



如何进行面向对象编程



张华

64174234@qq.com

内容

1.1 如何为对象添加方法

1.2 如何实现接口

1.3 一个仿真音乐播放器的实现

1.4 总结

1.5 思考

1.1如何为对象添加方法

□ 通过在func和函数名之间增加接收者的方式实现

```
package main
import "fmt"
type Student struct { //定义学生成绩类型
    name string //姓名
    stuno int    //学号
    score int    //成绩
}
func (s *Student) Stuin() {
    fmt.Println("input name of student ")
    fmt.Scanln(&s.name) //录入姓名
    fmt.Println("input stuno of student ")
    fmt.Scanln(&s.stuno) //录入学号
    fmt.Println("input score of student ")
    fmt.Scanln(&s.score) //录入成绩
}
func (s *Student) Stuout() {
    fmt.Println(*s)
}
func main() {
    stu := &Student{}
    stu.Stuin()
    stu.Stuout()
}
```

func 和 Stuin之间的(s *Student)为接收者，强调Stuin这个方法是属于*Student类型的对象。S同时也是Stuin的形参。
s是指针类型，方法Stuin可以修改s指向的值，即形参的改变能够影响到实参

对象的方法的调用方式：对象名.方法(参数列表)

1.1如何为对象添加方法(续)

□ 可以为任意类型添加相应的方法，指针类型除外

```
1 package main
2
3 import "fmt"
4
5 type Integer int
6
7 func (a Integer) Less(b Integer) bool { //面向对象的方法
8     return a < b
9 }
10 func main() {
11     var a Integer = 1
12     if a.Less(2) {
13         fmt.Println(a, "Less 2")
14     } else {
15         fmt.Println(a, "Not Less 2")
16     }
17 }
```

定义了一个新类型Integer,和int没有本质不同,只是又为内置的int类型增加了新方法Less.这样Integer就可以像一个普通的类一样使用

Less方法面向过程的定义方式为
func Less(a Integer,b Integer) bool {
 return a<b
}

Less方法面向过程的调用方式为
Less(a,2)

所以，面向对象只是换了一种语法方式表达而已

1.2如何实现接口

□ 什么是接口？

■ 接口类型是一组方法定义的集合。

□ 具体来说就是把我们经常用到的方法集中起来定义，形成**规范 and 标准**（输入类型、输出类型、方法名）

□ 如同电脑主板上的USB接口、COM口、PCI接口、...

■ 接口的实现本着**谁使用谁实现**的原则，按需定制化开发

□ 使用者都遵循接口定义的命名、输入及输出方式来实现接口

□ 如同U盘、USB键鼠、USB吸尘器、USB风扇、USB无线网卡一样，它们都按照USB接口标准实现了各自不同的功能



1.2如何实现接口(续)

□ 接口的定义及实现

```
type Abser interface { //定义一个接口类型Abser
    Abs() float64 //接口中的方法名为Abs, 无输入, 输出类型为float64
}
type MyFloat float64 //定义了一个Myfloat类型
//Myfloat实现了接口中的Abs方法, 功能为求绝对值
func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct { //定义了顶点坐标类型Vertex
    X, Y float64
}

/*Vertex实现了接口的Abs方法, 功能为顶点V到原点的距离
func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}
```

MyFloat类型和*Vertex类型都实现了接口Abser中的Abs方法, 但内部代码的功能各不相同。

一个类只要实现了接口要求的所有函数, 我们就说这个类实现了接口。

1.2如何实现接口(续)

□ 怎么区分一个对象类型中的方法是接口中的方法还是普通方法呢？

■ 根据软件业务人工区分，编译器自动识别

```
func main() {  
    //定义接口变量Abser  
    var a Abser  
    //定义变量f 初始值为-2的平方根，转换成MyFloat类型  
    var f = MyFloat(-math.Sqrt2)  
    //定义顶点变量v初始值为(3,4)  
    v := Vertex{3, 4}  
    // 接口赋值: Myfloat类型完全实现了接口A的全部方法,就可以把f赋给a  
    a = f  
    //调用并输出  
    fmt.Println(a.Abs())  
    // 接口赋值: *Vertex 实现了 Abser中的所有方法,就可以把&v赋给a  
    a = &v  
    //调用并输出  
    fmt.Println(a.Abs())  
}
```

