block和GCD是两种不同的技术，但是他们是一并引入的。

block是一种可在C、C++以及Objective-C代码中使用的“词法闭包”，它极其有用，这主要是因为借由此机制，开发者可以将代码像对象一样传递，令其在不同环境下运行。

GCD是一种与block有关的技术，它提供了对线程的抽象，而这种抽象则是基于派发队列（dispatch quene）。开发者可将block排入队列中，由GCD负责所有的调度事宜，GCD会根据系统资源情况，适时地创建、复用、摧毁后台线程，以便处理每个队列。

第三十七条：理解block这一概念

block的基础知识

定义形式：return_type (^block_name)(params)

实现形式：block_name = ^(params){ some implementation };

block的强大之处在于在声明它的范围里面，所有变量都可以为其捕获，即block内外的变量均可。

如果block捕获的变量是对象类型，那么就会自动保留它。系统在释放这个block的时候，也会将其一并释放。所以block本身可以视为对象，其他Objective-C可以响应的方法block也可以响应。同样block也有引用计数机制。

如果将block定义在Objective-C类的实例方法中，block除了可以访问所有的实例变量之外，还可以访问self变量，block总是可以修改实例变量，所以不用加**__block**关键字。**然而一定要注意，在block内访问实例变量或者属性由于会捕捉self变量，同时也可以将其retain，这样如果self所指代的对象也保留了block，就会造成循环引用问题。**

block的内部结构

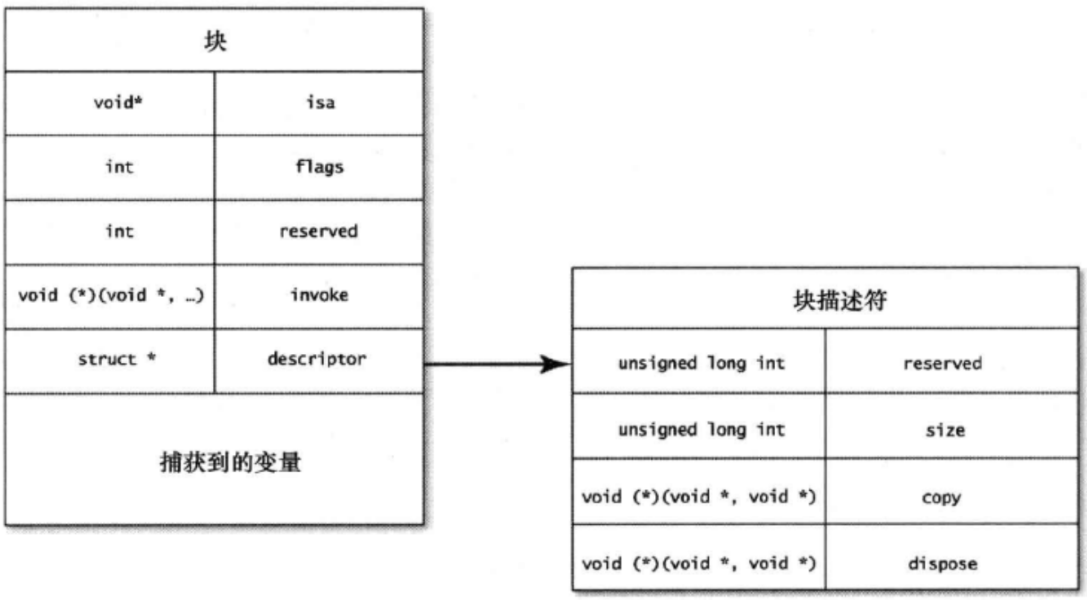

图 6-1　块对象的内存布局

ISA指针指向block对象，invoke指针指向实际的实现代码

全局、栈和堆block

定义block时，所占用的内存区域是在**栈中分配**的，也就是说block只在定义它的范围内有效。
下面的代码就很危险：

```
void (^block)();
if (some condition) {
        block = ^{     ........    };
} else {
        block = ^{     .........   };
}
block();
```

定义在if及else语句中的两个块都分配在栈内存中。编译器会给每个块分配好栈内存,然而等离开了相应的范围之后,编译器有可能把分配给块的内存覆写掉。于是,这两个块只能保证在对应的if或else语句范围内有效。
这样写出来的代码可以编译,但是运行起来时而正确,时而错误。若编译器未覆写待执行的块,则程序照常运行,若覆写,则程序崩溃。
为解决此问题,**可给块对象发送copy消息以拷贝之**。这样的话,就可以把块从栈复制到堆了。
拷贝后的块,可以在定义它的那个范围之外使用。而且,一旦复制到堆上,块就成了带引用计数的对象了。后续的复制操作都不会真的执行复制,只是递增块对象的引用,计数。如果不再使用这个块,那就应将其释放,在ARC环境下会自动释放,而手动管理引用计数时则需要自己来调用release方法。当引用计数降为0后,“分配在堆上的块” (heapblock)会像其他对象一样,为系统所回收。而“分配在栈上的块” (stack block)则无须明确释放,因为栈内存本来就会自动回收,刚才那段范例代码之所以有危险,原因也在于此。
明白这一点后,我们只需给代码加上两个copy方法调用,就可令其变得安全了:

```
void (^block)();
if (some condition) {
        block = [^{     ........    } copy];
} else {
        block = [^{     .........   } copy];
}
block();
```

