Effective Objective-C
nate

- 语法
- 对象、消息、运行时
- 接口和API设计
- 协议和分类
- 内存管理
- block GCD
- 框架

4 多用类型常量，少用#define

要点

- 不要用预处理指令定义常量。这样定义出来的常量不含类型信息，编译器只是会在编译前据此执行查找与替换操作。即使有人重新定义了常量值，编译器也不会产生警告信息，这将导致应用程序中的常量值不一致。
- 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”（translation-unit-specific constant）。由于此类常量不在全局符号表中，所以无须为其名称加前缀。
- 在头文件中使用 extern 来声明全局常量，并在相关实现文件中定义其值。这种常量要出现在全局符号表中，所以其名称应加以区隔，通常用与之相关的类名做前缀。

6 理解属性

实例变量

```
@interface EOCPerson : NSObject {
@public
    NSDate *_dateOfBirth;
    NSString *_firstName;
    NSString *_lastName;
@private
    NSString *_someInternalData;
}
@end
```

属性

用以访问结定类型中具有结定名称的变量。例如下面这个类：

```
@interface EOCPerson : NSObject
@property NSString *firstName;
@property NSString *lastName;
@end
```

对于该类的使用者来说，上述代码写出来的类与下面这种写法等效：

```
@interface EOCPerson : NSObject
- (NSString*)firstName;
- (void)setFirstName:(NSString*)firstName;
- (NSString*)lastName;
- (void)setLastName:(NSString*)lastName;
@end
```

自动合成

```
@implementation EOCPerson
@synthesize firstName = _myFirstName;
@synthesize lastName = _myLastName;
@end
```

前述语法会将生成的实例变量命名为 _myFirstName 与 _my

11 消息发送和方法调用

```
id returnValue = objc_msgSend(someObject,
                              @selector(messageName:),
                              parameter);
```

objc_msgSend 函数会依据接收者与选择子的类型来调用适当的方法。为了完成此操作，该方法需要在接收者所属的类中搜寻其“方法列表”（list of methods），如果能找到与选择子名称相符的方法，就跳至其实现代码。若是找不到，那就沿着继承体系继续向上查找，等找到合适的方法之后再跳转。如果最终还是找不到相符的方法，那就执行“消息转发”（message forwarding）操作。消息转发将在第 12 条中详解。

- 消息由接收者、选择子及参数构成。给某对象“发送消息”（invoke a message）[⊖] 也就相当于在该对象上“调用方法”（call a method）。
- 发给某对象的全部消息都要由“动态消息派发系统”（dynamic message dispatch system）来处理，该系统会查出对应的方法，并执行其代码。

24 类代码分散到分类中

```
#import <Foundation/Foundation.h>

@interface EOCPerson : NSObject
@property (nonatomic, copy, readonly) NSString *firstName;
@property (nonatomic, copy, readonly) NSString *lastName;
@property (nonatomic, strong, readonly) NSArray *friends;

- (id)initWithFirstName:(NSString*)firstName
            andLastName:(NSString*)lastName;
@end

@interface EOCPerson (Friendship)
- (void)addFriend:(EOCPerson*)person;
- (void)removeFriend:(EOCPerson*)person;
- (BOOL)isFriendsWith:(EOCPerson*)person;
@end

@interface EOCPerson (Work)
- (void)performDaysWork;
- (void)takeVacationFromWork;
@end

@interface EOCPerson (Play)
- (void)goToTheCinema;
- (void)goToSportsGame;
@end
```

28 通过协议提供匿名对象

```
@protocol EOCDatabaseConnection
- (void)connect;
- (void)disconnect;
- (BOOL)isConnected;
- (NSArray*)performQuery:(NSString*)query;
@end
```

然后，就可以用“数据库处理器”（database handler）单例来提供数据库连接了。这个单例的接口可以写成：

```
#import <Foundation/Foundation.h>

@protocol EOCDatabaseConnection;

@interface EOCDatabaseManager : NSObject
+ (id)sharedInstance;
- (id<EOCDatabaseConnection>)connectionWithIdentifier:
                                (NSString*)identifier;
@end
```

