13.debug（调试）

当出现bug时，调试过程分为5步

1.复现bug

这个听起来好像很蠢，但是在多线程，图形化等情况下，复现bug很有用

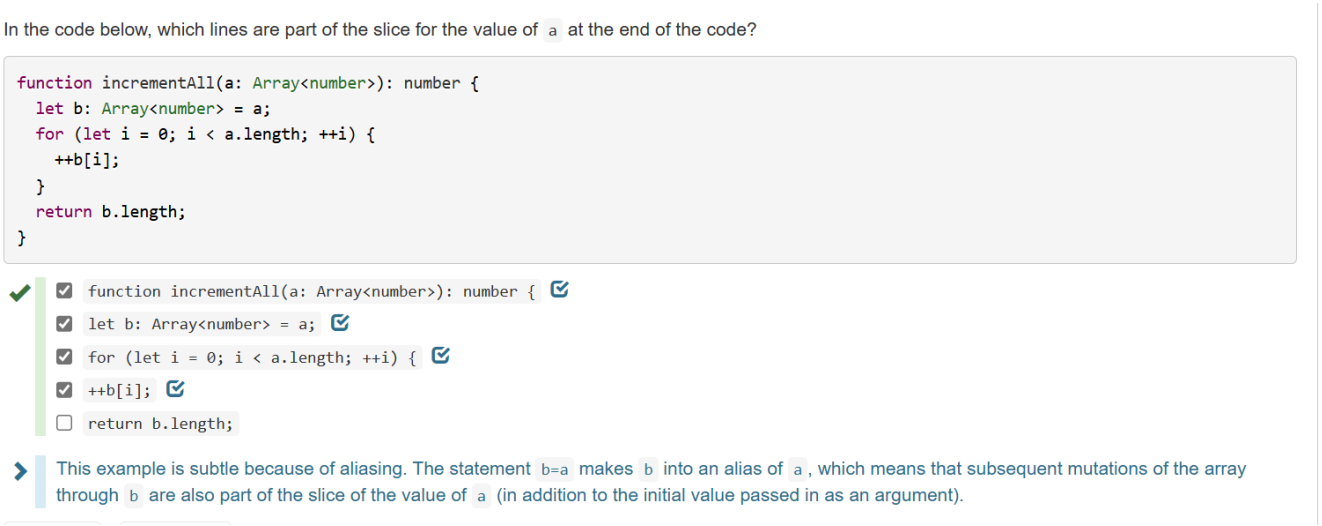
2.研究数据

一种重要的数据形式是异常产生的堆栈跟踪信息

在典型的堆栈跟踪信息中，最新的调用在顶部，最早的调用在底部。但堆栈顶部或底部的调用可能是你没有编写过的库代码。你自己的代码 —— 也就是最有可能存在 Bug 的地方 —— 通常在中间的某个位置。别因此而退缩。仔细浏览堆栈跟踪信息，直到看到熟悉的内容，然后在你的代码中找到它

3.假设

假设一个部分出错，然后**切片分析**与之相关的内容，例子：



还有一种假设后的处理方式是**差异调试**

代码示例层面的差异调试

假设我们有一个简单的函数 `calculateSum` 用于计算数组中所有元素的和：

收起

typescript

```
function calculateSum(arr: number[]): number {
    let sum = 0;
    for (let i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

现在我们有两个测试用例：

- **测试用例 1**：`calculateSum([1, 2, 3])`，预期结果是 `6`，实际运行结果也是 `6`，测试通过。
- **测试用例 2**：`calculateSum([1, '2', 3])`，预期结果应该是 `6`，但实际运行时可能会抛出类型错误，测试失败。

我们对这两个测试用例进行差异调试：

1. 首先分析输入差异，测试用例 1 的输入是纯数字数组 `[1, 2, 3]`，测试用例 2 的输入 `[1, '2', 3]` 中有一个字符串类型的元素。
2. 接着查看代码执行过程，在 `calculateSum` 函数中，`sum += arr[i];` 这一行代码在处理 `[1, '2', 3]` 时，由于 `'2'` 是字符串，在进行加法运算时会导致类型错误。因为在 JavaScript/TypeScript 中，数字和字符串相加会进行隐式类型转换，这里可能不是我们期望的计算方式。
3. 通过这样对比两个测试用例执行过程的差异，我们可以得出假设：Bug 出在对输入数组元素类型的检查和处理上。为了解决这个问题，可以在函数开头添加对数组元素类型的检查，例如：

收起

typescript

```
function calculateSum(arr: number[]): number {
    for (let element of arr) {
        if (typeof element!== 'number') {
            throw new Error('数组元素必须为数字');
        }
    }
    let sum = 0;
    for (let i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
```

```
    return sum;
}
```

版本控制层面的差异调试

假设我们在开发一个小型的 Web 应用程序，使用 Git 作为版本控制系统。

1. 一开始，程序运行正常，我们提交了一个版本 `A` 。
2. 然后我们添加了一个新功能，提交了版本 `B` ，但在测试时发现某个页面的功能出现了 Bug。
3. 我们使用 `git bisect` 命令（这是 Git 中用于二分查找引入 Bug 提交的工具）来进行差异调试。首先标记当前失败的版本 `B` 为 `bad` ，然后找到之前已知的正常版本 `A` 并标记为 `good` 。
4. Git 会自动选择一个中间版本 `C` 进行测试。如果 `C` 版本正常，那么说明 Bug 是在 `C` 到 `B` 之间引入的；如果 `C` 版本失败，那么 Bug 就在 `A` 到 `C` 之间。
5. 假设 `C` 版本失败，我们再将 `C` 标记为 `bad` ，Git 又会在 `A` 和 `C` 之间选择一个新的版本 `D` 进行测试。
6. 重复这个过程，不断缩小范围，最终找到引入 Bug 的那个具体提交。比如，经过几次测试，发现是在提交 `E` 中，开发人员不小心修改了某个函数的参数传递方式，导致了页面功能的 Bug。

通过以上两种方式的差异调试示例，可以更高效地定位和解决程序中的 Bug。

4.实验验证

1.可以在多个位置写输出语句，然后看哪里没有输出语句，或者语句不对
2.用断言
3.组件替换

如果你假设 Bug 存在于某个模块中，并且恰好有该模块的另一种实现方式，且这种实现满足相同的接口，那么你可以进行的一个实验就是尝试替换该模块。例如：

- 如果你怀疑 binarySearch () 的实现有问题，那么可以用更简单的 linearSearch () 来替代。
- 如果你怀疑 JavaScript 运行时环境有问题，可以在不同的网页浏览器或不同版本的 Node 环境中运行程序。
- 如果你怀疑操作系统有问题，可以在不同的操作系统上运行程序。
- 如果你怀疑硬件有问题，可以在不同的机器上运行程序。

然而，替换本身没有问题的组件会浪费大量时间，所以除非你有充分的理由怀疑某个组件，否则不要这么做。正如我们在 “对假设进行优先级排序” 中所讨论的，编程语言、操作系统或硬件在你的怀疑对象列表中应该排在非常靠后的位置。

5.重复上述方法，进行迭代