类型断言

使用在接口值上的操作

`x.(T)`，断言操作，判断接口值x是否持有T类型的值

基本语法

```
value, ok := x.(T)
```

if `x` 持有 `T` 类型的值：

`value` 获得具体类型的值

`ok = true`

else 不匹配：

`value` 为T类型的零值（缺省值）

`ok = false` （而不会panic）

e.g.

```
func typeAssertEx() {
        var w io.Writer
        w = os.Stdout // *os.File类型
        f, ok := w.(*os.File)
        fmt.Printf("%#v, %#v\n", f, ok) // true
        c, ok := w.(*bytes.Buffer)
        fmt.Printf("%#v, %#v\n", c, ok) // false
}
```

in ch7-2 接口值的结构：

```
type iface struct {
    tab  *itab // 类型信息（这里是*os.File）
    data unsafe.Pointer // 值指针（指向os.Stdout）
}
```

类型： `*os.File` ， 分别进行了两次类型断言

nil必败

如果断言操作的对象是一个nil接口值，那么不论被断言的类型是什么这个类型断言都会失败

```
var w io.Writer  // 声明但未赋值的接口变量，此时是nil接口值

// 尝试对nil接口值进行类型断言
f, ok := w.(*os.File)
fmt.Printf("断言*os.File: (%#v, %v)\n", f, ok)  // (nil, false)

// 甚至断言为interface{}也会失败
a, ok := w.(interface{})
fmt.Printf("断言interface{}: (%#v, %v)\n", a, ok)  // (nil, false)

w = os.Stdout
// 对非nil接口值进行类型断言
f, ok = w.(*os.File)
fmt.Printf("断言*os.File: (%#v, %v)\n", f, ok)  // (xxx, true)
```

区分不同错误类型

```
if err != nil {
    // 断言，如果确实是这类错误，ok = true，进入if进行处理
    if pathErr, ok := err.(*os.PathError); ok {
        // 处理文件路径错误
        fmt.Printf("操作 %q 路径 %q 失败: %v\n",
            pathErr.Op, pathErr.Path, pathErr.Err)
    } else if linkErr, ok := err.(*os.LinkError); ok {
        // 处理链接错误
        fmt.Printf("链接错误: %v\n", linkErr)
    } else {
        // 其他类型错误
        fmt.Printf("未知错误: %v\n", err)
    }
}
```

通过类型断言询问行为

```go
type Stringer interface {
    String() string
}

// 检查v是否实现Stringer接口
func printIfStringer(v interface{}) {
    if s, ok := v.(Stringer); ok {
        fmt.Println(s.String())
    } else {
        fmt.Printf("%v 没有实现 Stringer 接口\n", v)
    }
}
```

类型分支

`x.(type)`，到底是什么type?

`switch & case`：everyday，if-else go away! 🥳

```go
func processInput(input interface{}) {
    switch v := input.(type) {
    case int:
        fmt.Printf("整数: %d\n", v*2)
    case string:
        fmt.Printf("字符串长度: %d\n", len(v))
    case bool:
        fmt.Printf("布尔值: %t\n", v)
    default:
        fmt.Printf("不支持的类型: %T\n", v)
    }
}
```

当然也可以用于错误处理

```go
func handleError(err error) {
    switch e := err.(type) {
    case *os.PathError:
        fmt.Printf("文件错误: 操作=%s 路径=%s\n", e.Op, e.Path)
    case *json.SyntaxError:
        fmt.Printf("JSON语法错误(偏移量%d): %v\n", e.Offset, e)
    case nil:
        // 没有错误
        fmt.Printf("没有错误\n")
    default:
        fmt.Printf("未知错误: %v\n", err)
    }
}
```

7.15《一些建议》

YAGNI原则

You Aren't Gonna Need It
提前接口导致接口焦虑 & 接口浪费
《当真正需要多态行为时，接口会自然出现》

```go
// 不好的做法：为单一实现定义接口
type UserRepository interface {
    GetUser(id int) (*User, error)
}

type DBUserRepository struct{}

func (r *DBUserRepository) GetUser(id int) (*User, error) {
    // 数据库实现
}
```

就一个类型需要这个接口，别的又用不到，那你不如搞一个普通函数咯

接口方法调用比直接方法调用稍慢，过度抽象害死人嘞😋

小而精 / 正交性设计

像 `io.Writer` ，只包含了一个方法的接口，但是造福了诸如 `os.Stdout` 等几百种类型

劳模接口，简单、专业、又好用😘

避免"上帝接口" All in one？🤔谁家的超大杯接口🤭

《接口越大，抽象越弱》——Rob Pike（您哪位？）

实用主义优先

有点像第一点，优先使用具体类型和函数，当需要抽象时再引入接口

什么使用定义接口呢？**只有当有两个或更多具体类型需要统一处理时才定义接口**

即，适合抽象的时候才抽象😊 🤏 😊

```
// 好的做法：当有多种实现时自然产生接口
type Cache interface {
    Get(key string) ([]byte, error)
    Set(key string, value []byte) error
}

// 内存缓存实现
type InMemoryCache struct{}

// Redis缓存实现
type RedisCache struct{}

// MySql缓存实现
type MySqlCache struct{}

// 这时Cache接口才有存在价值
```

例外情况——跨包解耦

（唯一）允许"一对一"接口的情况：跨包解耦（你说唯一就唯一？你什么接口？😊）

定义接口的包：只关心"做什么"（行为契约）
实现接口的包：负责"怎么做"（具体实现）

使用接口的包：依赖抽象而非具体实现

```
// 在包A定义接口
package repository

type UserRepository interface {
    GetUser(id int) (*User, error)
}

// 在包B实现具体类型
package mysql

type UserRepo struct{}

func (r *UserRepo) GetUser(id int) (*User, error) {
    // MySQL实现
}

// mysql可以无痛切换到sqlite
package sqlite
...
```