```
print(t_p.pointee.name)    // 会产生运行错误
```

运行上面的代码会产生运行时错误，变量 t 是 Optional 类型的，当将其赋值为 nil 后，其引用的对象即被销毁，内存被释放，后面使用指针再对其进行访问就会产生错误。为了处理这个问题，可以使用 Unmanaged 类对引用计数进行管理，示例如下：

```
class Teacher {
    var name:String
    var subject:String

    init(name:String, subject:String) {
        self.name = name
        self.subject = subject
    }
}
var t_un = Unmanaged.passRetained(t!)              // 进行加引用计数操作
var t_p = t_un.toOpaque()                          // 转换成指针
t = nil
var tt = unsafeBitCast(t_p, to: Teacher.self)  // 将指针强制转换成某个类的实例
print(tt.name)
t_un.release()                                     // 手动释放所增加的引用计数
```

再次运行代码，这次程序将按照预期的效果执行。Unmanaged 对象被称为非托管对象，将普通对象转换为非托管对象可以使用 passRetained 方法或 passUnretained 方法：使用 passRetained 方法会使引用计数增加，需要开发者手动进行管理；使用 passUnretained 方法不会改变引用计数。Unmanaged 实例对象的 toOpaque 方法用来获取对象的指针，其是一个任意类型的指针，类似于 C 语言中 void *类型，unsafeBitCast 方法用来将指针强制转换成指定的类实例。需要注意的是这个方法非常危险，不会校验转换的可行性，需要开发者自行保证类型的一致。使用非托管对象还有一点需要注意，如果用了 passRetained 方法，则一定要在合适的时机调用 release 方法进行内存的释放。

## 3.4.4　Swift 中的 String 类型

Swift 中的字符串数组和字典与 Objective-C 中的这些数据类型的一大区别就是它们都是值类型，通过前面的学习，我们已经可以比较透彻地理解值类型。还有一点需要注意，在 Objective-C 中，NSString、NSArray 和 NSDictionary 都是定义在 Foundation 框架中的数据类型；在 Swift 中，String、Array 和 Dictionary 是定义在 Swift 语言核心库中的数据类型，Foundation 框架中通过扩展的方式对其功能进行了增加。

这些数据类型在 Swift 与 Objective-C 语言中的设计方式的区别在于，Objective-C 更多采用继承的方式实现功能，开发者在需要进行子类功能定制的时候通常需要重写父类的方法；Swift 则更多采用协议的方式实现功能，开发者想要定制功能时需要遵守相关的协议。本节我们就来学习 Swift 中的字符串类型在实际开发中不常用但非常重要的功能。

### 1. 自定义对象描述信息

当使用 print 函数对一个自定义的对象进行打印时，默认会输出这个对象的类型，如果需要自定义对象的描述信息，就需要实现 CustomStringConvertible 协议，CustomStringConvertible 协议中只包含一个 get 属性 description，示例代码如下：

```
class Teacher: CustomStringConvertible {
    var name:String

    var description: String {
        get {
            return "教师对象:\(self.name)"
        }
    }

    init(name:String) {
        self.name = name
    }
}
let t = Teacher(name: "Jaki")
print(t)   // 教师对象:Jaki
```

与 CustomStringConvertible 协议对应的还有 CustomDebugStringConvertible 协议。这个协议提供了 debugDescription 属性，用来自定义在 Debug 环境下的打印信息。

### 2. 字符串迭代器

我们知道使用 for-in 可以对字符串进行快速枚举，这其实就应用到了设计模式中的迭代器模式。在 String 结构体的内部定义了一个名为 Iterator 的内部结构体，实现了 Swift 中的迭代器协议 IteratorProtocol，调用字符串的 makeIterator 方法接口获取到这个内部的迭代器结构体示例，代码如下：

```
var string = "Hello"
var it:String.Iterator = string.makeIterator()
while let c = it.next() {
    print(c)
}
```

使用内部类、内部结构体、内部枚举也是 Swift 语言常用的一种开发思路。在 String 结构体中，除了定义内部的迭代器结构体外，还有内部的字符下标 Index 结构体等。

### 3. 与字符串迭代相关的几个高级方法

map 方法是最常用的字符串处理方法，其需要传入一个闭包参数，字符串依次遍历出的字符会作为闭包的参数，闭包的返回值为处理后的结果。下面的示例代码演示了逐个将字符串的

字符变成大写的方法：

```
var newString = "Hello".map { (c) -> String.Element in
    return c.uppercased().first!
}
print(String(newString))  //HELLO
```

filter 方法用来进行字符串中字符的过滤，其需要传入一个闭包参数，字符串依次遍历出的字符会作为闭包的参数，闭包需要返回一个布尔值，如果返回布尔值 false，则当前字符会被过滤掉，下面的代码会过滤掉字符串中所有的大写字母：

```
var newString = "Hello".filter { (c) -> Bool in
    if c.asciiValue! <= "Z".first!.asciiValue! &&  c.asciiValue! >=
"A".first!.asciiValue! {
        return false
    }
    return true
}
print(newString)  //ello
```

reduce 函数也被称为累加器，其第 1 个参数为累加前的初始结果，第 2 个参数为闭包，闭包中会将上一次执行累加后的结果和遍历出的字符作为参数传入。例如，下面的代码会在字符串的每个字符前插入感叹号：

```
var newString = "Hello".reduce("") { (result, c) -> String in
    return result + "!" + String(c)

}
print(newString)  //!H!e!l!l!o
```

其实，上面列举的方法并非是 String 所独有的。在 Swift 中，集合类型都可以调用这些迭代方法。与 map 方法类似的还有 flatMap 和 compactMap。flatMap 在调用时会将返回的二维集合进行降维，即可以将二维数组中的元素全部合并到一个数组中。compactMap 方法可以自动提出新集合中的 nil 值。

### 3.4.5　Swift 中的 Array 类型

Array 是 Swift 中非常强大的一种数据类型，只要将其声明为变量类型，就可以方便地调用方法对其增删。其实，Array 类型采用了动态扩容的方式实现可变性。示例代码如下：

```
var array:Array<Int> = [1, 2, 3]
print(array.capacity)   // 3
array.append(4)
print(array.capacity)   // 6
array.append(contentsOf: [5, 6, 7])
```