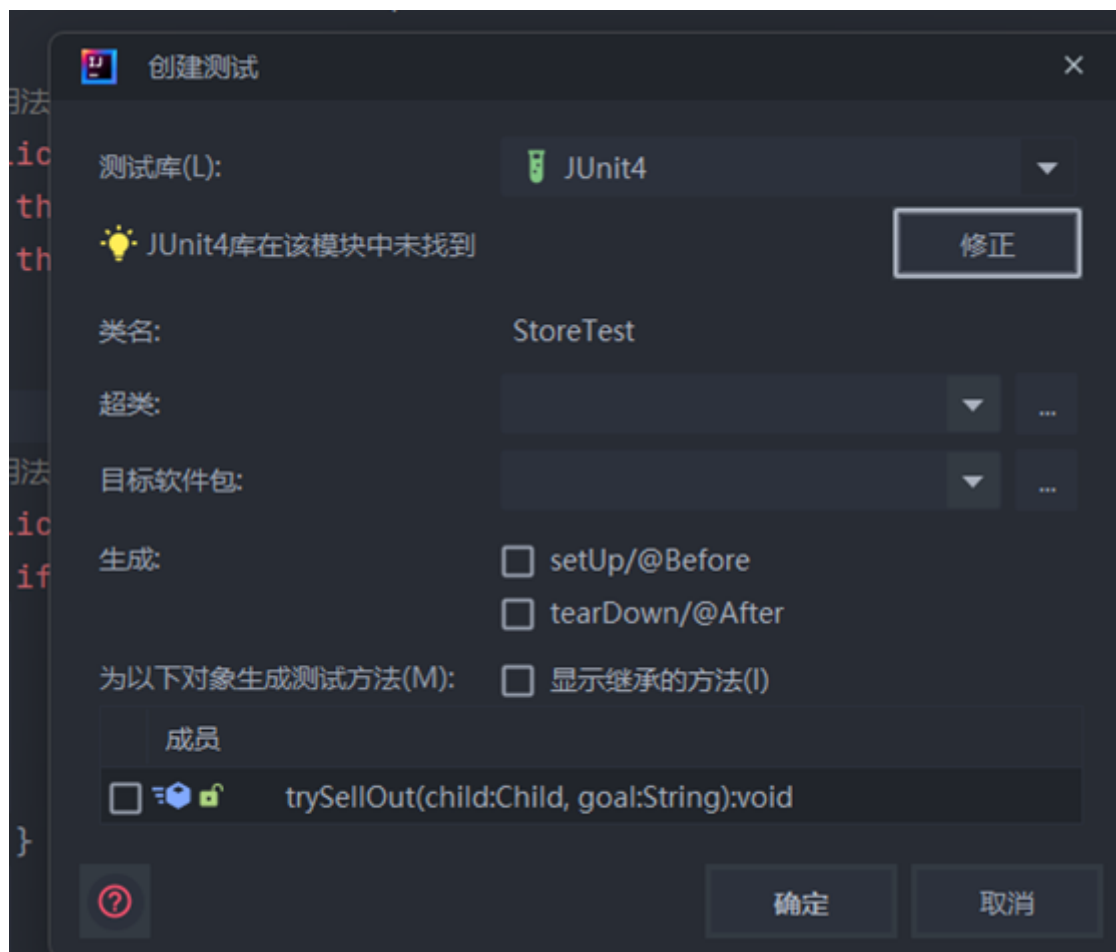
我们把鼠标移动到转到那一行（不用点击）然后会自动出现一个扩展栏，我们看到这个扩展栏最下面（图片右下角）会有一个测试，点击它。



