11.抽象函数与表示不变式

不变式

不变式是程序的一种属性，对于程序的每一个可能的运行时状态，这种属性总是成立的。
比如：
1.immutability（就是不变量）

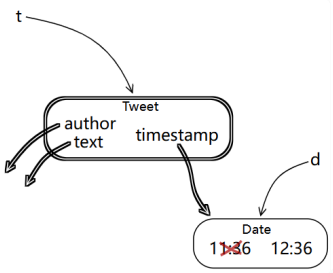
```
class Tweet {

    private readonly author: string;
    private readonly text: string;
    private readonly timestamp: Date;

    public constructor(author: string, text: string, timestamp: Date) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }

    /**
     * @returns Twitter user who wrote the tweet
     */
    public getAuthor(): string {
        return this.author;
    }

    /**
     * @returns text of the tweet
     */
    public getText(): string {
        return this.text;
    }

    /**
     * @returns date/time when the tweet was sent
     */
    public getTimestamp(): Date {
        return this.timestamp;
    }

}
```

其中的author text timestamp 都是创建后就不会改变的
当然这还不是真正的不变量，因为有办法改变其中的数值，以timestamp为例，

But that's not the end of the story: the rep is still exposed! Consider this perfectly reasonable client code that uses `Tweet`:

```
/**
 * @returns a tweet that retweets t, one hour later
 */
function retweetLater(t: Tweet): Tweet {
    const d: Date = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("rbmllr", t.getText(), d);
}
```

`retweetLater` takes a tweet and should return another tweet with the same message (called a *retweet*) but sent an hour later. The `retweetLater` method might be part of a system that automatically echoes funny things that Twitter celebrities say.



这个会改变timestamp

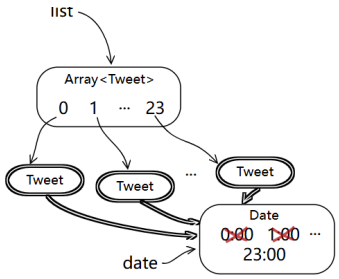
改为

```
public getTimestamp(): Date {
    return new Date(this.timestamp.getTime());
}
```

这样新建data对象，可以防止改变timestamp

以及这种问题：

```
/**
 * @returns an array of 24 inspiring tweets, one per hour today
 */
function tweetEveryHourToday(): Array<Tweet> {
    const array: Array<Tweet> = [];
    const date: Date = new Date();
    for (let i = 0; i < 24; i++) {
        date.setHours(i);
        array.push(new Tweet("rbmllr", "keep it up! you can do it", date));
    }
    return array;
}
```



构造函数可以改为这样：

```
public constructor(author: string, text: string, timestamp: Date) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

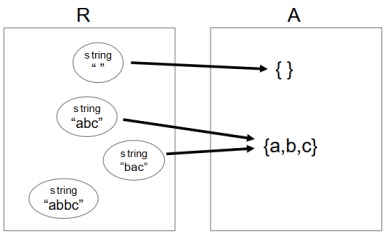
2.变量的类型（例如 `i: number` 意味着 `i` 始终是一个数字）

3.变量之间的关系（例如，当 `i` 用于遍历数组的索引时， `0 <= i < array.length` 就是循环体内部的一个不变式）
等等

更高层次看 表示不变式和抽象函数

Then the rep space R contains strings, and the abstract space A is mathematical sets of characters. We can show the two value spaces graphically, with an arrow from a rep value to the abstract value it represents. There are several things to note about this picture:

- **Every abstract value is mapped to by some rep value**. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- **Some abstract values are mapped to by more than one rep value**. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.
- **Not all rep values are mapped**. In this case, the string "abbc" is not mapped, because we have decided that the rep string should not contain duplicates. This will allow us to terminate the `remove` method when we hit the first instance of a particular character, since we know there can be at most one.



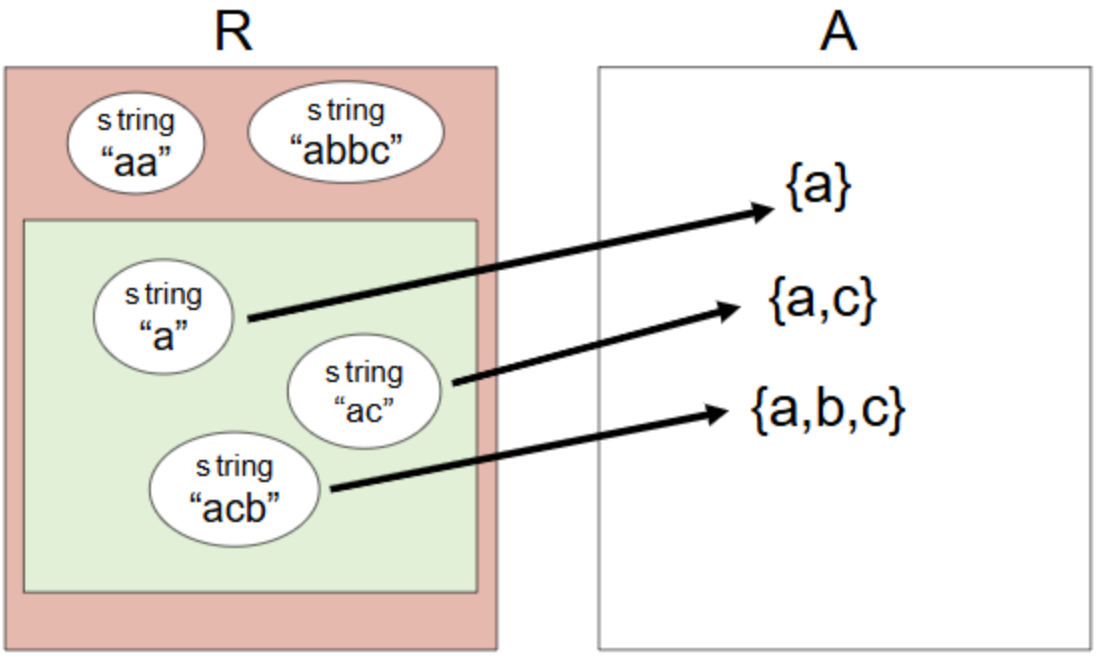
上面的内容讲白了就是说R（表示值rep values）中的内容可以不对应A（抽象函数）中的内容，亦可以“多对一”，但是A（抽象函数）对于R（表示值rep values）可以“一对多”，但至少是“一对一”

1. An *abstraction function* that maps rep values to the abstract values they represent:

$$AF : R \rightarrow A$$

2. A *rep invariant* that maps rep values to booleans:

$$RI : R \rightarrow boolean$$



表示不变式可以用于检查

