iOS 5 Programming Cookbook

第五章
并发

版本 1.0

翻译时间：2012-06-11

DevDiv 翻译： kyelup cloudhsu 耐心摩卡
wangli2003j3 xiebaochun dymx101
DevDiv 校对： laigb kyelup
DevDiv 编辑： BeyondVincent

写在前面

目前，移动开发被广大的开发者们看好，并大量的加入移动领域的开发。

鉴于以下原因：

- 国内的相关中文资料缺乏
- 许多开发者对 E 文很是感冒
- 电子版的文档利于技术传播和交流

DevDiv.com 移动开发论坛特此成立了翻译组，翻译组成员具有丰富的移动开发经验和英语翻译水平。组员们利用业余时间，把一些好的相关英文资料翻译成中文，为广大移动开发者尽一点绵薄之力，希望能对读者有些许作用，在此也感谢组员们的辛勤付出。

关于 DevDiv

DevDiv 已成长为国内最具人气的综合性移动开发社区

更多相关信息请访问 DevDiv 移动开发论坛。

技术支持

首先 DevDiv 翻译组对您能够阅读本文以及关注 DevDiv 表示由衷的感谢。

在您学习和开发过程中，或多或少会遇到一些问题。DevDiv 论坛集结了一流的移动专家，我们很乐意与您一起探讨移动开发。如果您有什么问题和技术需要支持的话，请访问 DevDiv 移动开发论坛 或者发送邮件到 BeyondVincent@DevDiv.com，我们将尽力所能及的帮助你。

关于本文的翻译

感谢 kyelup、cloudhsu、耐心摩卡、wangli2003j3、xiebaochun 和 dymx101 对本文的翻译，同时非常感谢 laigb 和 kyelup 在百忙中抽出时间对翻译初稿的认真校验，指出了文章中的错误。才使本文与读者尽快见面。由于书稿内容多，我们的知识有限，尽管我们进行了细心的检查，但是还是会存在错误，这里恳请广大读者批评指正，并发送邮件至 BeyondVincent@devdiv.com，在此我们表示衷心的感谢。

读者查看下面的帖子可以持续关注章节翻译更新情况

[DevDiv 翻译]iOS 5 Programming Cookbook 翻译各章节汇总

目录

前言

参考帖子
[DevDiv翻译]iOS 5 Programming Cookbook翻译_前言

第 1 章 基础入门

参考帖子

[DevDiv 翻译]iOS 5 Programming Cookbook 翻译_第一章_基础入门

第 2 章 使用控制器和视图

参考帖子

[DevDiv 翻译]iOS 5 Programming Cookbook 翻译_第二章_使用控制器和视图(上)

[DevDiv 翻译]iOS 5 Programming Cookbook 翻译_第二章_使用控制器和视图(下)

第 3 章　构造和使用 Table View

参考帖子

[DevDiv 翻译]iOS 5 Programming Cookbook 翻译_第三章_构造和使用 Table View

第 4 章 Storyboards

参考帖子

[DevDiv 翻译]iOS 5 Programming Cookbook 翻译_第四章_Storyboards

第 5 章 并发

5.0. 简介

当两个或两个以上的任务同时执行时就发生了并发。即使只有一个 CPU，现代操作系统也能够在同时执行多个任务。要实现这一点，它们需要给每个任务从 CPU 中分配一定的时间片。例如，要在 1 秒钟内执行 10 个同样优先级的任务，操作系统会用 10（任务）来平均分配 1000 毫秒（每秒钟有 1000 毫秒），那么每个任务就会有 100 毫秒的 CPU 时间。这就意味着所以的任务会在同一秒钟内执行，也就是并发执行。

然而，随着技术进步，现在我们的 CPU 有不止一个内核。这就意味着 CPU 真正具备了同时执行多个任务的能力。操作系统将任务分配到 CPU 并等到任务执行完成。就是这么简单！

Grand Central Dispatch，或者简称 GCD，是一个与 Block Object 产生工作的低级的 C API。GCD 真正的用途是将任务分配到多个核心又不让程序员担心哪个内核执行哪个任务。在 Max OS X 上，多内核设备，包括笔记本，用户已经使用了相当长的时间。通过多核设备比如 iPad2 的介绍，程序员能为 iOS 写出神奇的多核多线程 APP。