当然我们现在应当是什么测试都没有的，显示0很正常。点击创建新测试。



点击确定。



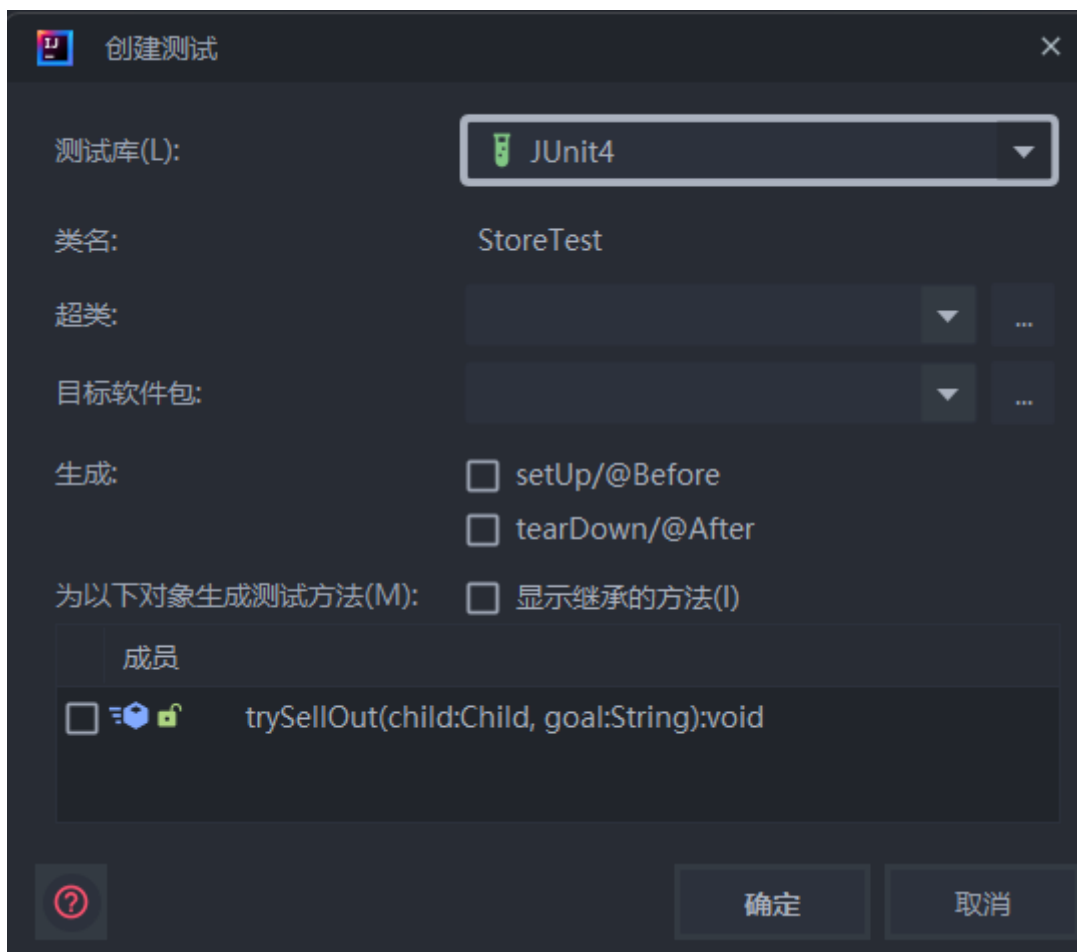
很重要的界面。测试库，就是我们选择JUnit工具的哪个版本。这里课程组统一规定使用JUnit4

