

1.3 Singleton Pattern

单件模式

Debug

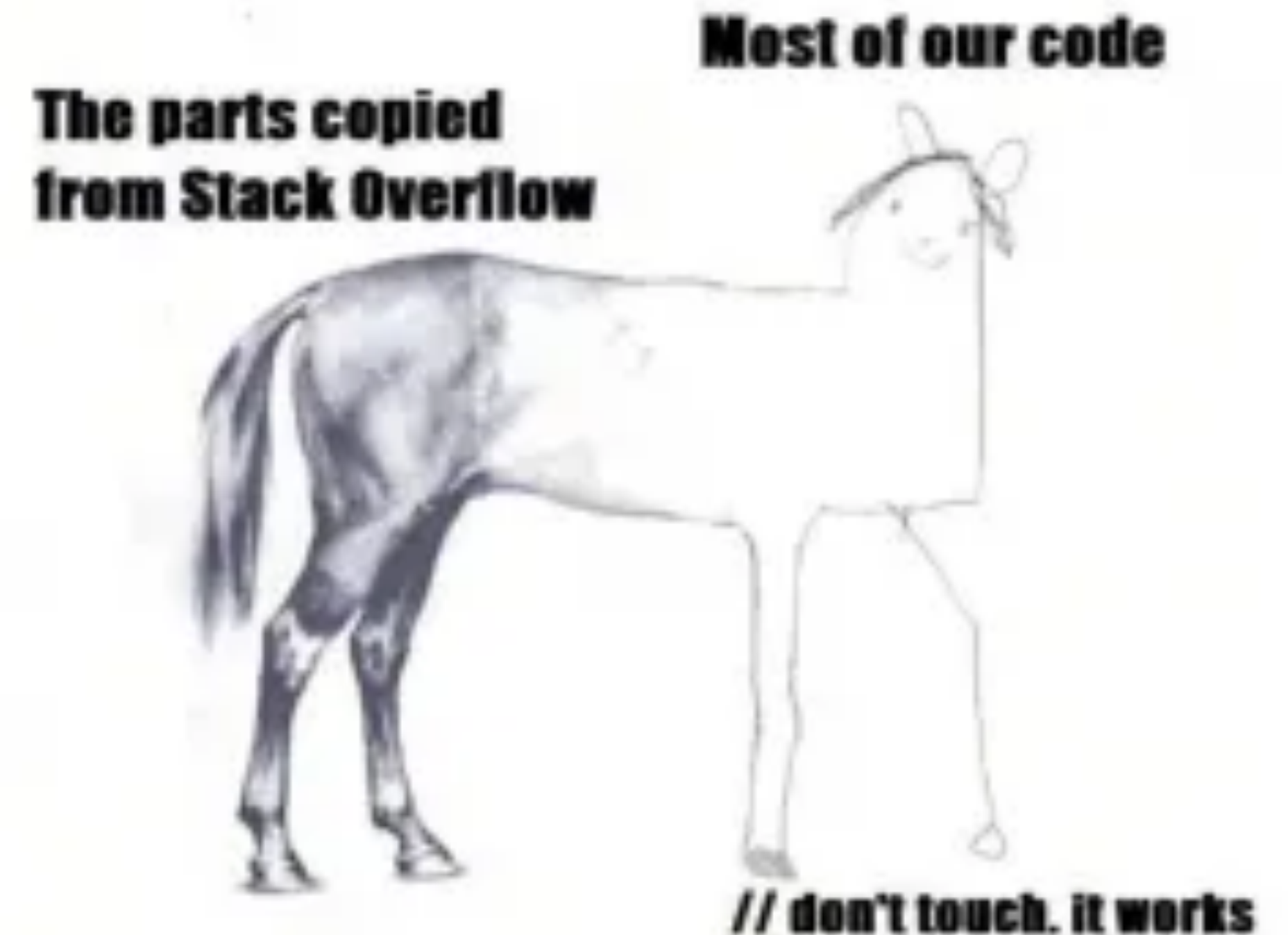
What is Bug?

Bug是什么？

软件 Bug 是指计算机程序中的一个缺陷、错误或故障，导致程序行为与预期不符

可以是：

- 程序崩溃；
- 输出错误结果；
- 用户操作失败；
- 功能未按预期执行；
- 甚至是体验不佳（如界面混乱、性能极差）



Bug

漏洞

Bug有五条判断规则：

- 1 - 软件未能实现需求文档中规定的功能
- 2 - 软件实现了需求文档中明确说不应该做的功能
- 3 - 软件做了需求文档中没有提及的事
- 4 - 软件没有实现虽然没有写，但是“应该有的”常识性功能
- 5 - 软件难以理解、难用、运行缓慢或用户体验极差

Source of Bug

Bug的根源

软件中的 Bug 往往不是偶然出现的，它们大多来自三个主要阶段：需求、设计、编码

Source of Bug

Bug的根源

需求（Specification）问题 —— 最常见的根源

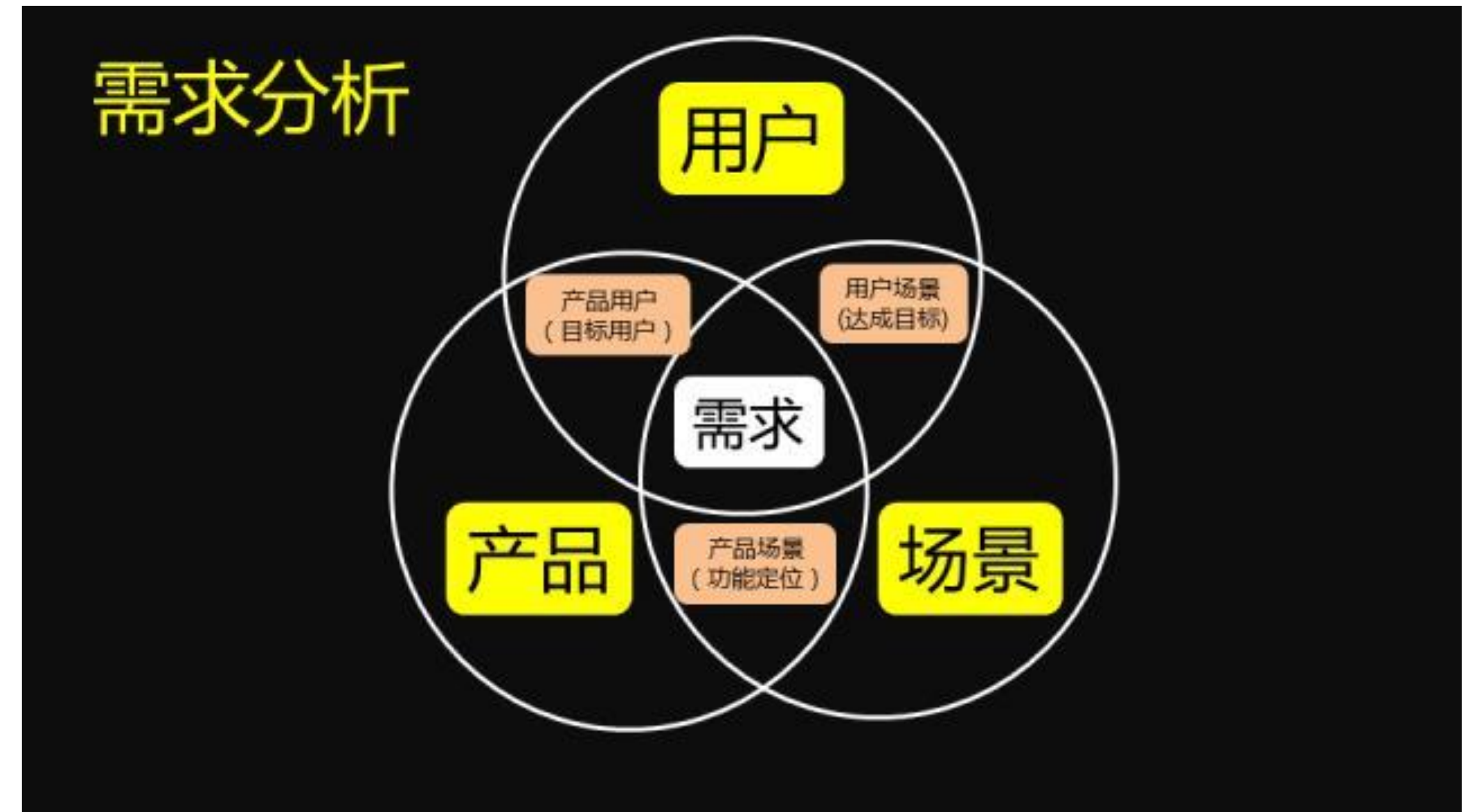
“写出来的东西错了，或者根本没写清楚。”