GCD 的核心是分派队列。不论在 iOS 还是 Max OS X 分派队列，正如我们快看到的是由位于主操作系统的 GCD 来管理的线程池。你不会直接与线程有工作关系。你只在分派队列上工作，将任务分派到这个队列上并要求队列来调用你的任务。GCD 为运行任务提供了几个选择：同步执行、异步执行和延迟执行等。

要在你的 APP 开始使用 GCD，你没有必要将任何特殊库导入你的项目。Apple 已经在 GCD 中纳入了各种框架，包括 Core Foundation 和 Cocoa/Cocoa Touch。GCD 中的所有方法和数据类型都以 dispatch_关键词开头。例如，dispatch_async 允许你在一个队列上分派任务来异步执行，而 dispatch_after 允许你在一个给定的延迟之后运行一个 block。传统上，程序员必须创建自己的线程来并行执行任务。例如，一个 iOS 开发者会创建一个与下面这个类似的线程就要执行一个操作 1000 次：

```
- (void) doCalculation{
   /* Do your calculation here */
}
- (void) calculationThreadEntry{

   @autoreleasepool {
      NSUInteger counter = 0;
      while ([[NSThread currentThread] isCancelled] == NO){
         [self doCalculation];
         counter++;
         if (counter >= 1000){
            break;
         }
      }
   }

}
- (BOOL)                    application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
```

```
    /* Start the thread */
    [NSThread detachNewThreadSelector:@selector(calculationThreadEntry)
                                toTarget:self
                              withObject:nil];

    self.window = [[UIWindow alloc] initWithFrame:
                   [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

程序员必须开始手动写线程然后为线程创建要求的结构（切入点，autorelease pool 和线程的主循环）。当我们在 GCD 写同样的代码时，就没有必要做这么多事情：

