泛型 Generics

泛型指使用时才定义类型，即类型可以像参数一样定义，主要解决类、接口、函数的复用性，让它们可以处理多种类型。

#基本使用

下面示例返回值类型是 any，这不是我们想要的，因为我们想要具体返回类型

```
function dump(arg: any) {
    return arg;
}

let hd = dump('后盾人') //类型为 any
let xj = dump(true) //类型为 any

```

使用了泛型定义后，返回值即为明确的类型

```
function dump<T>(arg: T): T {
    return arg;
}
let hd = dump<string>('后盾人')

```

如果调用时不指定类型系统也会自动推断类型

```
...
let hd = dump('后盾人') //hd 类型为 string
...

```

#类型继承

下面的代码是不严谨的，我们不需要处理数字，因为数字没有 length 属性，同时我们希望返回类型不是 any

```
function getLength(arg: any) {
```

```
    return arg.length;
}
console.log(getLength('houdunren.com')); //13
console.log(getLength(['后盾人'])); //1
console.log(getLength(18)); //undefined

```

泛型是不确定的类型，所以下面读取 length 属性将报错

```
function getLength<T>(arg: T): number {
    return arg.length; //类型“T”上不存在属性“length”
}

```

我们可以通过继承来解决这个问题

```
function getLength<T extends string>(arg: T): number {
    return arg.length;
}

```

上例只能处理字符串，不能处理数组等包含 length 的数据，我们可以通过继承 extends 继承，让泛型定义包含 length 属性

```
function getLength<T extends { length: number }>(arg: T): number {
    return arg.length;
}

//或使用 interface 或 type

type LengthType = { length: number }
function getLengthAttribute<T extends }>(arg: T): number {
    return arg.length;
}

```

如果你的类型只是字符串或数组，也可以使用联合类型

```
function getLength<T extends string | any[]>(arg: T): number {
```

```
    return arg.length
}

console.log(getLength('houdunren.com'))
console.log(getLength(['后盾人', '向军']))

```

TS 也会自动推断，比如下面参数是 T[]，TS 会推断为数组类型，所以这时候是存在 length 的，不会报错

```
function getLength<T>(arg: T[]): number {
    return arg.length;
}

```

将泛型理解为动态类型，他最终也会是一个类型，所以使用方式与我们其他类型一样的。比如下面的返回值类型，我们就返回了一个元组，包括泛型与数值类型

```
function getLength<T extends string>(arg: T): [T, number] {
    return [arg, arg.length];
}

let hd = getLength<string>('houdunren.com')

```

#类

下面我们来掌握在类中使用泛型的方法

使用泛型复用类

下面是对数值与字符串类型的集合进行管理，因为业务是一样的，所以下面的实现是重复的

```
class CollectionNumber {
    data: number[] = []
    public push(...items: number[]) {
        this.data.push(...items)
    }
    public shift() {
        return this.data.shift()
```

```
    }
}

class CollectionString {
    data: string[] = []
    public push(...items: string[]) {
        this.data.push(...items)
    }
    public shift() {
        return this.data.shift()
    }
}

const numberCollection = new CollectionNumber()
numberCollection.push(1)
const stringCollection = new CollectionString()
stringCollection.push('后盾人', '向军')

console.log(stringCollection.shift());

```

上例使用泛型来控制就好多了

```
class Collection<T> {
    data: T[] = []
    public push(...items: T[]) {
        this.data.push(...items)
    }
    public shift() {
        return this.data.shift()
    }
}

const collections = new Collection<number>()
collections.push(1)

type User = { name: string, age: number }
const hd: User = { name: "后盾人", age: 18 }
const userCollection = new Collection<User>()
```

```
17
18 userCollection.push(hd)
19 console.log(userCollection.shift());
20
```

接口结合泛型

下面的代码是不稳定的，我们的意图是传递用户数据，但没有类型约束情况下，可以传递任何类型

```
1  class User {
2      (protected _user) { }
3      public get() {
4          return this._user
5      }
6  }
7
8  const instance = new User({ name: '后盾人' })
9  console.log(instance.get());
10
```

对类使用泛型处理后，可以保证传递与返回值的类型，并具有良好的代码提示

```
1  class User<T>{
2      (protected _user: T) { }
3      public get(): T {
4          return this._user
5      }
6  }
7
8  interface UserInterface {
9      name: string, age: number
10 }
11 const instance = new User<UserInterface>({ name: '后盾人', age: 18 })
12 console.log(instance.get().age);
13
```

#接口

下面对接口的类型使用泛型定义，比如 isLock 可以为 `number` 或`boolean`，并对文章的评论内容进行定义。

这样处理代码会有严格类型约束，并有良好的代码提示

```
//文章接口
interface articleInterface<T, B> {
    title: string,
    isLock: B,
    comments: T[],
}

//评论类型
type CommentType = {
    comment: string
}

//定义文章数据包含评论内容
const hd: articleInterface<CommentType, boolean> = {
    title: '后盾人官网',
    isLock: true,
    comments: [
        { comment: '这是一个评论' }
    ]
}

console.log(hd);

```

#值类型

下面解构得到的变量类型不是具体类型，面是数组类型，比如变量 y 的类型是 **string | (() => void)**
这在写项目时是不安全的，因为可以将 y 随时修改为字符串，同时也不会有友好的代码提示

```
function hd() {
    let a = '后盾人'
    let b = (x: number, y: number): number => x + y
    return [a, b]
}

const [x, y] = hd() //变量 y 的类型为 string | (() => void)
```

```
8
```

使用 as const 就可以很高效的解决上面的问题，可以得到具体的类型，来得到更安全的代码，同时会有更好的代码提示

```
function hd() {
  let a = '后盾人'
  let b = (): void => {}
  return [a, b] as const
}

const [x, y] = hd() //变量 y 的类型为 () => void

```

也可以使用泛型来得到具体的值类型

```
function hd() {
    const a: string = '后盾人'
    const b: number = 2090
    return f(a, b)
}
function f<T extends any[]>(...args: T): T {
    return args;
}
const [r, e] = hd()
```