常见问题：

- 未写清楚（Unwritten）：重要功能或限制没有被提到；
- 模糊（**Ambiguous**）：描述模棱两可，不同人理解不同；
- 频繁变更（Changing specs）：文档今天改，明天又变，程序跟不上；
- 沟通不畅（Not Communicated）：开发团队根本没看到最新版本的需求。

示例：

用户需求说“支持多语言”，但没说明是哪几种语言，也没说是否自动识别，结果导致实现方向完全偏差。



Source of Bug

Bug的根源

设计（Design）问题

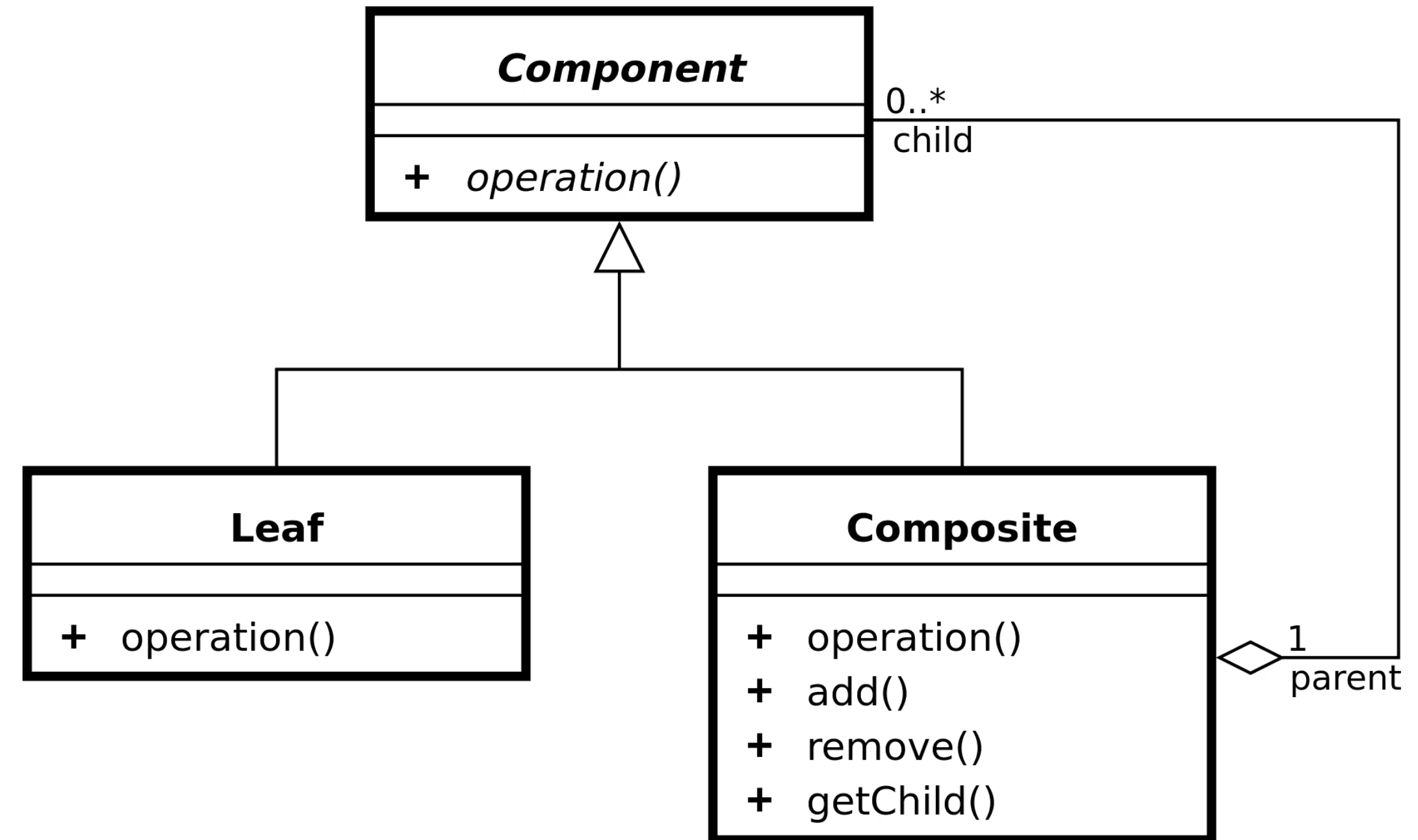
“思路错了，哪怕代码没错也会出问题。”

常见问题：

- 建模不当（Inappropriate modeling）：系统的核心抽象搞错了；
- 缺乏建模工具（No modeling tools）：设计随意，图都不画，靠脑补；
- 赶时间（Time-to-market pressure）：设计阶段被压缩，草草收场；
- **开发者轻视设计**：认为“能写代码就够了”，忽视结构设计。

示例：

在线商城将“用户订单”直接绑定数据库表，而不是设计成独立的逻辑层，结果每次订单更改都影响性能。



Source of Bug

Bug的根源

编码（Coding）问题

写错了代码

常见原因：

- 复杂度高（Complexity）：代码太绕，逻辑嵌套太深；
- 文档不足（Poor documentation）：别人看不懂，也难维护；
- 时间紧（Time pressure）：临时赶出来的代码容易埋雷；
- 开发者经验不足：对语言、框架、算法理解有限。

常见编码错误：

错误的条件判断（if 写反了）；覆盖方法错误；使用未初始化变量；
数据流错误；忘记释放资源；超出数组边界；逻辑重复或缺漏。



Software Reliability

软件可靠性

软件在特定时间、特定环境下无故障运行的概率。
不出问题的概率越高，可靠性越高

可靠性常见指标：

1. $R(t)$ ：
系统在时间 t 之前仍然正常运行的概率。
2. Availability（可用性）：
系统在任意时刻处于可服务状态的比例，常见如 99.9%。
3. MTBF（Mean Time Between Failures）：
平均多长时间系统才会出现一次故障。



