15.相等性

很好理解，就是两者等价

- **自反性**：(t, t) ∈ E ∀ t ∈ T （对于所有 t 属于 T，(t, t) 都属于 E)
- **对称性**：(t, u) ∈ E ⇒ (u, t) ∈ E （如果 (t, u) 属于 E，那么 (u, t) 也属于 E)
- **传递性**：(t, u) ∈ E ∧ (u, v) ∈ E ⇒ (t, v) ∈ E （如果 (t, u) 属于 E 且 (u, v) 属于 E，那么 (t, v) 属于 E)

不变量的等价

对于不变量的等价，有两种
1.抽象函数的等价

Using the abstraction function. Recall that an abstraction function AF: R → A maps concrete instances of a data type to their corresponding abstract values. To use AF as a definition for equality, we would say that a equals b if and only if AF(a) = AF(b).

2.观察等价

Using observation. We can say that two objects are equal when they cannot be distinguished by observation – every operation we can apply produces the same result for both objects. Consider the set expressions {1,2} and {2,1}. Using the observer operations available for sets, cardinality |…| and membership ∈, these expressions are indistinguishable:

- |{1,2}| = 2 and |{2,1}| = 2
- 1 ∈ {1,2} is true, and 1 ∈ {2,1} is true
- 2 ∈ {1,2} is true, and 2 ∈ {2,1} is true
- 3 ∈ {1,2} is false, and 3 ∈ {2,1} is false
- … and so on

引用相等性和值相等性

绝大多数语言都有两种相等性的判断：

- **引用相等性**：用于测试两个引用是否指向内存中的同一存储位置。从我们一直绘制的快照图角度来看，如果两个引用的箭头指向同一个对象 “气泡”，那么通过引用相等性判断它们是相等的。
- **值相等性**：用于测试两个对象是否表示相同的值，换句话说，就是我们在本次阅读材料中一直在讨论的相等性概念。

语言	引用相等性	值相等性
Python	is	==
Java	==	equals()
Objective - C	==	isEqual:
C#	==	Equals()
TypeScript/JavaScript	===	无内置方法

可变类型的相等性

- **观察相等性**：意味着在程序的当前状态下，两个引用无法被区分。客户端只能通过调用那些不会改变任何一个对象状态的操作（即仅调用观察操作（observers）和生产操作（producers），而不调用变更操作（mutators）），并比较这些操作的结果，来尝试区分它们。这用于测试两个引用在对象当前状态下是否 “看起来” 相同。
- **行为相等性**：意味着在现在以及未来，两个引用都无法被区分，即使对其中一个对象调用变更操作来改变其状态，而另一个对象状态不变。这用于测试在当前以及所有未来状态下，这两个引用是否会 “表现” 一致。

可变类型示例（以自定义的 `Counter` 类为例）

typescript

```typescript
class Counter {
    private value: number;

    constructor() {
        this.value = 0;
    }

    // 变更操作：增加计数器的值
    public increment(): void {
        this.value++;
    }

    // 观察操作：获取计数器当前的值
    public getValue(): number {
        return this.value;
    }
}

const counter1 = new Counter();
const counter2 = new Counter();
```

- **观察相等性**:
  在初始状态下，`counter1.getValue()` 和 `counter2.getValue()` 都返回 `0`。此时，仅通过调用不会改变对象状态的观察操作（这里是 `getValue` 方法），无法区分 `counter1` 和 `counter2`，所以在当前状态下它们是观察相等的。
  但是，如果对 `counter1` 调用 `increment` 变更操作，`counter1.increment();` 此时 `counter1.getValue()` 返回 `1`，而 `counter2.getValue()` 仍返回 `0`。通过调用观察操作发现两者出现了差异，所以在 `counter1` 调用变更操作后，它们不再是观察相等的。
- **行为相等性**:
  假设在初始状态下 `counter1` 和 `counter2` 是观察相等的（都为 `0`）。行为相等性要求在

未来无论对其中一个对象进行何种变更操作，两个对象在所有状态下都无法区分。例如，如果对 `counter1` 调用 `increment` 操作，`counter1.increment();` 此时 `counter1` 和 `counter2` 在当前状态下观察不相等。如果要满足行为相等性，那么对 `counter2` 也进行相同的 `increment` 操作后，它们在后续所有状态下的观察结果都应该相同。但在这个例子中，如果只对 `counter1` 进行操作，它们就不满足行为相等性。只有当对两个对象进行相同的一系列操作（包括变更操作和观察操作）时，它们的结果始终相同，才满足行为相等性。

深度相对

```typescript
const arr1: Array<number> = [1, 2, 3];
const arr2: Array<number> = [1, 2, 3];

const set1: Set<string> = new Set(["a", "b"]);
const set2: Set<string> = new Set(["b", "a"]);
```

由于 `Array` 和 `Set` 是可变的，`===` 操作符按预期实现的是行为相等性。在这种情况下，`arr1!== arr2` 且 `set1!== set2`，因为它们在内存中各自指向不同的可变对象。

但是，没有类似于我们在上一节中引入的 `equalValue()` 操作那样的用于观察相等性的标准操作符。没有内置的方法来判断 `arr1` 和 `arr2` 当前是否表示相同的元素序列，同样也无法判断 `set1` 和 `set2` 是否表示相同的元素集合。

一些库试图通过提供 “深度相等” 操作来弥补这一不足，这种操作不仅适用于 `Array` 和 `Set` 这样的集合，还适用于各种各样的对象类型：

- Node 库中的 `Assert.deepStrictEqual()`
- Underscore.js 及其继任者 Lodash 中的 `isEqual()`

这些操作之所以被称为 “深度”，是因为它们可以深入多层集合进行比较。例如，对于 `Array<Map<T, Set<U>>>` 类型的两个值，也可以进行观察相等性的比较。

但是，使用这些深度相等操作时必须格外小心。`Assert.deepStrictEqual()` 的规范对 `Map` 和 `Set` 这样的内置集合有特殊处理，例如，会忽略元素的顺序，以便能够正确比较这些集合的抽象值。但是对于其他对象类型，这些深度相等操作会盲目地逐个字段比较表示形式，而完全不考虑用于解释表示形式的抽象函数。