接口赋值：确定将哪个对象接入接口
接口调用：执行接入对象的接口方法

任何实现了接口中全部方法的对象
都可赋给接口变量。

接口变量会根据他当前的对象值，
去执行相应的接口实现方法

1.2如何实现接口(续)

- 怎么区分一个对象类型中的方法是接口中的方法还是普通方法呢？
 - 有时候不需要区分，实现后，直接按普通方法使用

```
func main() {  
    //定义变量f 初始值为-2的平方根，转换成MyFloat类型  
    var f = MyFloat(-math.Sqrt2)  
    //定义顶点变量v初始值为(3,4)  
    v := Vertex{3, 4}  
    fmt.Println(f.Abs()) //输出  
    fmt.Println(v.Abs()) //输出  
}
```


1.2如何实现接口(续)

□ 怎么区分一个对象类型中的方法是接口中的方法还是普通方法呢？

■ 有时无需要知道接口在哪定义, 即**隐式实现接口**

```
package main
import "fmt"
type Person struct {
    Name string
    Age  uint
    Sex  byte
} //定义了一个名为Person的结构体类型
func (p Person) String() string { //具体实现fmt包中的Stringer接口
    /*Printf 将参数列表a填写到格式控制串format的占位符中*/
    /*func Printf(format string, a ...interface{}) string*/
    return fmt.Sprintf("\n%v(%c,%v years)\n", p.Name, p.Sex, p.Age)
}
```

□Stringer是一个可用字符描述自己的接口类型
是一个普遍存在的接口, 定义在fmt包中:

```
type Stringer struct {
    String() string
}
```

```
func main() {
    F1 := Person{"言承旭", 37, 'm'}
    F2 := Person{"周渝民", 33, 'm'}
    F3 := Person{"吴建豪", 36, 'm'}
    F4 := Person{"朱孝天", 35, 'm'}
    fmt.Println(F1, F2, F3, F4)
}
```

[`go run oop7.go` | done: 885.0506ms]

言承旭(m,37 years)

周渝民(m,33 years)

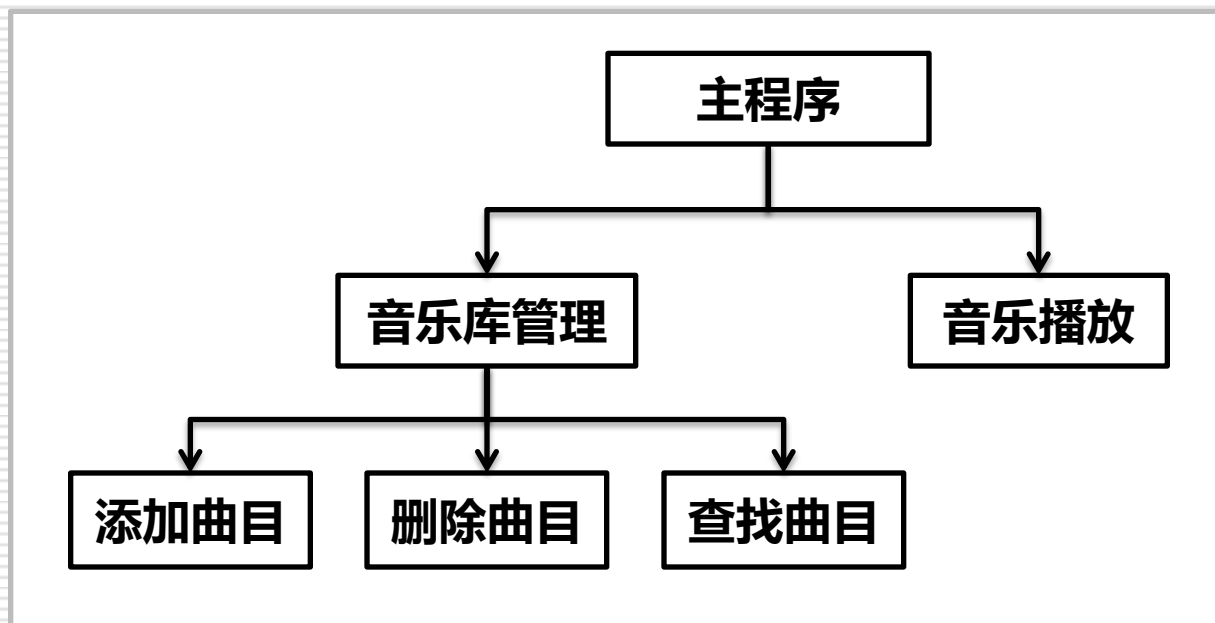
吴建豪(m,36 years)