Break Point Debug

断点调试

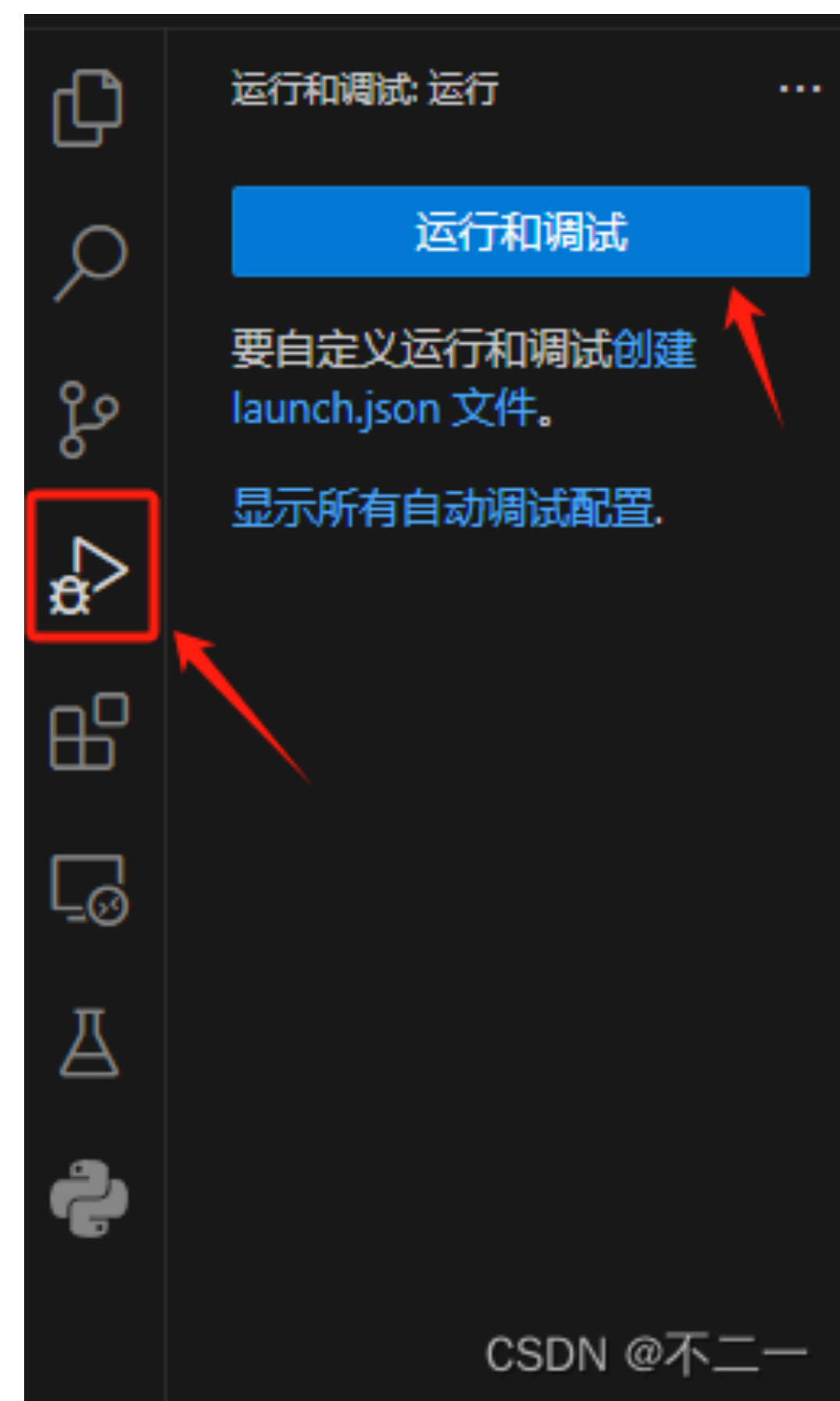
1. 启动调试

此处以Visual Studio Code为例

点击图标，或者

按Ctrl + Shift + D 进入窗口

点击开始运行和调试



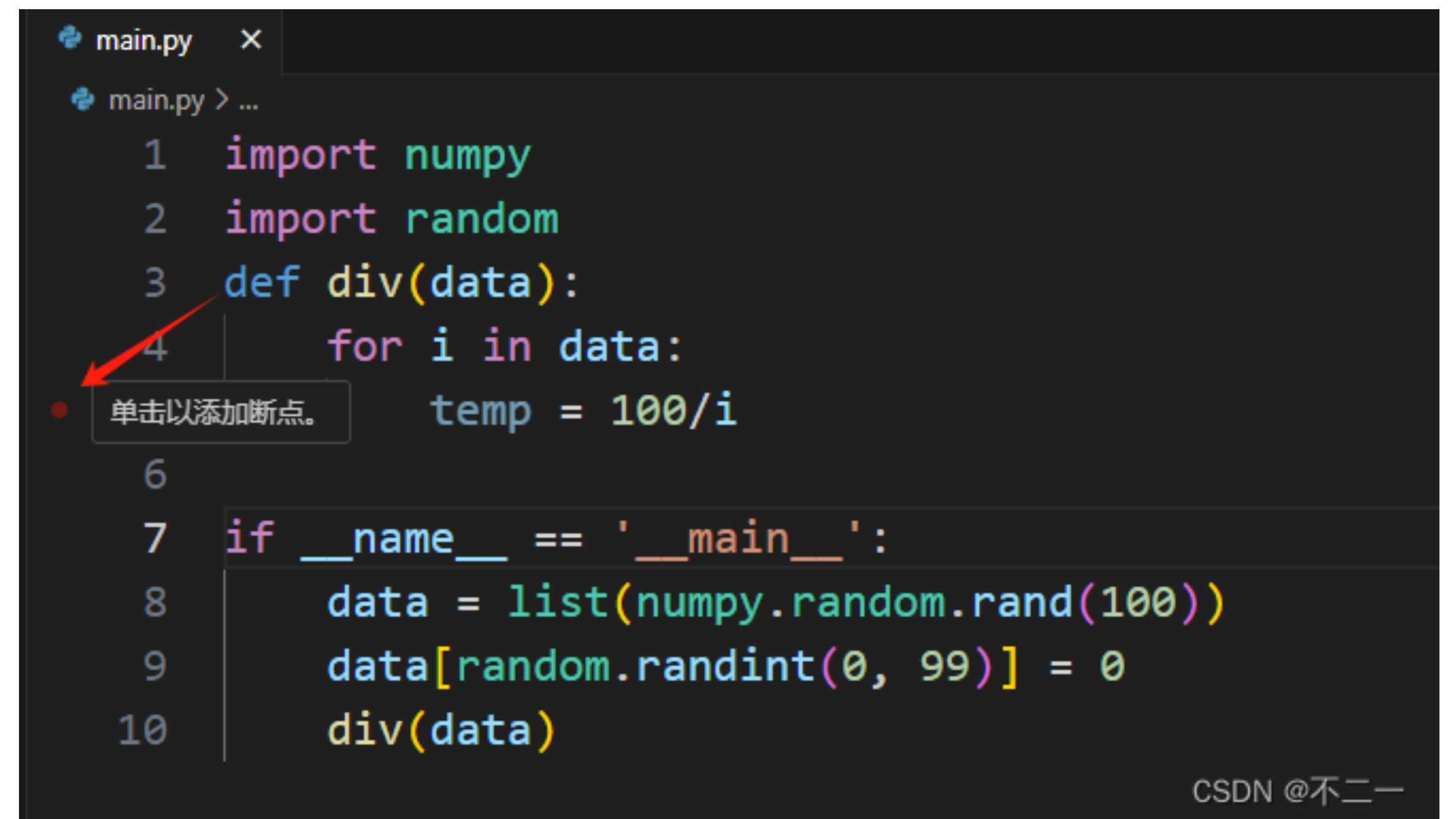
Break Point Debug

断点调试

2. 标记断点

断点即程序在调试模式运行时会暂时停在此处

便于我们观察行为和变量变化



The screenshot shows a code editor with a file named `main.py`. The code is as follows:

```
1 import numpy
2 import random
3 def div(data):
4     for i in data:
5         temp = 100/i
6
7 if __name__ == '__main__':
8     data = list(numpy.random.rand(100))
9     data[random.randint(0, 99)] = 0
10    div(data)
```

A red arrow points to the left margin next to line 5, where a break point (a small red dot) has been added. A tooltip box next to the dot contains the text "单击以添加断点。" (Click to add breakpoint.).