```
dispatch_queue_t queue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
size_t numberOfIterations = 1000;
dispatch_async(queue, ^(void) {
    dispatch_apply(numberOfIterations, queue, ^(size_t iteration){
        /* Perform the operation here */
          });
        });
```

在本章中，你将掌握所有关于 GCD 的内容，了解如何用 GCD 为 iOS 和 Mac OS X 实现神奇性能的多内核设备例如 iPad2 写出现代多线程 APP。我们会一直和分派队列打交道，所以请确保你完全理解了其背后的概念。有 3 种分派队列：

Main Queue

这个队列在主线程上执行它的所有任务，Cocoa 和 Cocoa Touch 允许程序员在主线程上调用一切 UI-related 方法。使用 dispatch_get_main_queue 函数检索到主队列的句柄。

Concurrent Queues

为了执行异步和同步任务，你可以在 GCD 中检索到这写队列。多个并发队列能够轻而易举的并行执行多个任务，没有更多的线程管理，酷！使用 dispatch_get_global_queue 函数检索一个并发队列的句柄。

Serial Queues

无论你提交同步或者异步任务，这些队列总是按照先入先出（FIFO）的原则来执行任务，这就意味着它们一次执行一个 Block Object。然而，他们不在主线程上运行，所以对于那些要按照严格顺序执行并不阻塞主线程的任务而言是一个完美的选择。使用 dispatch_queue_create 函数创建一个串行队列。一旦你使用完整队列，必须使用 dispatch_release 函数释放它。

在 APP 生命周期内的任何时刻，你可以同时使用多个分派队列。你的系统只有一个主队列，但是你可以创建多个串行队列来实现任何你需要 APP 实现的功能，当然是在合理的范围内。你也可以检索多个并发队列并将任务分派给它们，任务可以通过 2 种方式传递分派队列：Block Objects 和 C 函数，这些我们会在 5.4 中详细讲解。

Block Object 是通常在 Objective-C 中以方法形式出现的代码包。Block Objects 和 GCD 共同创建了一个和谐环境，在这个环境里你能在 iOS 和 Mac OS X 上发布高性能的多线程 APPs。Block Objects 和 GCD 有什么特别的地方呢？你可能会问。很简单：没有太多的线程！所有你要做的事情就是把代码放进 Block Objects 然后要求 GCD 来为你小心执行代码。Objective-C 中的 Block Object 是编程领域所调用的一类的对象。这意味着你可以动态生成的代码、把 Block Object 作为参数传递给方法，并从一种方法返回一个 Block Object。所有这些事情能让你更容易地选择你想在运行库做什么，并且更容易改变一个程序的活动。特别

的，Block Objects 能够通过 GCD 在单个线程上运行。作为 Objective-C 对象，Block Objects 可以看成和其他任何对象一样。

Block Objects 有时被称作闭包。构建 Block Objects 和构建传统的 C 函数类似，可以在 5.1 了解到。.Block Objects 可以有返回值，可以接受参数。BlockObjects 可以内敛定义，或者当做一个独立的代码块来看待，这一点与 C 函数相同。当内联创建时与 Block Object 作为一个独立代码块来执行时相比，能够访问 Block Object 的变量范围完全不同。

GCD 与 Block Objects 一起工作。当使用 GCD 执行任务时，你可以传递一个 Block Object，它可以同步或者异步执行，取决于你在 GCD 中使用的方法。然而，你可以创建一个 Block Object 负责下载一个作为参数传递到它的 URL。单独的 Block Object 可以同步或者异步应用于 APP 的任何地方，这个位置依赖于你想如何运行它。你不必使 Block Object 本身同步或异步，你只要简单通过同步或者异步的 GCD 方法来调用它，它就会工作。

对写 iOS 和 OS X APPs 的程序员而言 Block Objects 是非常新的概念。然而事实上，Block Objects 不像线程那么受欢迎，可能是因为它们的语法和纯粹的 Objective-C 方法不同，并且要复杂很多。不过，Block Objects 功能异常强大，Apple 正在大力推动来把它纳入 Apple 库。在类中你可能已经看到这些添加例如 NSMutableArray，在类中程序员可以使用一个 Block Object 选择数组。

本章完全来讲解在 iOS 和 Mac OS X APPs 上建构和使用 Block Objects，使用 GCD 向操作系统、线程和计时器分派任务。我想强调，掌握 Block Object 语法的唯一方法就是自己去写一个 Block Object；看看本章的样板代码，然后试着执行自己的 Block Objects。

此处你了解一些 Block Object 的基础知识，接下来会一些更高级的主题比如 GCD、线程、计时器、操作和操作队列。在讲解 GCD 内容之前你会理解你需要了解的关于 Block Objects 的一切事情。根据我的经验，掌握 Block Objects 的最好途径就是通过例子来学习，所以本章中你会看到大量例子。确保你试图通过 Xcode 中的例子而真正理解 Block Objects 的语法。

操作可以同步或者异步执行代码块。你可以手动管理操作或者是把它们放到操作队列中，这个队列会促使并发以致于你不需要思考底层线程管理。本章中，你会了解在 APP 上同步或者异步执行任务时如何使用操作和操作队列，如何使用基础线程和计时器。Cocoa 提供了 3 种不同的操作：

Block Operations

它们促使执行一个或多个 Block Objects.

Invocation Operations

这些允许你在另一个当前存在的对象中调用一个方法。

Plain Operations

这些都是需要被继承普通操作的类。将要执行的代码会被写入操作对象的主要方法中。

和前面提到的一样，Operations 由拥有 NSOperationQueue 数据类型的操作队列管理。再将后面提到的操作类型（块、调用或普通操作）实例化之后，你可以把它们增加到一个操作队列中，并用这个队列来管理操作。

一个操作对象可以和其他操作对象有依存项，在执行与其相关任务之前必须等待一个或者多个操作完成。除非你增加了依存项，否则你无法控制操作运行的顺序。例如，按照一定顺序把它们增加到一个队列上不能保证它们按照这个顺序执行，尽管使用了队列选项。

在用操作队列和操作的过程中，有几个要点要记住：

- 默认情况下，操作使用启动实例方法在启动它们的线程上运行。如果你想让操作异步

工作，必须使用一个操作队列和一个类 NSOperation，在运行的主要实例方法上分离一个新的线程。

- 一个操作在开始它自己的运行之前可以等待另一个操作执行结束。但是要小心不要创建一个相互依存操作，一个常见的错误就是争用条件。换句话说，如果 B 已经依赖了 A,不要让操作 A 依赖操作 B。这将会导致双方永远等待下去，会占用内存甚至可能挂你的应用程序。
- 操作可以取消。因此，如果你由 NSOperation 子类来创建自定义操作对象，你必须确保使用 isCancelled 方法去检查这个操作在执行与其相关的任务之前是否已被取消掉。例如，如果操作的任务是每 20 秒检查一次网络连接的可用性，必须在每个运行开始就调用 isCancelled 实例方法来保证在尝试再次检查网络连接之前它已经被取消掉。如果操作多花了几秒钟（比如当你下载文件的时候），在运行任务的时候你应该定期检查 isCancelled。
- 操作对象的 key-value observing (KVO)符合各种关键路径（如 isFinished 的 IsReady，isExecuting）。我们将在后面的章节中讨论 Key Value Coding 和 Key Value Observing。
- 如果您计划将 NSOperation 子类化并提供自定义实现，必须在操作的主要方法中创建您自己的 autorelease 池，这个 autourelease 池从 start 方法获取调用。我们将在本章后面的内容中详细讨论这个问题。
- 始终保持对创建的操作对象的引用。操作队列的并发性，使得它不可能在操作被添加到操作队列后还会检索引用。

线程和定时器是继承 NSObject 的对象。线程比计时器要低一个级别。当 APP 在 iOS 上运行时，操作系统为 APP 创建了至少一个线程，称作主线程，每个线程必须添加到一个运行循环中。运行循环，顾名思义，是一个循环过程中的不同事件可以发生，如触发计时器或运行线程。关于运行循环的讨论超出了本章的范围，但是我们在此处仍或多少提及。

将运行循环想象成一种有一个开始点、一个完成条件和一系列在其生命周期内发生的事件的循环。一个线程或者计时器与一个运行附加到运行循环，事实上需要运行循环激活其功能。

一个 APP 的主线程是处理 UI 事件的线程。如果你在主线程执行一个长时间运行的任务，就要注意 APP 的 UI 会没有响应或者响应缓慢。为了避免这一点，你可以创建一个独立线程和/或计时器，它们会分别执行各自的任务（即使是一个长时间运行的任务）同时又不会阻塞主线程。

5.1. 构建 Block Objects

5.1.1. 问题

您希望能够编写您自己的 Block Objects 或与 iOS SDK 类使用 Block Objects。

5.1.2. 方案

你只需要理解 Block Objects 和经典 C 函数之间的区别，这些区别将在讨论部分解释。

5.1.3. 讨论

Block Objects 的对象可以内联或编码为独立的代码块。我们从后一种开始。假设你在 Objective-C 中有一个方法，既能接受 NSInteger 类的 2 个整数值，又能通过两个相减返回两者的差值作为 NSInteger：