朱孝天(m,35 years)

1.3 一个仿真音乐播放器的实现

□ 功能分析

- 音乐库管理，使用者可以查看、添加和删除里面的音乐曲目
- 播放音乐
- 支持MP3和WAV，也能随时扩展支持更多音乐格式
- 退出程序



1.3一个仿真音乐播放器的实现(续)

□ UI分析

- 这个程序是一个命令行程序。该程序在运行后进入一个循环，用于监听命令输入的状态。可以接收并执行以下命令。
 - 音乐库管理命令：lib，包括list/add/remove命令
 - 音乐播放：play命令，以歌曲名作为参数
 - 退出程序：q命令

1.3一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐类型定义

```
type Music struct { //音乐类型定义
    Id      string //音乐ID
    Name    string //音乐名称
    Artist  string //艺术家
    Source  string //播放路径
    Type    string //音乐类型
}
```

■ 音乐库类型定义

```
type MusicManager struct { //音乐库类型
    musics []Music //成员为一个音乐类型的切片
}
```

1.3 一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐库的构造函数定义

```
//构造函数，初始化为0首音乐。构造函数的样式为NewXXX()  
func NewMusicManager() *MusicManager {  
    return &MusicManager{make([]Music, 0)} //初始音乐切片长度为0  
}
```

■ 音乐库的音乐数量方法定义

```
//乐库类的Len方法，获取乐库的音乐数量  
func (m *MusicManager) Len() int {  
    return len(m.musics) //返回切片的长度即歌曲的数量  
}
```

1.3一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐库的获取音乐索引方法定义

```
//乐库类的Get方法,获取索引值为index的音乐信息
func (m *MusicManager) Get(index int) (music *Music, err error) {
    if index < 0 || index >= len(m.musics) { //索引不在合理范围
        return nil, errors.New("Index out of range.") //返回nil和错误信息
    }
    return &m.musics[index], nil //返回曲库中索引为index的音乐信息和nil
}
```

■ 音乐库音的按音乐名查找方法定义

```
//乐库类的按音乐名称查找方法
func (m *MusicManager) Find(name string) (*Music, int) {
    if len(m.musics) == 0 { //乐库为空
        return nil, -1 //返回nil,-1
    }
    for i, m := range m.musics { //乐库不为空, 遍历音乐库
        if m.Name == name { //找到
            return &m, i
        }
    }
    return nil, -1 //乐库不为空但没找到返回nil,-1
}
```

1.3一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐库的增加音乐方法定义

// 乐库类的增加音乐方法

```
func (m *MusicManager) Add(music *Music) {  
    m.musics = append(m.musics, *music) // 追加  
}
```

1.3一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐库的删除音乐方法定义

```
//乐库类的删除方法，
func (m *MusicManager) Remove(index int) *Music {
    if index < 0 || index >= len(m.musics) {
        return nil //删除的索引值范围不对，返回空
    }
    removeMusic := m.musics[index] //找到要删除的音乐信息赋给removeMusic
    //从切片中删除元素
    if index < len(m.musics)-1 { //中间元素
        m.musics = append(m.musics[:index-1], m.musics[index+1:]...)
    } else if index == 0 { //删除仅有的一个元素
        m.musics = make([]Music, 0)
    } else { //删除最后一个元素
        m.musics = m.musics[:index-1]
    }
    return &removeMusic //返回被删除的音乐信息
}
```


1.3 一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐播放接口定义

```
type Player interface { //定义播放接口
    Play(source string)
}
```

1.3 一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ 音乐播放函数定义

```
func Play(source, mtype string) { //定义播放函数
    var p Player //定义一个接口变量p
    switch mtype {
    case "MP3":
        p = &MP3Player{} //接口赋值，将MP3对象接入接口对象p
    case "WAV":
        //p = &WAVPlayer //方便以后扩展，暂时没有实现
    default:
        fmt.Println("Unsupported music type", mtype)
        return
    }
    p.Play(source) //调用p对应的类型所实现的paly接口
}
```