CSDN @不二一

Break Point Debug

断点调试

3. 调试工具

Continue - 执行到下一个断点

Step Over - 执行当前行，跳过方法内部

Step Into - 进入当前方法内部

Step Out - 跳出当前方法，返回调用点

Restart - 从头开始程序

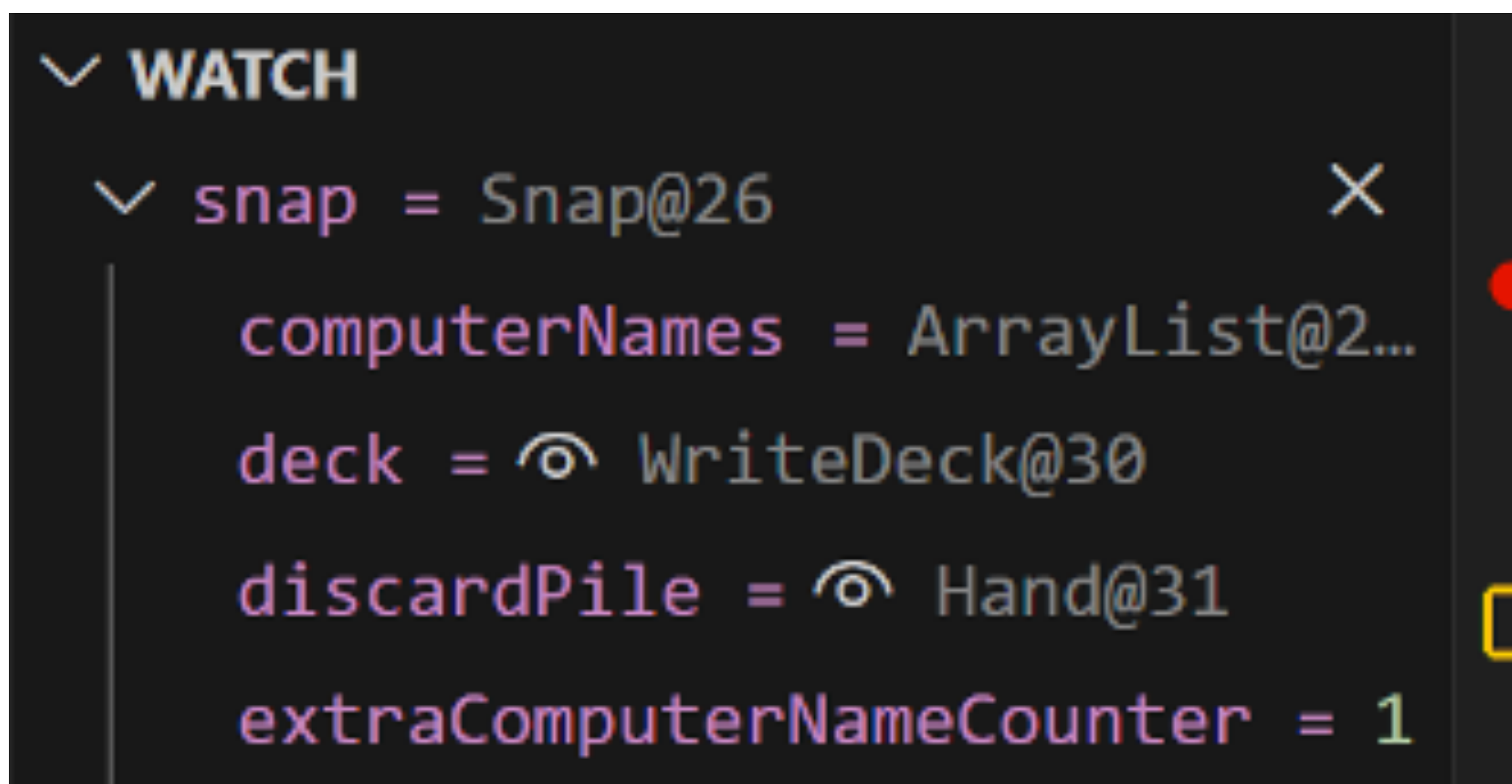
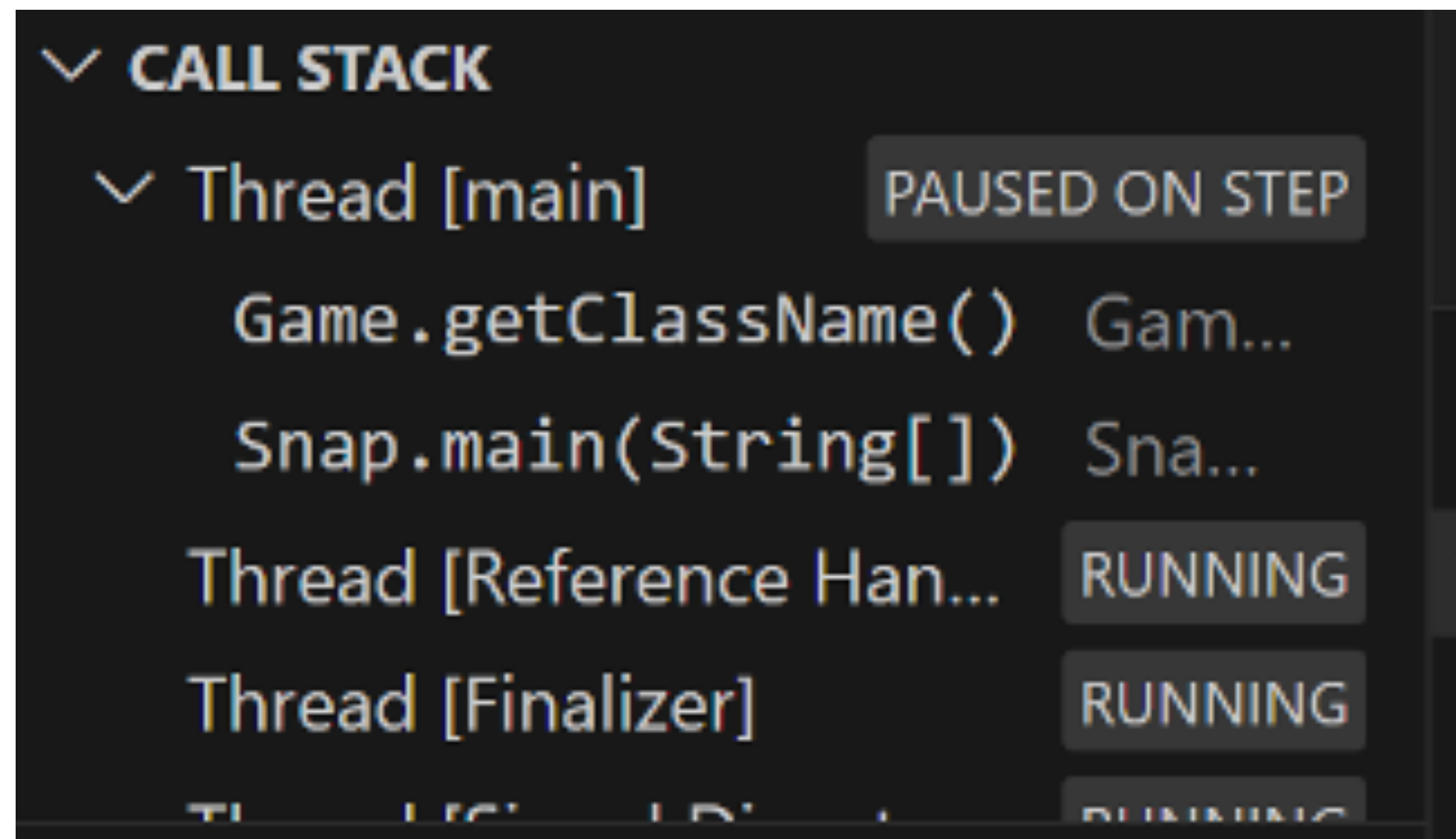
Stop - 终止程序运行