```
- (NSInteger) subtract:(NSInteger)paramValue
```

```
                    from:(NSInteger)paramFrom{
    return paramFrom - paramValue;
}
```

这个很简单，不是吗？现在我们把 Objective-C 代码翻译成纯粹 C 函数，这样我们就有了同样的功能来进一步掌握 Block Objects 语法：

```
NSInteger subtract(NSInteger paramValue, NSInteger paramFrom){
    return paramFrom - paramValue;
}
```

你可以看到 C 函数在语法上和 Objective-C 非常不同，现在我们看看作为 Block Object 我们如何编码同样的功能：

```
NSInteger (^subtract)(NSInteger, NSInteger) =
    ^(NSInteger paramValue, NSInteger paramFrom){
    return paramFrom - paramValue;
};
```

在我详细讲解 Block Objects 语法之前，我会给你几个例子。假设我们在 C 中有个函数，它携带类 NSUInteger 参数（一个无符号整数），并且返回时作为 NSString 类字符串。此处我们在 C 中如何执行：

```
NSString* intToString (NSUInteger paramInteger){
    return [NSString stringWithFormat:@"%lu",
                (unsigned long)paramInteger];
}
```

要了解 Objective-C 中独立于系统的格式说明符的格式字符串，请参考 Apple 网站的 iOS 开发者库中的字符串编码指南。

这个 C 函数的等效 Block Object 会在例 5-1 中会看到。

例 5-1.示例 Block Object 作为函数来定义

```
NSString* (^intToString)(NSUInteger) = ^(NSUInteger paramInteger){
    NSString *result = [NSString stringWithFormat:@"%lu",
                                (unsigned long)paramInteger];
    return result;
};
```

独立 Block Object 最简单的形式就是这个 Block Object 返回 void，而且不带任何参数：

```
void (^simpleBlock)(void) = ^{
    /* Implement the block object here */
};
```

Block Objets 可以以完全相同的方式调用 C 函数，如果他们有任何参数，你可以像 C 函数那样传递参数；任何返回值可以精确检索，就像检索 C 函数的返回值一样。这儿有个例子：