1.3一个仿真音乐播放器的实现(续)

□ 基本数据结构定义

■ MP3音乐类型定义

```
type MP3Player struct { //mp3类型定义
    stat      int
    progress int
}
```

■ MP3音乐类型的播放接口实现

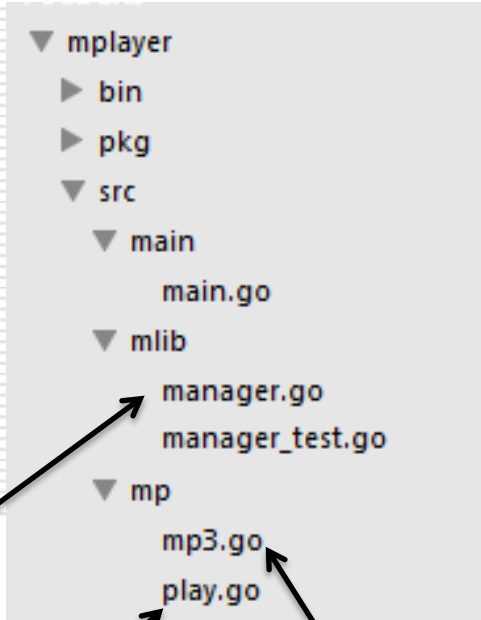
```
func (p *MP3Player) Play(source string) { //play接口实现
    fmt.Println("Playing MP3 music", source)
    p.progress = 0
    for p.progress < 100 {
        time.Sleep(100 * time.Second) //假装正在播放
        fmt.Println(".")
        p.progress += 10
    }
    fmt.Println("\nFinished playing", source)
}
```

1.3 一个仿真音乐播放器的实现(续)

□ 主程序

■ 包与全局变量及项目目录结构

```
package main
import (
    "bufio" //输入输出缓存包
    "fmt"
    "mlib"   //这个包里面定义音乐库数据结构
    "mp"     //这个包里面定义接口、播放函数、MP3类型及接口实现
    "os"     //系统包
    "strconv" //字符串转换包
    "strings" //字符串处理包
)
var lib *mlib.MusicManager //音乐库指针对象lib
var id int = 1             //音乐编号id
```



```
graph TD
    mplayer --> bin
    mplayer --> pkg
    mplayer --> src
    src --> main
    src --> mlib
    src --> mp
    main --> main_go[main.go]
    mlib --> manager_go[manager.go]
    mlib --> manager_test_go[manager_test.go]
    mp --> mp3_go[mp3.go]
    mp --> play_go[play.go]
```

The diagram illustrates the project directory structure. It shows a root directory 'mplayer' containing subdirectories 'bin', 'pkg', and 'src'. The 'src' directory contains 'main', 'mlib', and 'mp'. 'main' contains 'main.go'. 'mlib' contains 'manager.go' and 'manager_test.go'. 'mp' contains 'mp3.go' and 'play.go'. Arrows indicate the relationships between these directories and files, showing how the 'main' package imports the 'mlib' and 'mp' packages.

1.3一个仿真音乐播放器的实现(续)

□ 主程序

■ 主函数

```
func main() {
    fmt.Println(`
        Enter following commands to control the player:
        lib list -- View the existing music lib
        lib add <name><artist><source><type> --Add a music to the music lib
        lib remove <name> --Remove the specified music from the lib
        play <name> --Play the specified music
        `) //Println 用``可以对其字符串按照所见即所得模式管理
    lib = mlib.NewMusicManager() //新建立一个音乐库对象
    r := bufio.NewReader(os.Stdin) //建立一个缓存对象r, 可以接收来自控制台的输入
    for {
        fmt.Print("Enter command->")
        rawLine, _, _ := r.ReadLine() //读取一行数据到rawLine数组
        line := string(rawLine)       //将rawLine转换成字符串赋给line
        if line == "q" || line == "e" { //如果等于q或e退出程序
            break
        }
        tokens := strings.Split(line, " ") //用空格分隔命令串
        if tokens[0] == "lib" {             //如果第一部分为lib
            handleLibCommand(tokens) //执行处理音乐库管理功能
        } else if tokens[0] == "play" { //如果第一部分为play
            handlPlayCommand(tokens) //执行音乐播放功能
        } else { //否则 提示命令无法识别
            fmt.Println("Unrecognized command:", tokens[0])
        }
    }
}
```