看见灯泡这里，显示JUnit4库在该模块中未找到。点击修正。



点击确定。下载到可点可不点。

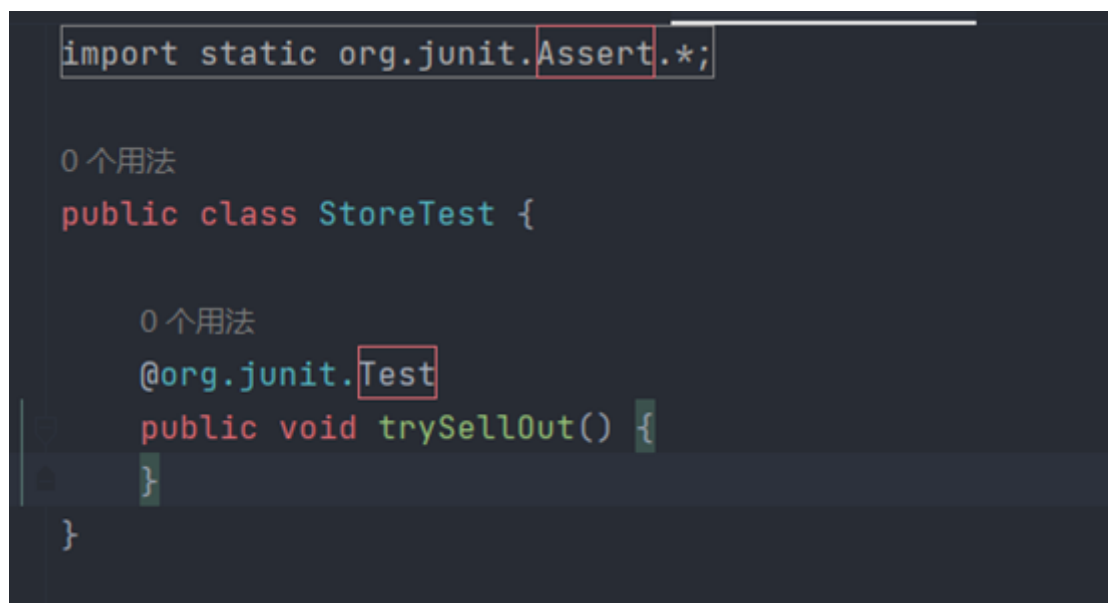
之后界面会变成现在这样



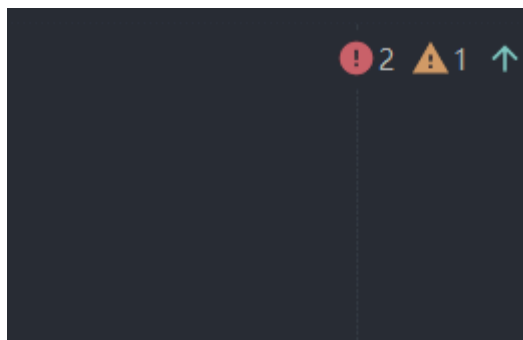
勾选那个成员里面的方框，代表你将对这个方法进行测试。Before和After可选可不选。之后点击确定。

(补充解释一下，这两个选项是会自动生成两个方法，Before在所有测试前调用，After在所有测试后调用，如果你写的测试方法里有相同的部分，可以选择将这一部分放到Before和After中，请同学们感兴趣的自己摸索，该部分非必须了解)

我们发现出现了一个新的文件。文件名是类名+Test。



你可能看见了红色的报错。没关系。IDEA有自动向你提供修复手段的能力：

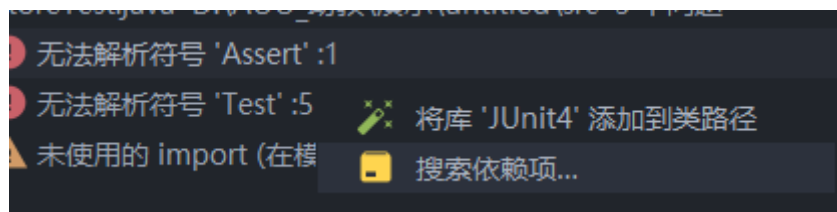


右上角会出现这样的提示。

鼠标点击一下，在下面会出现这样的报错，在报错上右键点击：



(也许显示快速修复那里是灰色的，多尝试几次就好了)



点击：将库添加到类路径。

然后报错就消失了，我们就能正常使用JUnit了。

以上自动出现的代码都是Idea自动生成的，为了方便评测，在此，我们提出一点格式的规定，在此罗列出你应当修改的部分：

- 在test文件代码前输入 `import org.junit.Test;`
- 保证每一个注解的内容是 `@Test`

通俗来讲，就是：

```
import org.junit.Test;
0 个用法 何立群
public class ChildTest {

    0 个用法 何立群
    @Test
    public void subMoney() {
        Child child = new Child( money: 20);
        child.subMoney( count: 5);
        assert (child.getMoney() == 15);
        // 看看是不是确实的减去了那么多钱? , 比如敲错了写成=而不是-=了
    }

    0 个用法 何立群
    @Test
    public void addOneFruit() {
        Child child = new Child( money: 20);
        child.addOneFruit( goal: "apple");
        assert (child.getAppleCount() == 1);
        //是否加上了?
        child.addOneFruit( goal: "banana");
    }
}
```

你需要保证在文件起始部分有相同的import代码

同时需要保证每一个测试方法的注解为@Test

就像这样：

```
@Test
public void add
```

代码编写

初步运行

在看这一步之前，你需要将Child文件的测试文件也生成好。

我们将从Child开始介绍如何写测试文件。

```

5  ▶▶  public class ChildTest {
6
7      @Test
8  ▶▶  public void subMoney() {
9      }
10
11     @Test
12  ▶▶  public void addOneFruit() {
13     }
14
15     @Test
16  ▶▶  public void eat() {