```
class BankAccount {
    private balance: number;

    constructor(initialBalance: number) {
        this.balance = initialBalance;
        this.checkRep();
    }

    // 存款方法
    public deposit(amount: number): void {
        this.balance += amount;
        this.checkRep();
    }

    // 取款方法
    public withdraw(amount: number): void {
        if (amount <= this.balance) {
            this.balance -= amount;
            this.checkRep();
        } else {
            console.log("余额不足，无法取款");
        }
    }

    // 检查表示不变式的方法
    private checkRep(): void {
        // 余额不能为负数
        if (this.balance < 0) {
            throw new Error("表示不变式被违反：余额不能为负数");
        }
    }
}
```

其中表示不变式是balance的值不为负数，所以每次调用完函数后都检查一下是否符合表示不变式（可以用“断言”）

表示值默认不是null和undefined等空值

记录抽象函数（AF）、表示不变式（RI）以及防止表示泄露的安全性说明

```
// Immutable type representing a tweet.
class Tweet {

    private readonly author: string;
    private readonly text: string;
    private readonly timestamp: Date;

    // Rep invariant:
    //   author is a Twitter username (a nonempty string of letters, digits, underscores)
    //   text.length <= 280
    // Abstraction function:
    //   AF(author, text, timestamp) = a tweet posted by author, with content text,
    //                                 at time timestamp
    // Safety from rep exposure:
    //   All fields are private;
    //   author and text are Strings, so are guaranteed immutable;
    //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
    //         make defensive copies to avoid sharing the rep's Date object with clients.

    // Operations (specs and method bodies omitted to save space)
    public constructor(author: string, text: string, timestamp: Date) { ... }
    public getAuthor(): string { ... }
    public getText(): string { ... }
    public getTimestamp(): Date { ... }
}
```

抽象数据类型（ADT）的不变式取代前置条件

现在让我们来综合理解一下。设计良好的抽象数据类型有一个极大的优势，即它封装并强制保证了一些属性，而这些属性如果没有它，我们就不得不通过前置条件来规定。例如，与其使用这样带有复杂前置条件的规格说明：

typescript

```typescript
/**
 * @param set1 是一个已排序且无重复字符的集合
 * @param set2 同理
 * @returns 出现在其中一个集合但不出现在另一个集合中的字符，
 *          以已排序且无重复的形式返回
 */
static exclusiveOr(set1: string, set2: string): string;
```

我们不妨使用一个能体现所需属性的抽象数据类型：

收起

typescript

```typescript
/**
 * @returns 出现在其中一个集合但不出现在另一个集合中的字符
 */
static exclusiveOr(set1: SortedCharSet, set2: SortedCharSet): SortedCharSet;
```

但是我还是觉得之前的那种在小规模编程时有优势