补充材料:

1)分程序结构&Lambda表达式 2)面向对象语言的编译方法

> 史晓华 北京航空航天大学 2023-9

SWIFT语言是一种分程序结构语言

• Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

为什么要使用DISPLAY区处理分程序结构 对外层变量的引用?

```
func A(){
         var int localA;
         func B(){
                   ... C(); ...
                   localA = ....
         func C(){
                   ... B() ; ...
                   localA = ...
         ...B()...
```

·如果不使用DISPLAY区,在B()或者C()中如何访问localA?

Lambda表达式(Closures)是分程序吗?

- Lambda表达式可以理解为一种"匿名"的内联函数
 - 不同语言的Lambda表达式的设计和实现不同,有的(有时)可以内联,有的需要动态调用
- C++为例

 auto ptr = []() {cout << "hello" << endl; }; //lambda表达式

```
auto ptr = []() {cout << nello << endi; }; //lambda表达式 ptr();//调用函数
```

```
//带参数的lambda表达式
auto fun = [](int x, int y)->int {cout << x + y << endl; return y;};
// 这里的->后面写的是返回值类型
auto z=fun(3, 4);
```

C++Lambda表达式变量作用域

- [captures] (params) mutable-> type{...}
 - 在 lambda 表达式引出操作符[]里的"captures"称为"捕获列表"。可以捕获表达式外部作用域的变量,在函数体内部直接使用。这种语言设计在编译时可以规避DISPLAY区
 - Java语言支持的Lambda表达式在访问外部变量时要求被访问变量为final类型,同样可以起到规避DISPLAY的作用
- 捕获列表里可以有多个捕获选项,以逗号分隔,使用了略微"新奇"的语法,规则如下
 - [] : 无捕获, 函数体内不能访问任何外部变量
 - [=] :以值(拷贝)的方式捕获所有外部变量,函数体内可以访问,但是不能修改。
 - [&] : 以引用的方式捕获所有外部变量, 函数体内可以访问并修改(需要当心无效的引用);
 - [var] : 以值(拷贝)的方式捕获某个外部变量,函数体可以访问但不能修改。
 - [&var]: 以引用的方式获取某个外部变量, 函数体可以访问并修改
 - [this] : 捕获this指针,可以访问类的成员变量和函数,
 - [=, &var]: 引用捕获变量var, 其他外部变量使用值捕获。
 - [&, var]: 只捕获变量var, 其他外部变量使用引用捕获。

分程序结构和Lambda表达式

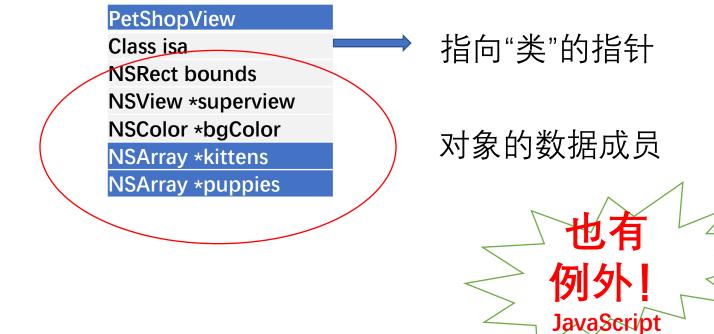
- lambda表达式是分程序结构在软件工程和编程方法意义上的进阶版,但是lambda表达式不完全等同于传统分程序结构
 - 它们对外层变量的引用方式上有较大不同,编译方法也不一样
- lambda表达式在不同语言中的设计不同,这也会导致编译方法的 差异较大,但一般倾向于低负载的调用机制
 - 规避DISPLAY
 - 内联

面向对象语言的编译技术

- 针对面向对象语言的3个主要基本特征的编译方法
 - 封装
 - 继承
 - 多态

对象的封装

- 隐藏对象的属性和实现细节, 仅对外提供公共访问方式
- 编译器对对象的数据管理机制



PetShopView VTable method0 method1_OV method2

method3_OV

method4

method5

类的方法: **虚表**

JavaScript的对象和类

uint32[] propHashtable Hashtable of properties get_field ".baz" JSValue BEFORE object starts JSS rape JSObject. JSObject JSShape [GC header] [GC header] **JSProperty** foo. bool is lashed uint16 classId Data Union "2" .bar bool may HareFastArray JSShape* shape JSValue value 10086 .baz uint3 l hash JSProperty* prop JSVarRef* empty empty uint32 ropCount Lweak references] JSObject. JSShape represents the order JSObject prototype getter, setter Data Union of object properties List of properties autoinit JSShapeProperty Formany uint32 next_hash closure Data structures JSFunctionBytecode* uint32 size JSTypedArray* ta for special classes JSAtom atom bytecode e.g. Typed Array JSValue* values uint 8* values ... JSVarRef** varRefs uint32 count JSObject * homeOnj

对象的继承

• 成员和方法都可以继承

NSView	ı (Leopard)
0	Class isa
4	NSRect bounds
20	NSView *superview
24	NSColor *bgColor

PetShopView		
Class isa		
NSRect bounds		
NSView *superview		
NSColor *bgColor		
NSArray *kittens		
NSArray *puppies		

NSView (Leopard) Vtable		
0 method0		
4 method1		
8 method2		
12 method3		

PetShopView VTable
method0
method1_OV
method2
method3_OV
method4
method5

对象成员

方法虚表

多态的实现

• 基于虚表的实现

NSView (Leopard) Vtable		
0 method0		
4method1		
8 method2		
12 method3		

PetShopView VTable
method0
method1_OV
method2
method3_OV
method4
method5

虚表

//call obj.method3()

mov ecx, [ebp+8] //ecx = obj mov eax, [ecx] //eax = obj.vtable call [eax+12] //call obj.vtable[3]



Objective-C Method Dispatching

- Clang编译器将所有函数调用转为对 objc_msgSend()的调用
 - passing an object, method selector and parameters as arguments
 - 例, [object message: param] 将变为: objc_msgSend(object, @selector(message:), param, nil)
 - objc_msgSend() 处理Object-C中所有method dispatching

```
    汇编实现、开源
id objc_msgSend ( id obj, SEL op, ... ){
        Class c = object_getClass(obj);
        IMP imp = CacheLookup(c, op);
        if (!imp) {
            imp = class_getMethodImplementation(c, op);
        }
        jump imp(obj, op, ...);
}
```