Break Point Debug

断点调试

4. 观察工具



Debug Strategy

调试策略

- 1. 定位 (Locate)** - 是在哪行报错？这行属于哪个方法哪个类？是在哪里被调用？是否设计其他的方法和类？
- 2. 理解 (Understand)** - 什么类型的错误？错误导致了什么？原本的预期是什么？
- 3. 分析 (Analyze)** - 是哪个判断出错？哪个输入未预期？是否假设错了？原始错误来源是什么？为什么出错？
- 4. 修复 (Fix)** - 修改出错的逻辑或调用，添加边界判断，修改数据传递，优化或添加异常处理，必要时修改设计和需求
- 5. 验证 (Verify)** - 测试修复后程序是否正常运行，如果不是就从第一步重新开始

Debug - Practice

Debug调试 - 练习

场景：零售店的自动折扣计算系统

某个零售店希望开发一个简单的程序，自动计算每件商品的**最终售价**。这家店有一个特殊的促销策略：

- 所有商品都存储在一个价格数组中；
- 商品按顺序排列，位置越靠后，折扣越大；
- 折扣按照商品在数组中的位置编号乘以 10% 计算：
 - 第 0 个商品：0% 折扣
 - 第 1 个商品：10% 折扣
 - 第 2 个商品：20% 折扣
 -以此类推；
- 对于价格超过 100 元的商品，系统希望能进一步通过“自动递减折扣”的方式优化价格，以刺激销售。

Debug - Practice

Debug调试 - 练习

预期逻辑流程

1. 遍历商品列表：
 - 从第一个商品到最后一个商品，逐个处理。
2. 根据编号计算折扣百分比：
 - 使用当前商品的索引 i 计算折扣为 $i * 10\%$ 。
3. 调用价格计算函数：
 - 如果折扣为 0% ，可按原价输出；
 - 若商品价格大于 100 元，则进一步调整价格；
 - 否则直接根据折扣百分比计算最终售价。
4. 输出每件商品的最终价格。

Debug - Practice

Debug调试 - 练习

预期结果（理想情况下输出）

以以下数组作为输入：

```
double[] productPrices = {120.0, 85.0, 150.0, 60.0};
```

预期输出应类似于：

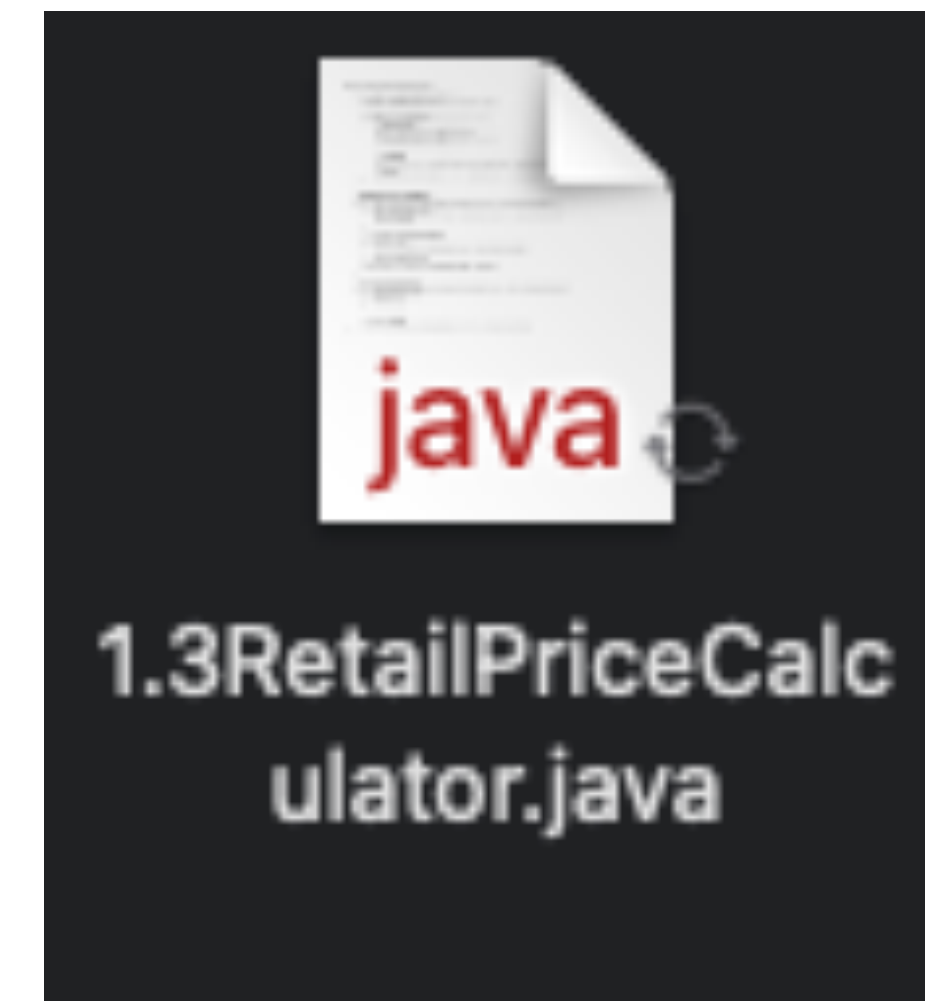
```
Product 0 final price: 120.0
```

```
Product 1 final price: 76.5
```

```
Product 2 final price: 120.0    // 原为 150 元，经过折扣调整
```

```
Product 3 final price: 42.0
```