- 协议可在某种程度上提供匿名类型。具体的对象类型可以淡化成遵从某协议的 id 类型，协议里规定了对象所应实现的方法。
- 使用匿名对象来隐藏类型名称（或类名）。
- 如果具体类型不重要，重要的是对象能够响应（定义在协议里的）特定方法，那么可使用匿名对象来表示。

面向对象-面向协议

```
class Animal {
    var leg: Int { return 2 }
    func eat() {
        print("eat food.")
    }
    func run() {
        print("run with \(leg) legs")
    }
}

class Tiger: Animal {
    override var leg: Int { return 4 }
    override func eat() {
        print("eat meat.")
    }
}

let tiger = Tiger()
tiger.eat() // "eat meat"
tiger.run() // "run with 4 legs"
```

```
protocol Greetable {
    var name: String { get }
    func greet()
}
```

```
struct Person: Greetable {
    let name: String
    func greet() {
        print("你好 \(name)")
    }
}
Person(name: "Wei Wang").greet()
```

实现很简单，`Person` 结构体通过实现 `name` 和 `greet` 来满足 `Greetable`。在调用时，我们就可以使用 `Greetable` 中定义的方法了。

动态派发安全性

除了 `Person`，其他类型也可以实现 `Greetable`，比如 `Cat`：

```
struct Cat: Greetable {
    let name: String
    func greet() {
        print("meow~ \(name)")
    }
}
```

26 勿在分类中声明属性

```
static const char *kFriendsPropertyKey = "kFriendsPropertyKey";

@implementation EOCPerson (Friendship)

- (NSArray*)friends {
    return objc_getAssociatedObject(self, kFriendsPropertyKey);
}

- (void)setFriends:(NSArray*)friends {
    objc_setAssociatedObject(self,
                             kFriendsPropertyKey,
                             friends,
                             OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

@end
```

这样做可行，但不太理想。要把相似的代码写很多遍，而且在内存管理问题上容易出错，因为我们在为属性实现存取方法时，经常会忘记遵从其内存管理语义。比方说，你可能通过属性特质（attribute）修改了某个属性的内存管理语义。而此时还要记得，在设置方法中也得修改设置关联对象时所用的内存管理语义才行。所以说，尽管这个做法不坏，但笔者并不推荐。

此外，你可能会选用可变数组来实现 friends 属性所对应的实例变量。若是这样做，就得

```
@interface NSCalendar (EOC_Additions)
@property (nonatomic, strong, readonly) NSArray *eoc_allMonths;
@end

@implementation NSCalendar (EOC_Additions)
- (NSArray*)eoc_allMonths {
    if ([self.calendarIdentifier
            isEqualToString:NSGregorianCalendar])
    {
        return @[@"January", @"February",
                 @"March", @"April",
                 @"May", @"June",
                 @"July", @"August",
                 @"September", @"October",
                 @"November", @"December"];
    } else if ( /* other calendar identifiers */ ) {
        /* return months for other calendars */
    }
}
@end
```

34 自动释放池，降低内存峰值

```
NSArray *databaseRecords = /* ... */;
NSMutableArray *people = [NSMutableArray new];
for (NSDictionary *record in databaseRecords) {
    EOCPerson *person = [[EOCPerson alloc]
                         initWithRecord:record];
    [people addObject:person];
}
```

EOCPerson 的初始化函数也许会像上例那样，再创建出一些临时对象。若记录有很多条，则内存中也会有很多不必要的临时对象，它们本来应该提早回收的。增加一个自动释放池即可解决此问题。如果把循环内的代码包裹在“自动释放池块”中，那么在循环中自动释放的对象就会放在这个池，而不是线程的主池里面。例如：

```
NSArray *databaseRecords = /* ... */;
NSMutableArray *people = [NSMutableArray new];
for (NSDictionary *record in databaseRecords) {
    @autoreleasepool {
        EOCPerson *person =
            [[EOCPerson alloc] initWithRecord:record];
        [people addObject:person];
    }
}
```

加上这个自动释放池之后，应用程序在执行循环时的内存峰值就会降低，不再像原来那么高了。内存峰值（high-memory waterline）是指应用程序在某个特定时段内的最大内存用量（highest memory footprint）。新增的自动释放池块可以减少这个峰值，因为系统会在块的末尾把某些对象回收掉。而刚才提到的那种临时对象，就在回收之列。