```

先看这些方法，你会发现所有的都有一个小箭头。

你可以去看一下main方法（我们都知道main是程序的入口）那里也有一个小箭头。

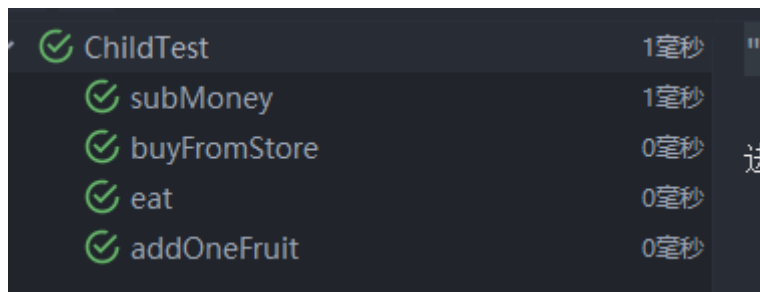
这说明了新建的方法每一个都拥有独立运行的能力，不和main方法相牵涉。因此，需要测试的时候，我们可以在保留原本main的内容的情况下，设计不同的测试程序，并且独立的运行。

当然，你现在可以点一下这个箭头试试。我们不仿点击class那里的箭头



有两个带运行的东西，一个是单纯的运行，一种是使用覆盖率运行。后者将是我们测评所依据的一个指标。

点击一下运行？它会运行所有在这个文件里面的测试方法。



啊显示全过了！

这里没有人和你对拍输出什么的，只要程序执行到最后就算通过。

因此，我们要设置一些东西，让它在有问题的情况下不能完整执行。

Assert检查

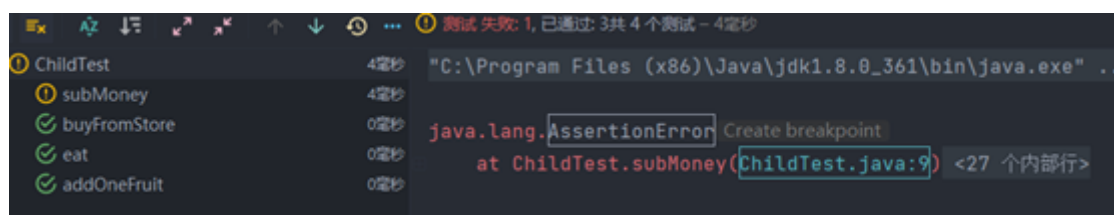
assert叫做断言，是一种能够在程序中发现错误并及时终止程序的手段。在这里，我们只讨论他在junit里面的使用。

```
assert(a==b);
```

在程序执行到这一行的时候，会检查布尔表达式 `a==b` 的值是否为真，假如是真，则跳过，否则会报错停止程序运行。

```
@Test
public void subMoney() {
    assert (1==2);
}
```

这么写一下，然后用刚才的方式运行。



报错了，显示有一个没有通过。这样就能发现错误了。

与方法相关

当然，`1==2`，并没有任何意义。我们要的是测试方法的正确与否，而不是看它爆个红图吉祥。

我们需要检查的是，**在一个方法执行前后，类的属性是否发生了我们所想要的变化，是否保持了我們不想改变的状态。**

在oo正课的第三单元，你会对这一点有着更深刻的体会，不过，那都是后话了。

但是，可能我们的类中，有一些信息，不能被外界直接访问，例如money只能通过

```
public void subMoney(int count) {  
    money -= count;  
}
```

来修改，并且没有向外界暴露它的手段。你可以选择用设置get方法来获取其当前值，也可以专门设置方法判断返回布尔值。

```
public int getMoney() {  
    return money;  
}  
public boolean moneyIs(int testValue) {  
    return this.money==testValue;  
}
```

这里我们选择前者。

要知道，我们方法的执行主体是对象，因此，我们需要在这个方法里面新建一个对象，然后让它调用这个方法：

```
public void subMoney() { //这个是测试文件中的，和Child.java中的不是一个方法  
    Child child=new Child(20);  
    child.subMoney(5);  
    .....  
}
```

我们要测的是subMoney方法的正确性，那么如何知道这个方法的正确性呢？

Money之前的数额减去传进来的参数，该剩下多少呢？

15 对吧，所以我们要测试的就是这个了：

```
public void subMoney() {  
    Child child=new Child(20);  
    child.subMoney(5);  
    assert(child.getMoney()==15);  
    // 看看是不是确实的减去了那么多钱？，比如敲错了写成=而不是-=了  
}
```

这下运行一下，嗯通过了，所以没有写成=，很不错。

但是在这里提几个问题给大家：

- 如果选择减去的数是10，能测出来如-=写成=的错误吗？

- 我们只测了一个数5，但是假如要测试的方法很复杂怎么办？或者原本的20换成某个接近int负极限的负数呢？所以一个测试够吗？

充分测试！！！！

覆盖率

什么事覆盖

上面的方法很简单，从开始直接执行就可以执行到底。

我们考虑这样一个方法：

```
public boolean lessThanHundred(int x) {  
    if(x<100) {  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

然后，假设你兴高采烈的输入这样一组输入来测试：

1,2,.....99

很多吧，99个，总够用了。

但是这些永远返回的是true。

一直在走的代码，只有这么几行