（注：具体数值视折扣/调整函数实现而定）



Singleton Pattern

单件模式

Singleton Pattern

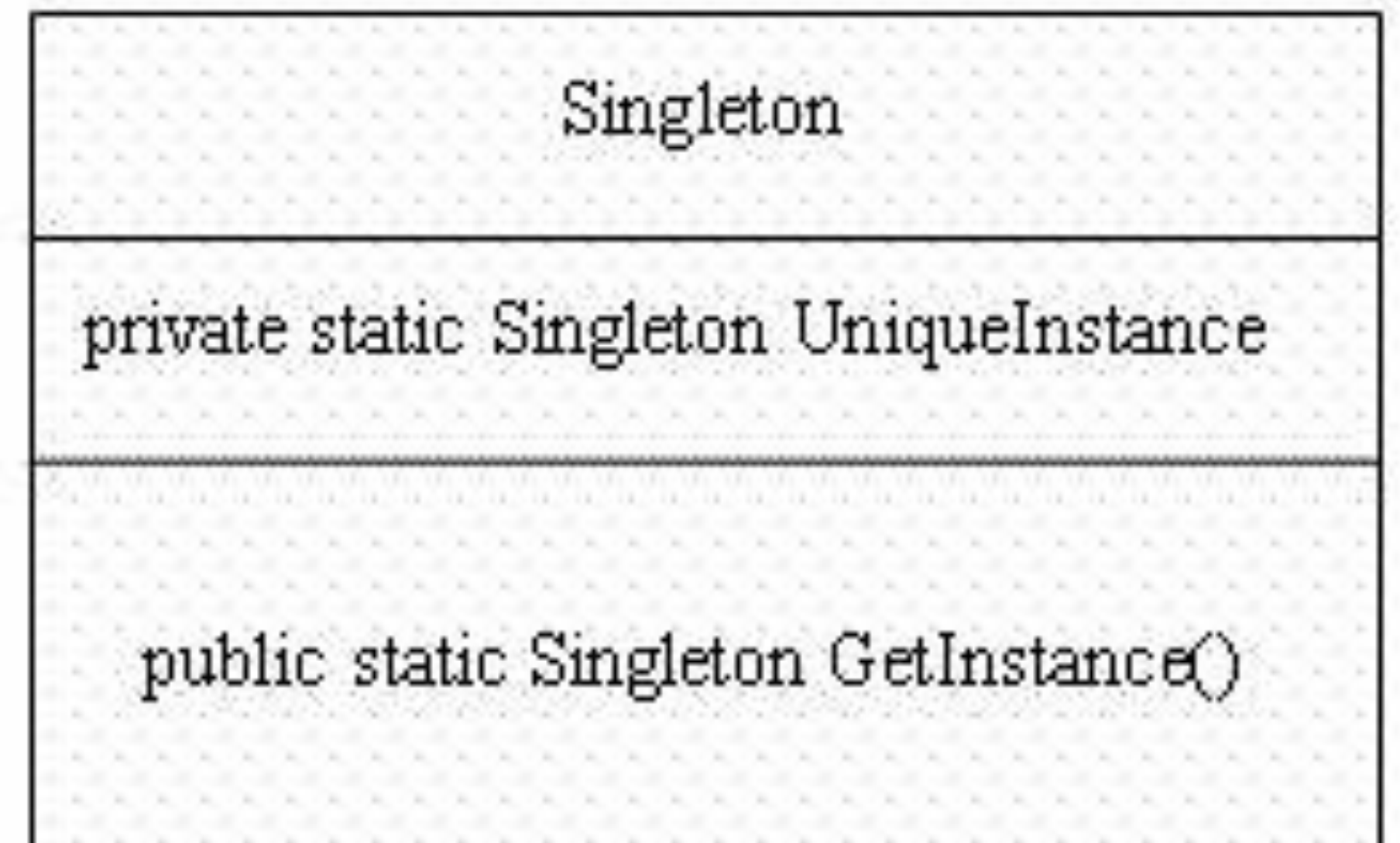
单件模式

确保一个类在系统中只有一个实例，并提供一个全局访问点来获取它。

设想这样一些场景：

- 日志类（全局唯一，不希望多个文件操作同一个日志文件）
- 数据库连接池（只需一个共享实例）
- 配置管理器（配置文件加载一次，供全系统使用）
- 线程池、缓存、驱动管理等系统资源控制器

这些对象有一个共同点：**只需要一个**，重复创建不仅浪费资源，还可能出错。



Singleton Pattern - Design

单件模式的设计

1. 饿汉式（线程安全，类加载时创建实例）
2. 懒汉式（延迟加载，线程不安全）
3. 懒汉式+线程同步（同步，性能较差）
4. 双重检查锁定（性能+线程安全）

Thread Safety

线程安全

多个线程同时访问某个资源或执行某段代码时，程序仍能按预期正确运行，不出现数据错乱、崩溃或不一致的结果。

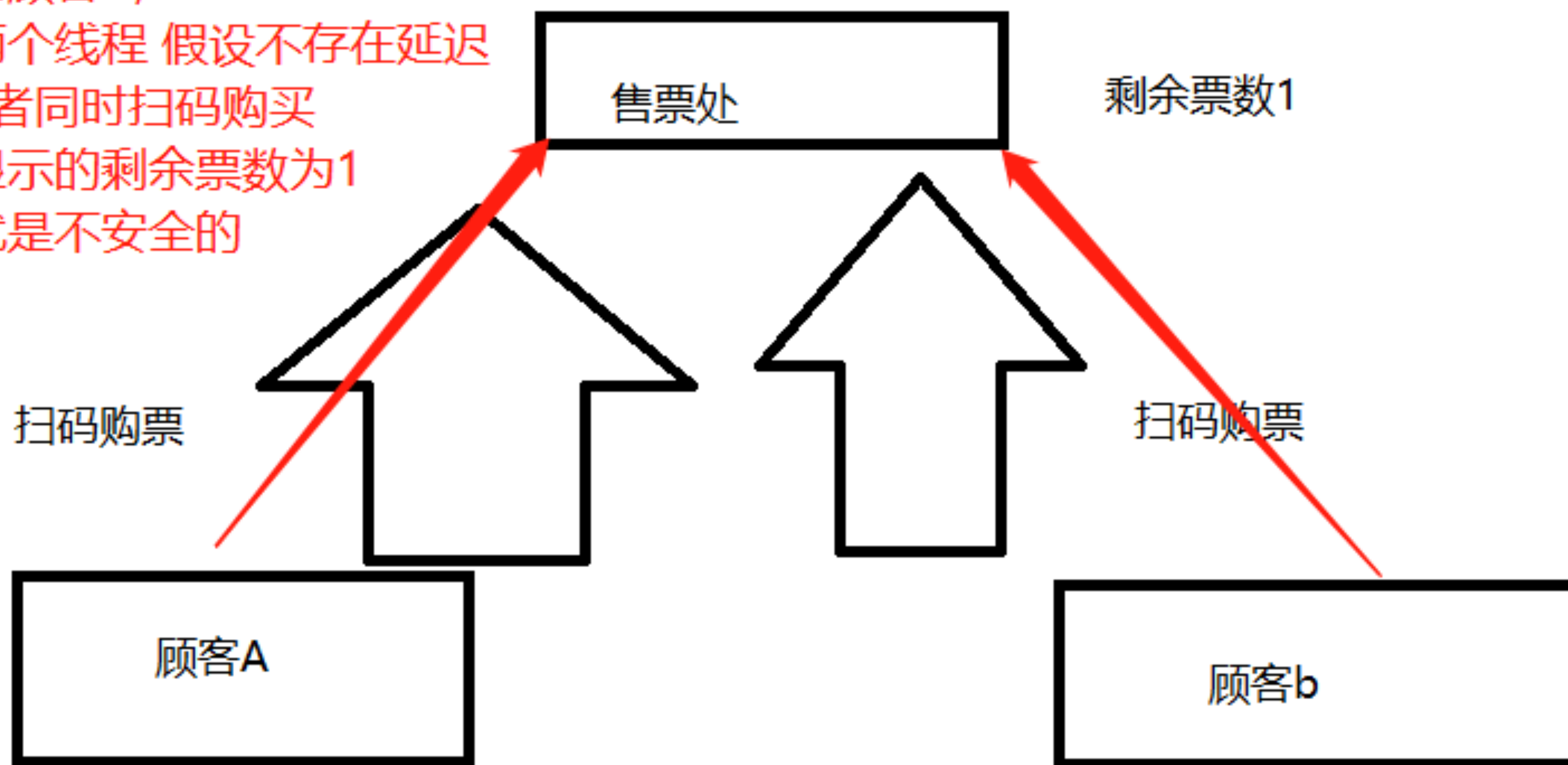
多个线程“同时读写共享资源”，而没有适当的同步机制，就会发生如下问题：

- 数据竞争（Race Condition）
- 脏数据 / 中间状态
- 程序行为不确定、难以复现

Thread Safety

线程安全

如果把顾客A,B
看作两个线程 假设不存在延迟
A,B两者同时扫码购买
那么显示的剩余票数为1
这样就是不安全的



Thread Safety

线程安全

```
public class Counter {  
    private int count = 1;  
  
    public synchronized void decrement() {  
        count--;  
    }  
  
    public synchronized int getCount() {  
        return count;  
    }  
}
```

两个方法加了 `synchronized`，确保同一时间只有一个线程能进入方法，因此是线程安全的。

Singleton Pattern - Design 1

饿汉式设计

```
public class Singleton {  
    // 类加载时立即创建实例  
    private static final Singleton instance = new Singleton();  
  
    // 构造函数私有化  
    private Singleton() {}  
  
    // 提供全局访问点  
    public static Singleton getInstance() {  
        return instance;  
    }  
  
    public void showMessage() {  
        System.out.println("Hello from Singleton!");  
    }  
}
```

Singleton Pattern - Design 2

懒汉式（延迟加载，线程不安全）

```
public class Singleton {  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton(); // 多线程时可能创建多个实例!  
        }  
        return instance;  
    }  
}
```