自动释放池机制就像“栈”（stack）一样，系统创建好自动释放池之后，就将其推入栈中

```objc
- (BOOL)validateDictionary:(NSDictionary *)dict usingChecker:(Checker *)checker
                                                      error:(NSError **)error {
  __block BOOL isValid = YES;
  [dict enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
    if ([checker checkObject:obj forKey:key]) return;
    *stop = YES; isValid = NO;
    if (error) *error = [NSError errorWithDomain:...];
  }];
  return isValid;
}
```

41多用派发队列 少用同步锁

用内置的“同步块”(synchronization block):

```
- (void)synchronizedMethod {
    @synchronized(self) {
        // Safe
    }
}
```

另一个办法是直接使用 NSLock 对象:

```
_lock = [[NSLockalloc] init];

- (void)synchronizedMethod {
    [_lock lock];
    // Safe
    [_lock unlock];
}
```

2

效率问题

刚才说过，滥用 @synchronized（self）会很危险，因为所有同步块都会彼此抢夺同一个锁。要是有很多个属性都这么写的话，那么每个属性的同步块都要等其他所有同步块执行完毕才能执行，这也许并不是开发者想要的效果。我们只是想令每个属性各自独立地同步。

顺便说一下，这么做虽然能提供某种程度的“线程安全”（thread safety），但却无法保证访问该对象时绝对是线程安全的。当然，访问属性的操作确实是“原子的”。使用属性时，必定能从中获取到有效值，然而在同一个线程上多次调用获取方法（getter），每次获取到的

⊖ 也称“重入锁”。——译者注

结果却未必相同。在两次访问操作之间，其他线程可能会写入新的属性值。

串行同步队列

```
_syncQueue =
dispatch_queue_create("com.effectiveobjectivec.syncQueue", NULL);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_sync(_syncQueue, ^{
        _someString = someString;
    });
}
```

然而还可以进一步优化。设置方法并不一定非得是同步的。设置实例变量所用的块，并不需要向设置方法返回什么值。也就是说，设置方法的代码可以改成下面这样：

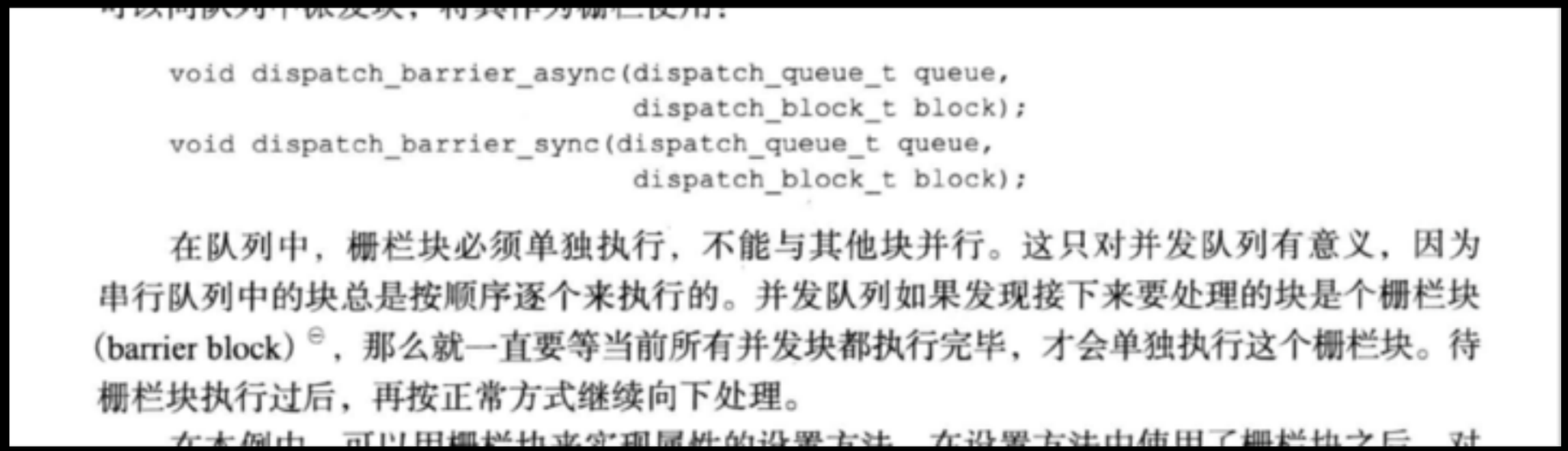
```
- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

并发队列