```
if(x<100){  
    return true;  
}
```

下面的代码，从来 没有被走到过。

假设我们把下面的代码改成：

```
else {  
    return true;  
}
```

我们照样不会发现错误，因为我们的测试，根本没有测到这一行。

或者说，没有覆盖到这一行。

被覆盖的行，就是我们的业务代码的测试中，被真正执行到的行。虽然在测试的时候，仅仅执行到了这一行，不足以说明这一行是正确的，但是假如执行它后没有发现错误，我们对这行代码的信心也会大大提高。

因此，我们在测评同学们的测试代码时，会将覆盖率作为参照的重要指标。

哪些覆盖率？

覆盖率本身是一个很宽泛的事情，例如，我们应当以什么为单位来计算这个率？

测试到这个方法了吗？测试到这个行了吗？测试到这个类了吗？

按照单位的细度不同，可能评价的结果不同。

下面是一些可能的具体覆盖率种类

- 函数覆盖率：定义的函数中有多少被调用
- 语句覆盖率：程序中的语句有多少被执行
- 分支覆盖率：有多少控制结构的分支（例如if语句）被执行
- 条件覆盖率：有多少布尔子表达式被测试为真值和假值
- 行覆盖率：有多少行的源代码被测试过

值得指出的是，假设代码的一行只写一条语句的话，行覆盖率和语句覆盖率应当是基本等价的。因此在JUnit+IDEA的工具链中，我们只需要评价测试的行覆盖率即可。

分支覆盖率

现在再来看我们刚才举的例子：

```
if(x<100) {  
    return true;  
}  
else {  
    return false;  
}
```

我们可以用分支覆盖的视角来审查这个方法的测试。

显然对于x的不同值，该方法中一共有两个分支，假设仅仅测试到 `if(x<100)`，那么仅仅覆盖到了一半的分支，所以分支的覆盖率应当是50%。

如果多加了一些代码呢？

```
if(x<100){
    return true;
}
else {
    /*
    do_something
    */
    return false;
}
```

假设 `do_something` 是一串很复杂的程序，那么仅仅有 `x<100` 的情况的时候分支覆盖率是多少呢？

还是50%！但是事实上，我们意识到，功能复杂的 `do_something` 很可能是我们写的代码里面要测试的重中之重。50%的覆盖率（如果草率的用分支覆盖率来评价的话），显然夸大了我们测试的效果，是不大合适的，所以，更多的，角度更丰富的其他覆盖率评价是必要的。

语句覆盖率

上面提到，分支覆盖率是不够的，我们来思考一下，那个很复杂的一串代码，它占比这么大是因为什么？

因为复杂。为什么复杂？因为语句多。

是的，因此，我们也需要以语句为单位考虑测试的覆盖率。

```
if(x>=60 ){
    return true;
}
else {
    noticeMakeUpExamintaion(); //通知补考
    arrangeSeat(); //安排座位
    registerFailureInformation(); //登记挂科信息
}
```

假设我们测试的时候，仅仅进行了 `x>=60` 的分支，那么语句执行的数目是1。

假设仅仅进行了 `x<60` 的分支那么语句的执行数目是3，语句覆盖率为75%。

当然，虽然上面的两种情况都是代表你做的测试是不充分的，但是语句覆盖率高的那种，更大可能做了更有效的测试。

IDEA+JUnit中的覆盖率

我们在IDEA中运行JUnit，默认会给出下面这几种覆盖率的反馈：

- 类覆盖率：有多少类被测试过
- 方法覆盖率：有多少方法被测试过
- 行覆盖率：有多少行的源代码被测试过

```
public void addOneFruit(String goal) {  
    if (goal.equals("apple")) {  
        appleCount++;  
    } else if (goal.equals("banana")) {  
        bananaCount++;  
    }  
}
```

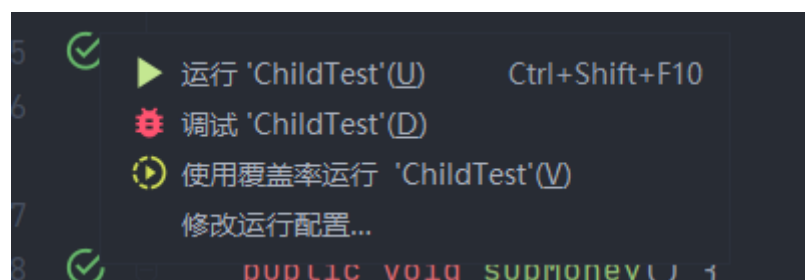
现在看看另一个函数。

它有两个分支。