全局block是不会捕捉任何状态信息，运行时也无须状态参与进来，它所使用的内存区域在编译器已经完全确定了，因为全局block创建在全局内存中，而不是在每次使用的时候创建在栈中，因此适当的全局block可以优化代码和程序性能。另外全局block的copy操作是空操作，因为全局block不能被系统所回收，相当于单例。下面就是全局block

```
void (^block)() = ^ {
```

```
        NSLog(@"this is a global block")
}
```

要点：

1. block是C，C++，Objective-C中的**闭包**；
2. block可接受参数，也可以返回值；
3. block可以分配在栈或者堆上，也可以是全局的。分配在栈中的block可以拷贝到堆里，这样就可以和标准的Objective-C对象一样，具备引用计数了。

第三十八条：为常用的block类型创建typedef

要点：

1. 以typedef重新定义block类型，可令block变量用起来更加简单；
2. 定义新类型时应该遵循现有的命名习惯，勿使其名称与别的名称相冲突；
3. 不妨为同一个block签名定义多个类型的别名。如果要重构的代码使用了block类型的某个别名，那么只需修改相应typedef中的block签名即可，无须改动其他typedef。

第三十九条：用handle block降低代码分散程度

要点：

1. 在创建对象时，可以使用内联的handler block将相关业务逻辑一并声明；
2. 在有多个实例需要监控时，如果采用委托模式，那么经常需要根据传入的对象来切换，而若采用handler block来实现，则可以直接将相关对象放在一起；
3. 设计API时，如果用到handler block，那么可以增加一个参数，使调用者可以通过此参数来决定应该把block安排在那个队列上执行。

第四十条：用block引用其所属对象时不要出现循环引用（retain cycle）

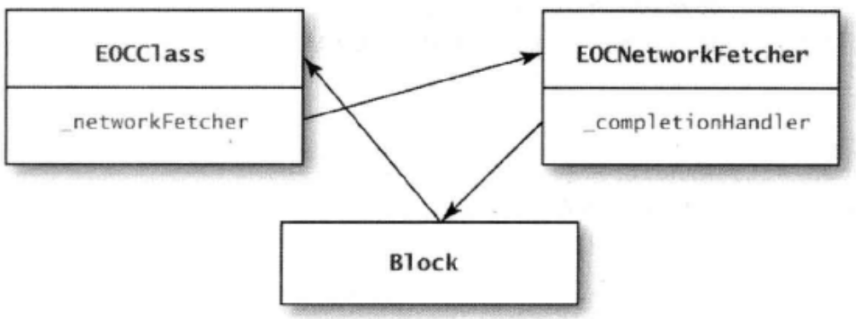

图 6-2　网络数据获取器和拥有它的 EOCClass 类实例之间构成了保留环

要点：

1. 如果block所捕获的对象直接或者间接的保留了block本身，那么就要小心循环引用问题；
2. 一定要找个适当的时机解除掉保留环，而不能把责任推给API的调用者。

第四十一条：多用派发队列，少用同步锁

线程同步问题在iOS中可以用三种方法解决：

方法一：内置的同步block

```
- (void)synchronizedMethod{
        @synchronized(self) {
             //execuable code.
        }
}
```

这种写法同步的对象是self，会给每个对象自动创建一个锁，在代码执行的时候lock，执行完之后释放unlock。然而，@synchronized(self)会降低代码的运行性能。

方法二：直接使用NSLock对象或者递归锁NSRecursive

```
_lock = [[NSLock alloc] init];
- (void)synchronizedMethod{
            [ _lock lock];
             //execuable code
            [_lock unlock];
    }
```

这样写的问题是在极端情况下回导致死锁，其效率也不见得高。

方法三：GCD

GCD可以高效的同步代码，比如属性需要同步的地方，用atomic修饰成原子性。然而这样做在某种程度上可以实现同步，但是却不能实现真正的同步，比如在同一个线程上多次调用getter，每次得到的属性未必相同。另一种简单有效的方法便是使用GCD提供的串行同步队列（serial synchronization quene），将setter和getter写在同一个队列中。

```
_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;
    });
}
```