```
_syncQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

像现在这样写代码，还无法正确实现同步。所有读取操作与写入操作都会在同一个队列上执行，不过由于是并发队列，所以读取与写入操作可以随时执行。而我们恰恰不想让这些

dispatch_barrier



```
void dispatch_barrier_async(dispatch_queue_t queue,
                            dispatch_block_t block);
void dispatch_barrier_sync(dispatch_queue_t queue,
                           dispatch_block_t block);
```

在队列中，栅栏块必须单独执行，不能与其他块并行。这只对并发队列有意义，因为串行队列中的块总是按顺序逐个来执行的。并发队列如果发现接下来要处理的块是个栅栏块(barrier block)[⊖]，那么就一直要等当前所有并发块都执行完毕，才会单独执行这个栅栏块。待栅栏块执行过后，再按正常方式继续向下处理。

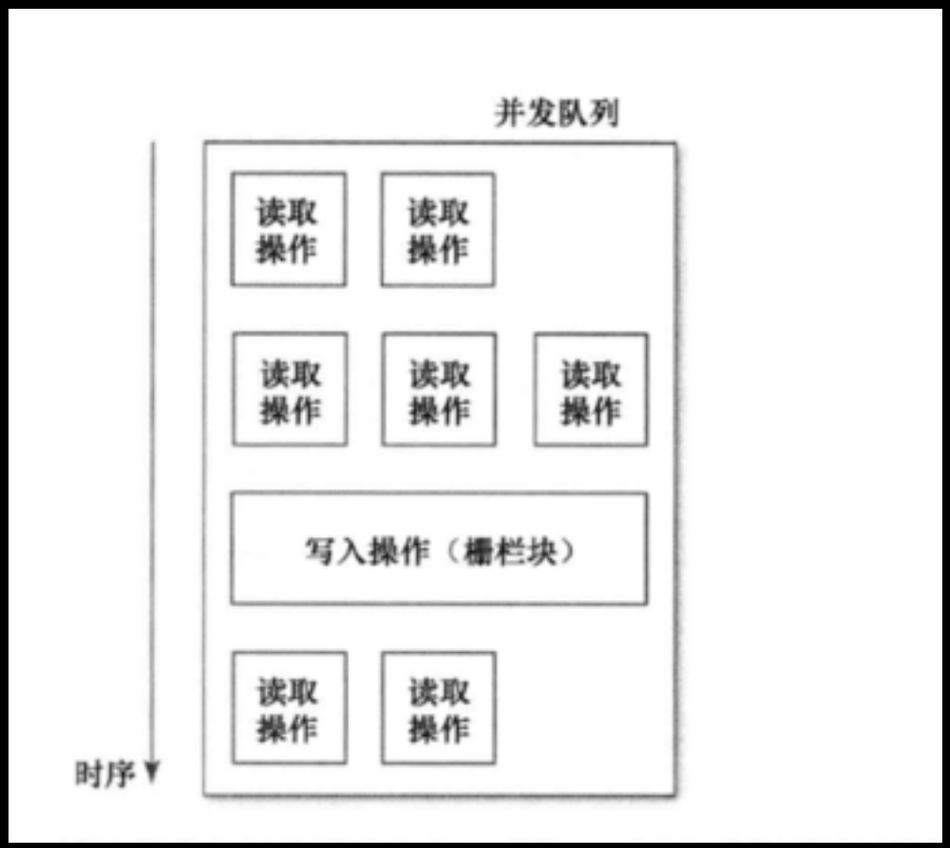
实现代码很简单：

```
_syncQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0);

- (NSString*)someString {
    __block NSString *localSomeString;
    dispatch_sync(_syncQueue, ^{
        localSomeString = _someString;
    });
```