Singleton Pattern - Design 3

懒汉式 + 线程安全（同步，性能较差）

```
public class Singleton {  
    private static Singleton instance;  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton(); // 线程安全，但性能较差  
        }  
        return instance;  
    }  
}
```

Singleton Pattern - Design 4

双重检查锁定（推荐方案，性能+线程安全）

```
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized(Singleton.class) {
                if (instance == null) {
                    instance = new Singleton(); // 创建实例
                }
            }
        }
        return instance;
    }
}
```

为什么是线程安全的？

1: 加锁 + 再检查

- 第一次 `if (instance == null)` 是为性能优化的快速路径（大多数情况下直接返回实例）
- 进入 `synchronized` 后再判断一次 是为了避免多个线程在第一次判断都为 `true` 的情况下创建多个实例

保证：只有一个线程能真正创建实例，其他线程等到锁释放后，检查 `instance != null`，直接返回已有实例。

2: 使用 `volatile` 关键字

- `volatile` 禁止指令重排序（JVM可能把 `new` 的三步拆开顺序执行）：

`instance = new Singleton();`

// 实际上是三个步骤：

1. 分配内存空间
2. 调用构造函数
3. 将对象引用赋值给 `instance`

- 没有 `volatile` 的话，可能出现：
 - Thread A 执行了 `instance = new Singleton()` 的 **第3步**
 - Thread B 读取到了非 `null` 的 `instance`，但实际还没完成构造 → **可能使用未初始化的对象**

`volatile` 保证了构造完成之前，其他线程不可见该对象引用，从而避免线程安全问题。

Singleton Pattern - Design 4

双重检查锁定（推荐方案，性能+线程安全）

```
public class Singleton {  
    private static volatile Singleton instance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized(Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton(); // 创建实例  
                }  
            }  
        }  
        return instance;  
    }  
}
```

优点

保证只有一个实例存在

不易扩展为多实例版本

提供全局访问点

缺点

多线程处理时需要注意并发和效率问题

可延迟加载，提高资源利用效率

可能隐藏依赖，使得代码更难测试

Singleton Pattern Useage

单例模式的调用

```
public class Main {  
    public static void main(String[] args) {  
        Singleton singleton = Singleton.getInstance();  
        singleton.showMessage(); // 输出: Hello from Singleton!  
    }  
}
```

```
public class Singleton {  
    private static volatile Singleton instance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (instance == null) {  
            synchronized(Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton(); // 创建实例  
                }  
            }  
        }  
        return instance;  
    }  
}
```


Singleton Pattern - Example

单例模式 - 示例

```
public class Logger {
    private static volatile Logger instance = null;
    private final StringBuilder logs = new StringBuilder();

    // 私有构造函数
    private Logger() {}

    public static Logger getInstance() {
        if (instance == null) {
            synchronized(Logger.class) {
                if (instance == null) {
                    instance = new Logger();
                    System.out.println("Logger instance created");
                }
            }
        }
        return instance;
    }

    // 同步 log 方法, 保证线程安全
    public synchronized void log(String message) {
        logs.append(message).append("\n");
    }

    public synchronized String getLogs() {
        return logs.toString();
    }
}
```

```
public class TestLogger {
    public static void main(String[] args) {
        Runnable task = () -> {
            Logger logger = Logger.getInstance();
            logger.log(Thread.currentThread().getName() + "
started");
        };

        Thread[] threads = new Thread[5];
        for (int i = 0; i < 5; i++) {
            threads[i] = new Thread(task, "Thread-" + i);
            threads[i].start();
        }

        for (int i = 0; i < 5; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        Logger logger = Logger.getInstance();
        System.out.println("Logs collected:\n" +
logger.getLogs());
    }
}
```

Singleton Pattern - Practice

单例模式 - 练习题

你正在开发一个桌面应用程序，该程序允许用户修改一些全局设置，比如：

- 语言 (language)
- 主题 (theme)
- 是否启用通知 (notificationsEnabled)

由于这些设置对整个应用程序是共享的，你需要设计一个类来统一管理这些设置，而且只能存在一个实例。

需求：

1. 创建一个 `SettingsManager` 类，使用线程安全的单件模式。
2. 提供以下方法：
 - `setLanguage(String lang)`
 - `setTheme(String theme)`
 - `setNotificationsEnabled(boolean enabled)`
 - `printSettings()`: 打印当前设置
3. 编写测试类 `TestSettings`，模拟多个模块同时设置参数并查看是否是同一个对象。



示例输出：

```
SettingsManager instance created
[Thread-A] 设置语言为: English
[Thread-B] 设置主题为: Dark
[Main] 当前设置:
Language: English
Theme: Dark
Notifications: true
```

这能说明多个线程操作的其实是**同一个实例**。

提示：

- 单件模式用 `private static volatile` 和双重检查锁
- 属性设置可以加 `synchronized` 保证线程安全（或者使用 `ConcurrentHashMap` 等结构）