这种写法中的全部加锁都是GCD实现的，它是在相当深的底层实现的，于是能够得到许多优化。然而还可以得到进一步优化。由于setter并不是要返回什么，所以可以异步执行，可以提高执行效率。

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

多个getter方法可以并发执行，但是setter和getter不能并发执行。利用这个特点可以写出更快的代码

```
_syncQuene = dispatch_get_global_quene(DISPATCH_QUENE_PRIORITY_DEFAULT,0);
-(NSString *)someString{
        __block NSString* localString ;
        dispatch_sync(_syncQuene,^{  localString = _someString});
        return localString;
}
-(void)setSomeString:(NSString*)someString{
    //必须单独执行，不可和其他并发block一起执行
    dispatch_barrier_sync(_syncQuene,^{ _someString = someString});
}
```

这里写入操作是用**dispatch_barrier_sync**实现的，**它表示后面的代码只能单独执行，不能与其他block**并行。这对并发队列是有效的，因为串行队列总是按照某种顺序执行的。在并发队列中，要执行的代码必须在所有并发block执行完毕之后才能执行这个barrier block，之后可以按正常的顺序执行。

图 6-3　在这个并发队列中，读取操作是用普通的块来实现的，而写入操作则是用栅栏块来实现的。读取操作可以并行，但写入操作必须单独执行，因为它是栅栏块

要点：

1. 派发队列可以用来表述同步语义，这种做法要比用@synchronized和NSLock更加容易简单；
2. 将同步与异步派发结合起来可以实现和普通锁一样的同步行为，而这么做也不会阻塞执行异步派发的线程；
3. 使用同步队列和barrier block，可以令同步行为更高效。

第四十二条：多用GCD少用performSelector

要点：

1. performSelector系列方法容易在内存泄露方面疏忽，他无法确定要执行的具体方法是什么，因为ARC编译器无法插入适当的内存管理方法；
2. performSelector系列方法所处理selector太局限了，返回值和参数都有限制；
3. 如果想把任务放在另一个线程上执行，最好不要使用performSelector方法，而应该把任务封装在block里，然后调用GCD执行。

第四十三条：掌握GCD和操作队列的时机

与GCD技术相关的一种技术为NSOperationQuene，它在GCD之前就有了。但是它们的实现方式却很不同。

GCD是用纯C函数实现的，而NSOperationQuene是用Objective-C实现的，GCD中任务使用block实现的，而block是轻量级的数据结构；NSOperationQuene则是重量级的Objective-C对象。

NSOperationQuene使用addOperationWithBlock:方法搭配NSOperationQuene来操作队列，语法与GCD甚像，其好处如下：

1. 可以很容易的取消某个操作，而GCD比较麻烦；
2. 指定操作间的依赖关系，使特定的在完成某个操作之后在执行等；
3. 通过KVO观测NSOperation的属性，比如isCancelled，isFinished；
4. 指定操作的优先级。
5. 重用NSOperation对象

要点：

1. 在解决多线程与任务管理问题时，派发队列并非唯一方案；
2. 操作队列提供了一套高层的Objective-C API，能实现GCD所具备的绝大部分功能，而且还能完成一些更为复杂的操作，那些操作若改用GCD来实现，则需要另外编写代码。

第四十四条：通过Dispatch Group机制，根据系统资源状况来执行任务

```
dispatch_group 可以将要执行的操作分组，在此组任务执行完毕后得到通知。
dispatch_group_async(<#dispatch_group_t _Nonnull group#>,
                                          <#dispatch_queue_t _Nonnull queue#>,
                                          <#^(void)block#>)
dispatch_group_enter(<#dispatch_group_t _Nonnull group#>)//使任务递增
dispatch_group_leave(<#dispatch_group_t _Nonnull group#>)//使任务递减
dispatch_group_wait(<#dispatch_group_t _Nonnull group#>, <#dispatch_time_t timeou
dispatch_group_notify(<#dispatch_group_t _Nonnull group#>,
                                          <#dispatch_queue_t _Nonnull queue#>,
                                          <#^(void)block#>)
```

要点：

1. 一系列任务可归入一个dispatch group之中，开发者可以在这组任务执行完毕时获得通知；

2. 通过dispatch group，可以在并发式派发队列里同时执行多任务，此时GCD会根据系统资源状况来调度这些并发执行任务。若开发者自己实现此功能，则需要编写大量代码。

第四十五条：使用dispatch_once来执行一次性代码

第四十六条：不要使用dispatch_get_current_quene

要点：

1. dispatch_get_current_quene函数的行为常常和开发者预期的不同。此函数已经废弃，只应该做调试用；
2. 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”这个概念。