```
public void addOneFruit() {  
    Child child =new Child(20);  
    child.addOneFruit("apple");  
    assert (child.getAppleCount()==1);  
    //是否加上了?  
    .....  
}
```

这样的代码是否足够？应该不太足够。

有什么可以量化这种不太足够吗？别忘了我们上面提到的覆盖率运行



点击一下，然后你会在右边发现一个很神奇的统计数据：

覆盖率: ChildTest x

元素 ^	类(%)	方法(%)	行(%)
所有	40% (2/5)	56% (9/16)	31% (18/57)
Child	100% (1/1)	62% (5/8)	37% (9/24)
ChildTest	100% (1/1)	100% (4/4)	100% (9/9)
Main	0% (0/1)	0% (0/1)	0% (0/9)
Store	0% (0/1)	0% (0/2)	0% (0/14)
StoreTest	0% (0/1)	0% (0/1)	0% (0/1)

类的百分号代表你是否测试了这个类。

方法的百分号代表你写的测试覆盖到了多少个这个类的方法。

行的百分号代表了你写的测试覆盖到了这个文件的多少个有效行（不是大括号那种）

同时，你点击类的文件，会发现其中多了很多红红绿绿的行：

```
public void subMoney(int count) {
    money -= count;
}

4 个用法
public void addOneFruit(String goal) {
    if (goal.equals("apple")) {
        appleCount++;
    } else if (goal.equals("banana")) {
        bananaCount++;
    }
}
```

仔细观察一下，绿的使我们在刚才的测试中测试到的行，红色的则并未涉及。

注意统计数据统计的是这个文件，我们才写了两个方法的测试，当然数字很可怜。我们再修改一下这个运行的方法

```
public void addOneFruit() {
    Child child = new Child(20);
    child.addOneFruit("apple");
    assert (child.getAppleCount() == 1);
    //是否加上了?
    child.addOneFruit("banana");
    assert (child.getBananaCount() == 1);
    //是否加上了?
}
```

再运行一下（记得选用覆盖率运行）

元素 ^	类(%)	方法(%)	行(%)
所有	40% (2/5)	62% (10/16)	38% (23/59)
Child	100% (1/1)	75% (6/8)	50% (12/24)
ChildTest	100% (1/1)	100% (4/4)	100% (11/11)
Main	0% (0/1)	0% (0/1)	0% (0/9)
Store	0% (0/1)	0% (0/2)	0% (0/14)
StoreTest	0% (0/1)	0% (0/1)	0% (0/1)

很棒，是很可观的进步！已经达到了50%的行覆盖率，再看看刚才红色那一部分

```
4 个用法
public void addOneFruit(String goal) {
    if (goal.equals("apple")) {
        appleCount++;
    } else if (goal.equals("banana")) {
        bananaCount++;
    }
}
```

已经变成绿色的了，代表我们测试覆盖了这个方法的所有行。

为什么方法的测试覆盖率也高了？不妨看看新的测试多了哪些方法？

接下来，按照这样的思路，你可以完成剩下的测试代码（见仓库文件）

但是需要注意的一点是，我们需要一些绝对保证正确的方法来构成测试可信度的基石：

```
public int getMoney() {
    return money;
}
```

这样简单的方法就是基石，但是假设基石不稳呢？

```
public int getMoney() {
    return 0;
}
```

这样基石不稳，那么今后的测试也就不具备任何可信度了！

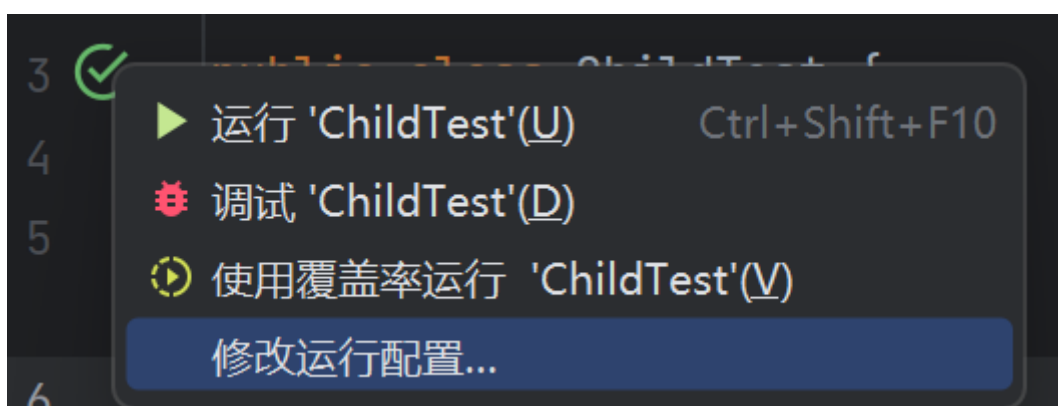
同时，如果一个方法在之前已经被我们测试好，那它也能被我们用来测试其他的方法：

