Swift第七节课–闭包

函数类型

之前在代码的书写过程中，我们已经或多或少的接触过函数，函数本身也有自己的类型，它由形式参数类型，返回类型组成。

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

var a = addTwoInts

func addTwoInts(_ a: Double, _ b: Double) -> Double {
    return a + b
}

func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

var a: (Double, Double) -> Double = addTwoInts

a(10, 20)

var b = a

b(20 ,30)
```

什么是闭包

闭包是一个捕获了上下文的常量或者是变量的函数。

```
func makeIncrementer() -> () -> Int {
    var runningTotal = 10
    func incrementer() -> Int {
        runningTotal += 1
        return runningTotal
    }
    return incrementer
}
```

闭包表达式

```
{ (param) -> (returnType)  in
    //do something
}
```

首先按照我们之前的知识积累， `OC` 中的 `Block` 其实是一个匿名函数，所以这个表达式要具备

- 作用域（也就是大括号）
- 参数和返回值
- 函数体（in）之后的代码

`Swift` 中的闭包即可以当做变量，也可以当做参数传递，这里我们来看一下下面的例子熟悉一下：

```
 var closure : (Int) -> Int = { (age: Int)  in
    return age
}
```

同样的我们也可以把我们的闭包声明一个可选类型：

```
//错误的写法
var closure : (Int) -> Int?
closure = nil
//正确的写法
var closure : ((Int) -> Int)?
closure = nil
```

还可以通过 `let` 关键字将闭包声明位一个常量(也就意味着一旦赋值之后就不能改变了)

```
let closure: (Int) -> Int

closure = {(age: Int) in
    return age
}

closure = {(age: Int) in
    return age
}
```

同时也可以作为函数的参数

```
func test(param : () -> Int){
    print(param())
}

var age = 10

test { () -> Int in
```

```
    age += 1
    return age
}
```

尾随闭包

当我们把闭包表达式作为函数的最后一个参数，如果当前的闭包表达式很长，我们可以通过尾随闭包的书写方式来提高代码的可读性。

```
func test(_ a: Int, _ b: Int, _ c: Int, by: (_ item1: Int, _ item2: Int, _ i
   return  by(a, b, c)
}

test(10, 20, 30, by: {(_ item1: Int, _ item2: Int, _ item3: Int) -> Bool in
   return (item1 + item2 < item3)




})
```

其中闭包表达式是 `Swift` 语法。使用闭包表达式能更简洁的传达信息。当然闭包表达式的好处有很多：

- 利用上下文推断参数和返回值类型
- 单表达式可以隐士返回，既省略 `return` 关键字
- 参数名称的简写（比如我们的 $0）
- 尾随闭包表达式

```
var array = [1, 2, 3]

array.sort(by: {(item1 : Int, item2: Int) -> Bool in return item1 < item2 })

array.sort(by: {(item1, item2) -> Bool in return item1 < item2 })

array.sort(by: {(item1, item2) in return item1 < item2 })

array.sort{(item1, item2) in item1 < item2 }

array.sort{ return $0 < $1 }

array.sort{ $0 < $1 }

array.sort(by: <)
```

捕获值

在讲闭包捕获值的时候，我们先来回顾一下 Block 捕获值的情形

```
- (void)testBlock{
    NSInteger i = 1;

    void(^block)(void) = ^{
        NSLog(@"block %ld:", i);
    };

    i += 1;

    NSLog(@"before block %ld:", i);
    block();
    NSLog(@"after block %ld:", i);
}
```

那么如果我们想要外部的修改能够影响当前 block 内部捕获的值，我们只需要对当前的 i 添加 __block 修饰符

```
- (void)testBlock{
    __block NSInteger i = 1;

    void(^block)(void) = ^{
        NSLog(@"block %ld:", i);
    };

    i += 1;

    NSLog(@"before block %ld:", i);
    block();
    NSLog(@"after block %ld:", i);
}
```

回顾一下 `Block` 是怎么处理变量 `i` 的?

OC Block 和 Swift 闭包相互调用

我们在OC中定义的Block，在Swift中是如何调用的那？我们来看一下

```
typedef void(^ResultBlock)(NSError *error);

@interface LGTest : NSObject

+ (void)testBlockCall:(ResultBlock)block;

@end
```

在 Swift 中我们可以这么使用

```
LGTest.testBlockCall{ error in
    let errorcast = error as NSError
```

```
    print(errorcast)
}

func test(_ block: ResultBlock){
       let error = NSError.init(domain: NSURLErrorDomain, code: 400, userInf
       block(error)
}
```

比如我们在 Swift里这么定义，在OC中也是可以使用的

```
class LGTeacher: NSObject{
   @objc static var closure: (() -> ())?
}

+ (void)test{
//    LGTeacher.test{}

    LGTeacher.closure = ^{
         NSLog(@"end");
    };
//    LGTeacher.closure = ^{
//
//    }
}

LGTest.test()