Composite Design Pattern

组合模式

Composite Design Pattern

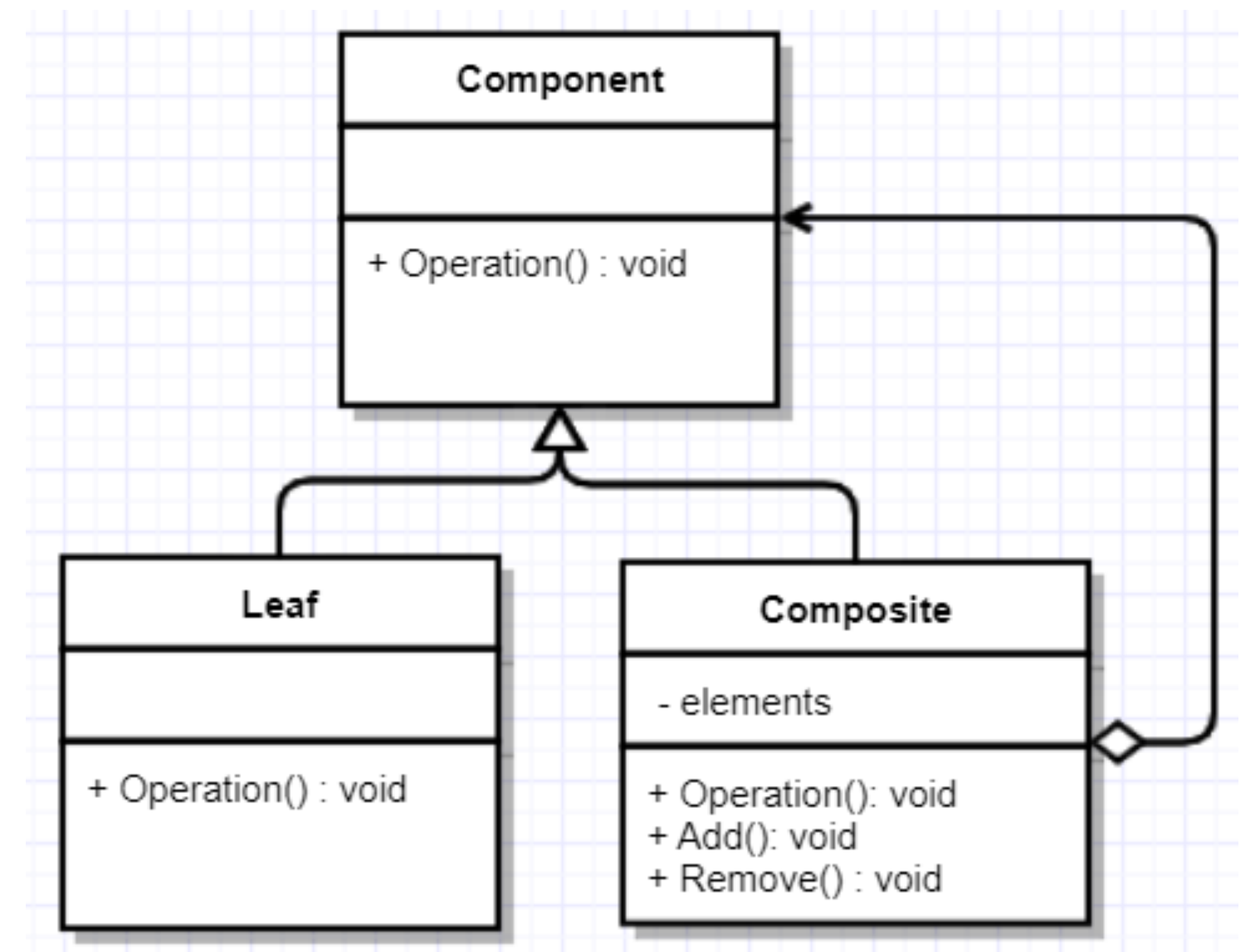
组合模式

将对象组合成树形结构以表示“部分-整体”的层次结构。
Composite 模式使得用户对单个对象和组合对象的使用具有一致性。

Component（组件） - 抽象类或接口，定义组合中对象的共同接口

Leaf（叶子节点） - 表示组合中的基本对象，不能有子节点

Composite（组合节点） - 包含子节点，可以添加、移除、访问子组件，实现递归结构



Composite Design Pattern - Example

组合模式 - 示例

例如一个文件系统Component

Composite是文件夹

Leaf是文件

Composite下面可以是Leaf，也可以是Composite

Composite: "根文件夹"

├── Leaf: "简历.docx"

├── Composite: "项目文件夹"

| ├── Leaf: "代码.java"

| └── Leaf: "说明.txt"

└── Leaf: "照片.jpg"

Composite Design Pattern - Example

组合模式 - 示例

抽象组件：FileSystemComponent

```
public abstract class FileSystemComponent {  
    protected String name;  
  
    public FileSystemComponent(String name) {  
        this.name = name;  
    }  
  
    public abstract void display(String indent);    //统一接口  
}
```

Composite Design Pattern - Example

组合模式 - 示例

叶子节点: File

```
public class File extends FileSystemComponent {  
  
    public File(String name) {  
        super(name);  
    }  
  
    @Override  
    public void display(String indent) {  
        System.out.println(indent + "File: " + name);  
    }  
}
```


Composite Design Pattern - Example

组合模式 - 示例

组合节点：Folder

```
import java.util.ArrayList;
import java.util.List;

public class Folder extends FileSystemComponent {
    private List<FileSystemComponent> children = new ArrayList<>();

    public Folder(String name) {
        super(name);
    }

    public void add(FileSystemComponent component) {
        children.add(component);
    }

    public void remove(FileSystemComponent component) {
        children.remove(component);
    }

    @Override
    public void display(String indent) {
        System.out.println(indent + "Folder: " + name);
        for (FileSystemComponent child : children) {
            child.display(indent + "    ");
        }
    }
}
```

Composite Design Pattern - Example

组合模式 - 示例

客户端: Main

```
public class Main {  
    public static void main(String[] args) {  
        FileSystemComponent file1 = new File("Resume.docx");  
        FileSystemComponent file2 = new File("Photo.jpg");  
        FileSystemComponent file3 = new File("Notes.txt");  
  
        Folder folder1 = new Folder("Documents");  
        folder1.add(file1);  
        folder1.add(file3);  
  
        Folder root = new Folder("Desktop");  
        root.add(folder1);  
        root.add(file2);  
  
        root.display("");  
    }  
}
```

Composite Deign Pattern

组合模式

优点

- 清晰表达 层级结构
- 客户端对单个对象与组合对象的使用一致
- 易于添加新类型的组件（符合开放封闭原则）

缺点

- 增加了设计的抽象性和复杂性
- 可能会为叶子对象引入不必要的方法（透明性带来的代价）

Composite Pattern Example

组合模式实现案例

实现案例

描述

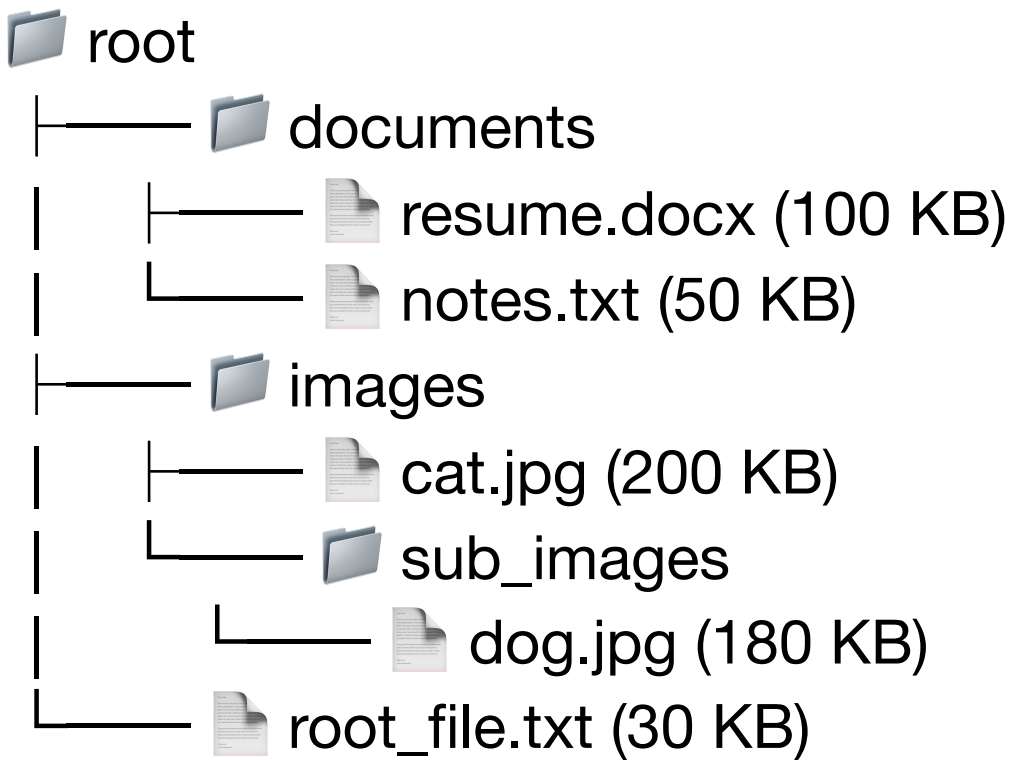
实现一个文件系统，包含两种类型的节点：

- File（文件）：有 name, size, type（如 .txt, .jpg）
- Folder（文件夹）：有 name，可以包含任意数量的 File 和 Folder

功能

1. 定义抽象类 FileSystemNode，包含：
 - String name
 - 抽象方法 long getSize()：返回大小（文件直接返回，文件夹返回递归总大小）
 - 抽象方法 void display(String indent)：层次结构显示自身及其子节点
2. 文件类 File：
 - 属性：long size, String type
 - getSize() 返回文件大小
 - display() 显示为  文件名.类型（大小）
3. 文件夹类 Folder：
 - 拥有子节点 List<FileSystemNode>
 - 方法：
 - void add(FileSystemNode child)
 - long getSize()：返回所有子节点的总大小
 - display()：递归显示文件夹和其中的文件/子文件夹
 - **List<File> searchByType(String type)：递归查找所有后缀为该类型的文件，例如 ".jpg"**

4. 编写 Main 函数构建如下结构并测试功能：



要求：

- 调用 root.display("") 显示结构
- 调用 root.getSize() 显示总大小
- 调用 root.searchByType(".jpg") 找出所有 jpg 文件并打印名称与大小