⊖ “barrier”一词也称“阻断器”、“障碍”、“屏障”。——译者注


```
    return localSomeString;
}

- (void)setSomeString:(NSString*)someString {
    dispatch_barrier_async(_syncQueue, ^{
        _someString = someString;
    });
}
```

并发队列


并发队列
读取操作
读取操作
读取操作
读取操作
读取操作
写入操作（栅栏块）
读取操作
读取操作
时序

42 多用GCD，少用 performSelector

```
SEL selector;
if ( /* some condition */ ) {
    selector = @selector(newObject);
```



```
} else if ( /* some other condition */ ) {
    selector = @selector(copy);
} else {
    selector = @selector(someProperty);
}
id ret = [object performSelector:selector];
```

例如，要延后执行某项任务，可以有下面两种实现方式，而我们应该优先考虑第二种：

```
//Using performSelector:withObject:afterDelay:
[self performSelector:@selector(doSomething)
           withObject:nil
           afterDelay:5.0];

//Using dispatch_after
dispatch_time_t time = dispatch_time(DISPATCH_TIME_NOW,
                                     (int64_t)(5.0 * NSEC_PER_SEC));
dispatch_after(time, dispatch_get_main_queue(), ^(void){
    [self doSomething];
});
```

想把任务放在主线程上执行，也可以有下面两种方式，而我们还是应该优选后者：

```
//Using performSelectorOnMainThread:withObject:waitUntilDone:
[self performSelectorOnMainThread:@selector(doSomething)
                       withObject:nil
                    waitUntilDone:NO];

//Using dispatch_async
//(or if waitUntilDone is YES, then dispatch_sync)
dispatch_async(dispatch_get_main_queue(), ^{
    [self doSomething];
});
```

不要使用

dispatch_get_current_queue

不可重入的函数

```
dispatch_queue_t queueA =
  dispatch_queue_create("com.effectiveobjectivec.queueA", NULL);
dispatch_queue_t queueB =
  dispatch_queue_create("com.effectiveobjectivec.queueB", NULL);

dispatch_sync(queueA, ^{
```



```
  dispatch_sync(queueB, ^{
      dispatch_sync(queueA, ^{
          // Deadlock
      });
  });
});
```

图 6-4　派发队列层级体系

```
dispatch_sync(queueA, ^{
    dispatch_sync(queueB, ^{
        dispatch_block_t block = ^{ /* ... */ };
        if (dispatch_get_current_queue() == queueA) {
            block();
        } else {
            dispatch_sync(queueA, block);
        }
    });
});
```

然而这样做依然死锁，因为 dispatch_get_current_queue 返回的是当前队列，在本例中就是 queueB。这样的话，针对 queueA 的同步派发操作依然会执行，于是和刚才一样，还是死锁了。

```
dispatch_queue_t queueA =
  dispatch_queue_create("com.effectiveobjectivec.queueA", NULL);
dispatch_queue_t queueB =
  dispatch_queue_create("com.effectiveobjectivec.queueB", NULL);
dispatch_set_target_queue(queueB, queueA);

static int kQueueSpecific;
CFStringRef queueSpecificValue = CFSTR("queueA");
dispatch_queue_set_specific(queueA,
                            &kQueueSpecific,
                            (void*)queueSpecificValue,
                            (dispatch_function_t)CFRelease);

dispatch_sync(queueB, ^{
    dispatch_block_t block = ^{ NSLog(@"No deadlock!"); };

    CFStringRef retrievedValue =
                    dispatch_get_specific(&kQueueSpecific);
    if (retrievedValue) {
        block();
    } else {
        dispatch_sync(queueA, block);
    }
});
```

- dispatch_get_current_queue 函数的行为常常与开发者所预期的不同。此函数已经废弃，只应做调试之用。
- 由于派发队列是按层级来组织的，所以无法单用某个队列对象来描述“当前队列”这一概念。
- dispatch_get_current_queue 函数用于解决由不可重入的代码所引发的死锁，然而能用此函数解决的问题，通常也能改用“队列特定数据”来解决。

“事不过三、三则重构”

–Martin Fowler

谢谢