LGTeacher.closure!()
```

闭包的本质

再看具体的内容的时候，我们先来熟悉一下简单的 `IR` 语法：

数组

```
[<elementnumber> x <elementtype>]
//example
alloca [24 x i8], align 8   24个i8都是0

alloca [4 x i32] === array
```

结构体

```
%swift.refcounted = type { %swift.type*, i64 }

//表示形式
%T = type {<type list>}  //这种和C语言的结构体类似
```

指针类型

```
<type> *

//example
```

```
i64*    //64位的整形
```

`getelementptr` 指令

LLVM中我们获取数组和结构体的成员，通过 `getelementptr` ，语法规则如下：

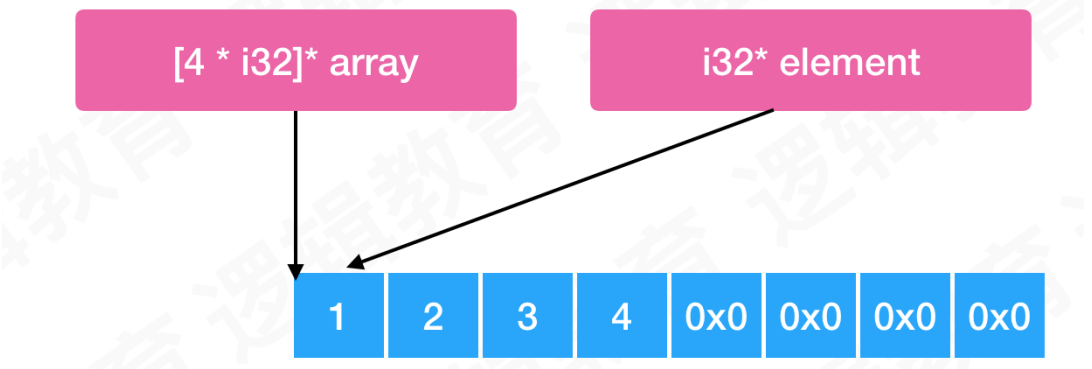
```
<result> = getelementptr <ty>, <ty>* <ptrval>{, [inrange] <ty> <idx>}*
<result> = getelementptr inbounds <ty>, <ty>* <ptrval>{, [inrange] <ty> <id
```

这里我们看 `LLVM` 官网当中的一个例子：

```
struct munger_struct {
  int f1;
  int f2;
};

void munge(struct munger_struct *P) {
  P[0].f1 = P[1].f1 + P[2].f2;
}

getelementptr inbounds %struct.munger_struct,  %struct.munger_struct %1, i64
getelementptr inbounds %struct.munger_struct,  %struct.munger_struct %1, i32


int main(int argc, const char * argv[]) {

    int array[4] = {1, 2, 3, 4};

    int a = array[0];

    return 0;
}
```

其中 `int a = array[0]` 这句对应的LLVM代码应该是这样的：

```
a = getelementptr inbounds [4 x i32], [4 x i32]* array, i64 0, i32 0, i32 0
```



总结：

- 第一个索引不会改变返回的指针的类型，也就是说ptrval前面的*对应什么类型，返回就是什么类型
- 第一个索引的偏移量的是由第一个索引的值和第一个ty指定的基本类型共同确定的。
- 后面的索引是在数组或者结构体内进行索引
- 每增加一个索引，就会使得该索引使用的基本类型和返回的指针的类型去掉一层

defer

定义：

`defer` {} 里的代码会在当前*代码块*返回的时候执行，无论当前*代码块*是从哪个分支 return 的，即使程序抛出错误，也会执行。

如果多个 defer 语句出现在同一作用域中，则它们出现的顺序与它们执行的顺序相反，也就是先出现的后执行。

这里我们看一个简单的例子：

```
func f() {
    defer { print("First defer") }
    defer { print("Second defer") }
    print("End of function")
}
f()
```

defer能做什么?

```
func append(string: String, terminator: String = "\n", toFileAt url: URL) th
    // The data to be added to the file
    let data = (string + terminator).data(using: .utf8)!

    // If file doesn't exist, create it
    guard FileManager.default.fileExists(atPath: url.path) else {
        try data.write(to: url)
        return
    }


    // If file already exists, append to it
    let fileHandle = try FileHandle(forUpdating: url)
    fileHandle.seekToEndOfFile()
    fileHandle.write(data)
    fileHandle.closeFile()
}

let url = URL(fileURLWithPath: NSHomeDirectory() + "/Desktop/swift.txt")
try append(string: "iOS面试突击", toFileAt: url)
try append(string: "Swift", toFileAt: url)
```

这里有时候如果当前方法中多次出现 `closeFile` ,那么我们就可以使用 `defer`

```
 func append(string: String, terminator: String = "\n", toFileAt url: URL) t
    // The data to be added to the file
    let data = (string + terminator).data(using: .utf8)!

     defer{
        fileHandle.closeFile()
     }

    // If file doesn't exist, create it
    guard FileManager.default.fileExists(atPath: url.path) else {
        try data.write(to: url)
        return
    }

    // If file already exists, append to it
    let fileHandle = try FileHandle(forUpdating: url)
    fileHandle.seekToEndOfFile()
    fileHandle.write(data)

}

let url = URL(fileURLWithPath: NSHomeDirectory() + "/Desktop/swift.txt")
try append(string: "iOS面试突击", toFileAt: url)
try append(string: "Swift", toFileAt: url)try append(string: "Line 1", toFil
try append(string: "Line 2", toFileAt: url)
```

比如我们在使用指针的时候

```
let count = 2
let pointer = UnsafeMutablePointer<Int>.allocate(capacity: count)
pointer.initialize(repeating: 0, count: count)

defer {
    pointer.deinitialize(count: count)
    pointer.deallocate()
}
```

比如我们在进行网络请求的时候，可能有不同的分支进行回调函数的执行

```
func netRquest(completion: () -> Void) {
  defer {
    self.isLoading = false
    completion()
  }
  guard error == nil else { return }
}
```