1.3一个仿真音乐播放器的实现(续)

□ 主程序

■ 音乐库管理函数

```
func handleLibCommand(tokens []string) { //参数为字符串切片
    switch tokens[1] {
    case "list": //列出音乐清单
        for i := 0; i < lib.Len(); i++ {
            e, _ := lib.Get(i) //获取索引为i的音乐
            fmt.Println(i+1, ":", e.Name, e.Artist, e.Source, e.Type)
        }
    case "add": //增加音乐
        if len(tokens) == 6 { //如果命令的长度为6, 正确的增加命令split后的长度为6
            id++
            //音乐入库
            lib.Add(&lib.Music{strconv.Itoa(id), tokens[2], tokens[3], tokens[4], tokens[5]})
        } else { //提示增加音乐命令的正确格式是什么
            fmt.Println("USAGE:lib add <name><artist><source><type>")
        }
    case "remove": //删除音乐
        if len(tokens) == 3 { //如果命令的长度为3, 正确的删除命令split后的长度为3
            _, i := lib.Find(tokens[2]) //找要删除的音乐
            if i != -1 { //找到
                lib.Remove(i)
            } else { //乐库为空或没找到
                fmt.Println("music", tokens[2], "is not exist in lib")
            }
        } else { //提示删除音乐命令的正确格式是什么
            fmt.Println("USAGE:lib remove<name>")
        }
    default: //提示命令无法识别
        fmt.Println("Unrecognized lib command:", tokens[1])
    }
}
```

1.3一个仿真音乐播放器的实现(续)

□ 主程序

■ 播放处理函数

```
func handlePlayCommand(tokens []string) { //音乐播放
    if len(tokens) != 2 { //如果命令长度不等于2，正确的播放命令split的长度为2
        fmt.Println("USAGE:play<name>") //提示正确的格式
        return
    }
    e, _ := lib.Find(tokens[1]) //查找音乐
    if e == nil { //没找到
        fmt.Println("The music", tokens[1], "does not exist.")
        return
    }
    mp.Play(e.Source, e.Type) //执行播放函数
}
```

1.4总结

- 方法的定义(掌握)
- 方法的调用(掌握)
- 接口的定义(掌握)
- 接口的实现(掌握)
- 接口的赋值(掌握)
- 接口的调用(掌握)
- 下面再补充一些
 - 接口间的赋值
 - 接口查询
 - 类型查询
 - 接口组合
 - Any类型

1.4总结(续)

□ 接口间赋值

- 任何实现了one.ReadWriter接口的类，均实现了two.Istream
- 任何one.ReadWriter接口对象可赋值给two.Istream,反之亦然
- 任何地方使用one.ReadWriter接口和two.Istream接口并无差别

```
package one
type ReadWriter interface{
    Read(buf []byte) (n int,err error)
    Write(buf []byte)(n int,err error)
}
```

```
package two
type IStream interface{
    Write(buf []byte)(n int,err error)
    Read(buf []byte) (n int,err error)
}
```

```
var file1 two.IStream = new(File)
var file2 one.ReadWriter = file1
var file3 two.IStream = file2
```

1.4总结(续)

□ 接口间赋值

- 接口赋值并不要求两个接口必须等价。如果接口A的方法列表是接口B的方法类别的子集，那么接口B可以赋值给接口A

```
package one
type ReadWriter interface{
    Read(buf []byte) (n int,err error)
    Write(buf []byte)(n int,err error)
}
```

```
package two
type IStream interface{
    Write(buf []byte)(n int,err error)
    Read(buf []byte) (n int,err error)
}
```

```
var file1 two.IStream = new(File)
var file4 Writer = file1
```



```
var file1 Writer = new(File)
var file4 two.IStream = file1
```



file1中没有Read方法

1.4总结(续)