```
public void eat() {
    child child =new child(20);
    child.eat("banana");
    assert (child.getBananaCount()==0);
    //是否没有判断就直接减了？
    child.addOneFruit("banana");
    //为了保证覆盖性，我们可能要测试一下这个意外条件
    child.eat("something");
    //这里的add方法在上面需要已经测试过正确性
    child.eat("banana");
    child.addOneFruit("apple");
    assert (child.getBananaCount()==0);
    child.eat("apple");
    assert (child.getAppleCount()==0);
}
```

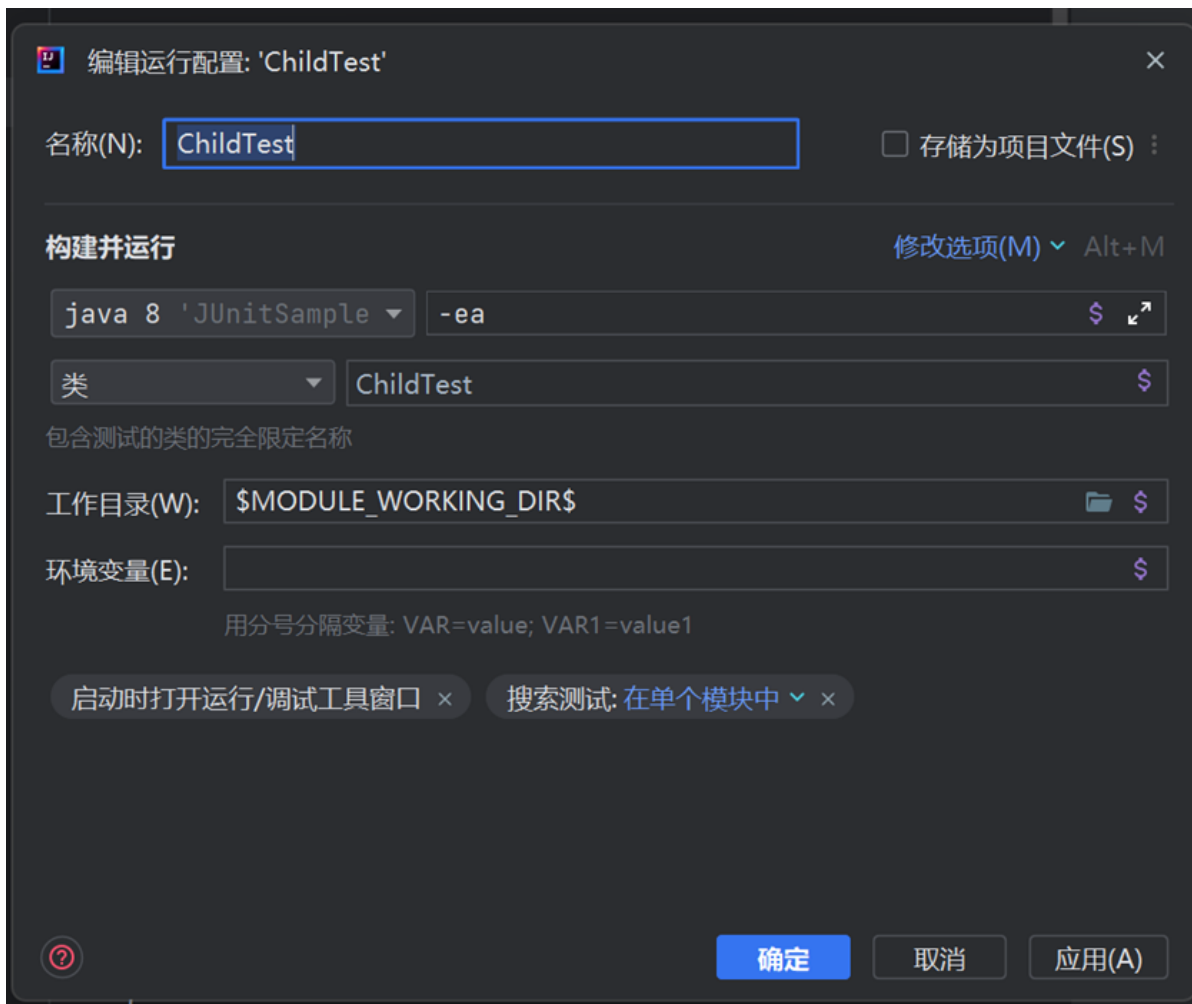
例如，在 `eat` 方法的测试中，我们可以利用已经测试好的方法 `addOneFruit()` 来进行 `eat` 方法的辅助测试。（什么地步是测试好呢？这个问题没有标准答案，但是很重要，还请同学们在以后的学习中体会）

添加分支覆盖率

我们上面提到，分支覆盖率是一种重要的测试评价依据。因此，下面我们展示在 IDEA 中如何添加 Unit 的分支覆盖率测试：



在刚才的几个选项中，我们选择了运行第一个和第三个选项，这里就需要用到第四个选项，修改运行配置。



(图中的配置请以自己电脑环境为准，一般不需在此步骤进行其他更改)

点击修改选项，点击后在栏中勾选使用跟踪（英文，Using tracing）：

添加运行选项

操作系统

允许多个实例 Alt+U

✓ 环境变量 Alt+E

Java

