第一章 熟悉OC

第一条：了解Objective-C语言的起源

OC这门语言使用“消息结构”，而非“函数调用”方式，因为OC是由Smalltalk演化而来的，它是消息型语言的鼻祖。它的方法一般像一个句子一样，融合参数，所以显得并不简洁，但是使用习惯之后便可以发现它的语法是比较清楚的。

*消息型语言和函数调用型语言的区别在于：消息结构语言在执行时所应执行的代码由运行环境所决定；而函数调用型语言是由编译器决定的。消息结构语言会在运行时检测所要执行的方法。OC运行时库中包含其面向对象特性所要的全部数据类型和函数。运行时组件的本质是一种与开发者所编写代码相链接的动态库，可以把开发者编写的代码粘合起来。

要点：

1. Objecive-C为C语言添加了面向对象的特性，是其超集。OC使用动态绑定的消息结构，也就是说，在运行时会检测对象的类型。收到一条消息之后究竟该执行何种代码是由运行时环境而非编译器来决定的。
2. 理解C语言的核心概念有助于写好OC程序，尤其要掌握内存模型和指针。

第二条：在类的头文件中尽量少引用其他头文件

将引用头文件的时机尽量后托，这样可以减少程序编译时间。可以使用“向前声明”方法（将#import "Class.h"换成@class Class），这样可以暂时屏蔽掉该类的一些实现细节。并且还有可能造成循环引用问题。

要点：

1. 除非确实需要，否则不要引入头文件。一般来说，应该用向前声明提及某个类，并在实现文件中引入其头文件。这样可以尽可能降低类的耦合。
2. 有时候不能使用向前声明，比如要声明该类遵循某项协议，在这种情况下，尽量把此声明移至分类中，或者将此协议放在一个单独的头文件中。

第三条：多用字面量语法，少用与之等价的方法

字面语法就是尽量少用alloc init方法常见一个对象，用OC Foundation框架中提供的更加简便、更加明显的方法创建对象。

如：

```
// 字符串
NSString *string = @"Effective Objective 2.0";
NSString *string = [[NSString alloc] initWithFormat:@"Effective Objective-C2.0"];
// 字面数值
NSNumber *intNumber = @1;
NSNumber *intNumber = [NSNumber numberWithInt:1];
// 字面量数组
NSArray *array = @[@"dog",@"cat",@"fish"];
NSArray *array = [NSArray arrayWithObjects:@"dog",@"cat",@"fish",nil];
array[i]    [array objectAtIndex:i]
```

不过用字面量语法创建数组时，如果遇到一个nil，将会抛出一个异常，因为字面量语法实际上是一种“语法糖”，其等效于先创建一个数组，再将所有的对象加入进去。抛出的异常是这样的：

```
***Terminating app due to uncaught exception
'NSInvalidArgumentException',reason:'***'
-[__NSPlaceholderArray initWithObjecets:count:]:attempt to insert nil object from
```

然而这样异常有时候对程序确实有用的。

字面量字典

```
NSDictionary *dict = @{@"key1":@"value1",@"key2":@"value2"};
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:@"key1","value1",
```

同样当key值遇见nil时还会抛出异常。

要点：

1. 应该使用字面量语法创建字符串，数值，数组，字典；
2. 通过下标操作符来取值或者设值；
3. 用字面量语法创建数组或者字典时，若遇到nil会抛出异常，因此务必保证操作的数中不含nil。

第四条：多用类型常量，少用#define预处理指令

#define LENGTH 100

这条预处理指令设置了一个常量LENGTH，其值为100，这样定义没有错，但是这个LENGTH常量没有类型，一方面会对程序的可读性产生影响；另一方面，如果定义在头文件中，由于OC中没有命名空间这一概念，就相当于定义了一个全局变量，这就有可能覆盖掉其他相同名称的宏定义。所以如果此常量只局限于某个编译单元，可以定义在其实现文件中，并且在前面加一个字母“k”或者其他字母；若是在类外可见，则加上类名作为前缀。

有时候，需要对外开放某个常量，比如某个对象给其他对象派发通知NSNotification,在注册通知的时候会用到通知的名称，因为外面监听通知的对象也要用到此名称，所以可以（在头文件中声明，实现文件中实现）一个外部可见的全局NSSting常量,它的名称一般以与之相关的类名为前缀（系统框架就是这么做）。

```
// in the header file:
extern NSString *const someNotificationName ;
in the implemention file:
NSString *const  someNotificationName = @"someName";
```

要点：

1. 不要使用预处理指令定义常量，这样定义的常量不含类型信息，编译器只是会在编译前执行查找和替换，即使有人重新定义也不会报错。
2. 在实现文件中使用static const 来定义“只在编译单元内可见的常量”，由于此类常量不在全局符号表中，所以无需为名称加前缀；
3. 在头文件中使用extern类声明全局常量，并在相关实现文件中定义值，这种常量会出现在全局符号表中，需要使用类名作为前缀。

第五条：使用枚举表示状态、选项和状态码

枚举是一种常量命名方式，每个对象所经历的各种状态便可以定义为一个简单的枚举。

定义枚举类型的方式是：

```
enum NeworkingState:NSInteger{
NetworkingStateConnecting,//==0
NetworkingStateConnected,
NetworkingStateDisconnected,
};
```

但是定义枚举却比较麻烦，因为定义每个对象时不仅要此枚举对象名，还有在前面加上enum关键字。

```
enum NetworkingState state = NetworkingStateConnected;
// 可以使用下面的方式简写：
typedef (enum NetworkingState) NetworkingState;
// 这样每次就不用既写enum又写枚举名
```

Foundation框架中定义了一些宏，在定义相关数据类型时也可以指定底层的数据类型，这些宏具有向后兼容的能力（如果目标平台支持新标准，将采用新标准，否则改用旧式语法）。

```
typedef NS_ENUM(NSINteger,NetworkingState) {
NetworkingStateConnecting,//==0
NetworkingStateConnected,
NetworkingStateDisconnected,
};
typedef NS_OPTIONS(NSInteger,WRDeviceDirection) {
WRDeviceDirectionTop                =1<<0 ,
WRDeviceDirectionRight              =1<<1,
WRDeviceDirectionBottom     =1<<2,
WRDeviceDirectionLeft               =1<<3,
}
```

要点：

1. 应该用枚举表示状态机的状态，传递给方法的选项以及状态码等值，给这些值取个易懂的名字；

2. 如果把传递给某个方法的选项表示为枚举类型，而多个选项又可以同时使用，那么就将各选项定义为2的幂，以便通过按位或操作将其组合起来；

3. 用NS_ENUM和NS_OPTIONS宏来定义枚举类型，并指明底层数据类型，这样可以保证枚举是用开发者选择的底层数据类型实现出来的，而不是采用编译器所选的类型。