□ 接口查询

- 如何查询一个file1接口指向的对象实例是否实现了two.IStream接口？

```
var file1 Writer = new(File)
var file4 two.IStream = file1
```



```
var file1 Writer = new(File)
if file5, ok := file1.(two.IStream); ok {
    ...
}
```

- 这个if语句检查file1接口指向的对象实例是否实现了two.IStream接口，如果实现了，则执行特定的代码
- 接口查询是否成功，要在运行期才能够确定。它不像接口赋值，编译器只需要通过静态类型检查即可判断赋值是否可行。

1.4总结(续)

□ 类型查询

- 在Go语言中，还可以直接了当地询问接口指向的对象实例的类型



```
var v1 interface{} = ...  
switch v := v1.(type) {  
case int: // 现在v的类型是int  
case string: // 现在v的类型是string  
...  
}
```

1.4总结(续)

□ 类型查询

■ 类型查询不常用

■ 下面是Println的实现机理，用到了接口查询和类型查询

```
type Stringer interface {  
    String() string  
}  
  
func Println(args ...interface{}) {  
    for _, arg := range args {  
        switch v := v1.(type) {  类型查询  
            case int: // 现在v的类型是int  
            case string: // 现在v的类型是string  
            default:  
                if v, ok := arg.(Stringer); ok {  接口查询 // 现在v的类型是Stringer  
                    val := v.String()  
                    // ...  
                } else {  
                    // ...  
                }  
            }  
        }  
    }  
}
```

□Println()采用穷举法，将每个类型转换为字符串进行打印。对于更一般的情况，首先确定该类型是否实现了String()方法，如果实现了，则用String()方法将其转换为字符串进行打印。

□否则，Println()利用反射功能来遍历对象的所有成员变量进行打印。

1.4总结(续)

□ 接口组合

- 如同之前介绍的匿名类型组合一样，Go语言同样支持接口组合，已知io.Reader接口和io.Writer接口

// ReadWriter接口将基本的Read和Write方法组合起来

```
type ReadWriter interface {
```

```
    Reader
```

```
    Writer
```

```
}
```

□ 可以认为接口组合是类型匿名组合的一个特定场景，只不过接口只包含方法，而不包含任何成员变量

- 这个接口组合了Reader和Writer两个接口，它完全等同于如下写法：

```
type ReadWriter interface {
```

```
    Read(p []byte) (n int, err error)
```

```
    Write(p []byte) (n int, err error)
```

```
}
```

1.4总结(续)

□ Any类型

- 由于Go语言中任何对象实例都满足空接口interface{}，所以interface{}看起来像是可以指向任何对象的Any类型
 - 因为空接口相当于里面有个空方法，对任何类型而言除了实现了自身的普通方法之外，也实现了这个所谓的空方法即也实现了空接口interface{}。因此任何类型的对象都可以赋值给空接口对象即任何对象都可以接入空接口

```
var v1 interface{} = 1 // 将int类型赋值给interface{}  
var v2 interface{} = "abc" // 将string类型赋值给interface{}  
var v3 interface{} = &v2 // 将*interface{}类型赋值给interface{}  
var v4 interface{} = struct{ X int }{1}  
var v5 interface{} = &struct{ X int }{1}
```

1.4总结(续)

□ Any类型

- 当函数可以接受任意的对象实例时，我们会将其声明为 `interface{}`，最典型的例子是标准库 `fmt` 中 `PrintXXX` 系列的函数

```
func Printf(fmt string, args ...interface{})  
func Println(args ...interface{})  
...
```

- `args ...interface{}` 含义是0-n个任意类型的参数
- `fmt string` 一个名为 `fmt` 的 `string` 类型的变量，`fmt` 在 `Printf` 中是个必填参数。
- `Println` 的参数可以为空

1.4总结(续)

- `m.musics = append(m.musics[:index-1], m.musics[index+1:]...)`
 - `append`的原型为：
`append([]interface{}, ...interface{})`
 - 功能：表示可以为任何类型(`[]interface{}`)的切片追加0-n个元素(`...interface{}`)，
 - `m.musics[index+1:]...`的含义就是把切片 `m.musics[index+1:]` 打散成一个一个的可追加的参数

1.5思考

- 编程实现1.3中的仿真音乐播放器

Thank you very much

*Any comments and suggestions
are beyond welcome*