运行前不构建

✓ 添加 VM 选项 Alt+V

使用模块的类路径 Alt+O

修改类路径

缩短命令行

测试

重复 一次 >

复刻模式 无 >

日志

指定要在控制台中显示的日志

将控制台输出保存到文件

在消息打印到 stdout 时显示控制台

在消息打印到 stderr 时显示控制台

代码覆盖率

指定类和软件包

排除类和软件包

指定替代覆盖率运行程序

使用跟踪

在测试文件夹中收集覆盖率

启动前

添加启动前任务

✓ 启动时打开运行/调试工具窗口

开始前显示运行/调试配置设置

由于可以跟踪测试、查看覆盖率统计信息，
以及获得每个类要运行的其他信息，因此

确定

1=value1

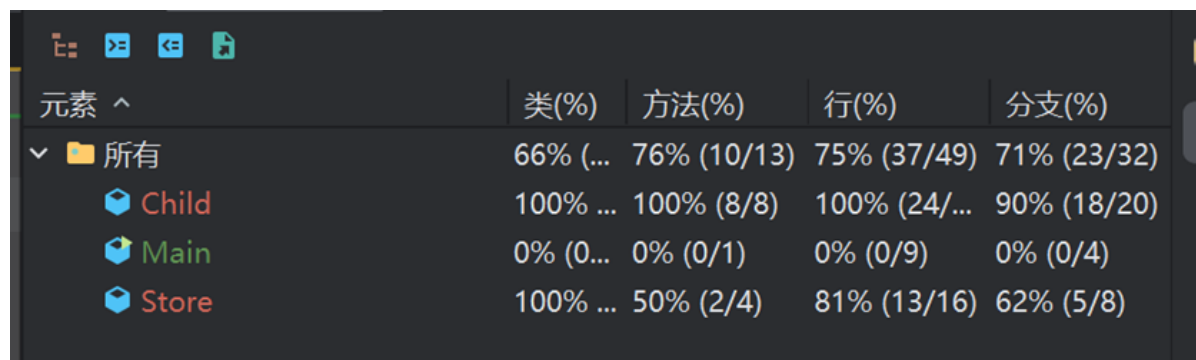
试: 在单个模块中 ✓ ×

money: 20);

);

Count()==0);

之后点击按照覆盖率运行，会发现多了一系列结果（如果未发现可以尝试重启IDEA）



元素 ^	类(%)	方法(%)	行(%)	分支(%)
所有	66% (...)	76% (10/13)	75% (37/49)	71% (23/32)
Child	100% ...	100% (8/8)	100% (24/...	90% (18/20)
Main	0% (0...	0% (0/1)	0% (0/9)	0% (0/4)
Store	100% ...	50% (2/4)	81% (13/16)	62% (5/8)

这样，我们就完成了几个必要的运行覆盖率的检验工作，你可以通过这些数据来评判你的测试代码。

如果想测试Store文件的话，我们也可以通过相似的思路来解决。

在oopre课程中，我们不需要设置main函数有关的JUnit测试，因此可以不必建立main的测试。

接下来，请用教程中的步骤和思想，对你的hw2程序进行JUnit编写吧！