```
NSString* (^intToString)(NSUInteger) = ^(NSUInteger paramInteger){
   NSString *result = [NSString stringWithFormat:@"%lu",
                              (unsigned long)paramInteger];
   return result;
};
- (void) callIntToString{
   NSString *string = intToString(10);
   NSLog(@"string = %@", string);
}
```

通过输入值 10 作为唯一函数这个 Block Object 并且把这个 block object 的返回值放入字符串的本地变量，CallIntToString Objective-C 方法可以调用 intToString Block Object。现在，我们知道如何把 Block Objects 作为独立的代码块来写，我们看下把 Block Objects 作为参数传递到 Objective-C 的方法。我们将不得不抽象思考下以下示例的目的。假设我们有一个 Objective-C 方法，它接受整数并且能本身发生一些转换，这些转换主要取决于我们程序中发生的事情。我们知道我们已经有个一个整数输入和一个字符串输出，但是精确转换由 Block Object 决定，Block Object 在我们每次运行方法时都会不同。所以这个方法接受被转换的整数和将要转换整数的 Block 作为它的参数。对我们的 Block Object 而言，我们将使用相同的 intToString Block Object，它在前文的例 5-1 中已经执行过。现在我们需要一个 Objective-C 方法，这个方法接受一个无符号的整数参数和一个 Block Object 作为参数。无符号的整数参数很简单，但是我们如何才能调用这个必须接受一个和 intToString Block Object 相同类型的 Block Object 方法。首先我们 **typedef** 这个 **intToString**Block Object 的签名，这个签名会告诉编译器我们的 Block Object 会接受什么参数：

```
typedef NSString* (^IntToStringConverter)(NSUInteger paramInteger);
```

这个 **typedef** 告诉编译器 Block Objects 接受一个整数参数并且返回一个被 IntToString Converter 命名的标示符来展现的字符串。现在我们继续来写一个 Objective-C 方法，它接受一个整数和 IntToStringConverter 类型的 Block Object:

```
- (NSString *) convertIntToString:(NSUInteger)paramInteger
                    usingBlockObject:(IntToStringConverter)paramBlockObject{
   return paramBlockObject(paramInteger);
}
```

现在我们必须通过已选择的 Block Object 调用 **convertIntToString:**方法(例 5-2)。

例 5-2.在另一个方法中调用 Block Object

```
- (void) doTheConversion{
   NSString *result = [self convertIntToString:123
                                  usingBlockObject:intToString];
   NSLog(@"result = %@", result);
}
```

现在我们已经了解了关于独立 Block Objects 的一些内容，让我们转入内联 Block Objects。在 doTheConversion 方法中我们看到，我们把 Into String Block Objects 当作参数传递到 convertIntToString:usingBlockObject:方法。如果我们没有现成的 Block Objects 传递到这个方法会怎么样呢？好，问题来了。前面提到过，Block Objects 是第一类函数并且可以在运行库中构建。我们来看看一个 doTheConversion 方法的可选实现（例 5-3）。

例 5-3. Block Object 定义为函数的举例

```
- (void) doTheConversion{
   IntToStringConverter inlineConverter = ^(NSUInteger paramInteger){
      NSString *result = [NSString stringWithFormat:@"%lu",
                                 (unsigned long)paramInteger];
      return result;
   };
   NSString *result = [self convertIntToString:123
                                    usingBlockObject:inlineConverter];
   NSLog(@"result = %@", result);
}
```

把例 5-3 和前面的例 5-1 比较下，我已经移动了 Block Object 签名的初始代码，其中签名由一个名称和参数(^intToString)(NSUInteger)组成；让所有其他 BlockObjects 保持不变，现在会有一个匿名 Object。但这并不是说我没有办法引用这个 Block Object。我将使用等号赋给它类型和名称：IntToStringConverter 内联转换器。现在我使用数据类型强制执行方法的正确使用，并且使用名称实际传输给 Block Object。另外在构建刚刚所示的内联 Block Objects 时，我们可以在将它作为一个参数传递的同时构建 Block Objects：

```
- (void) doTheConversion{
   NSString *result =
   [self convertIntToString:123
            usingBlockObject:^NSString *(NSUInteger paramInteger) {
      NSString *result = [NSString stringWithFormat:@"%lu",
                                 (unsigned long)paramInteger];
      return result;
   }];
   NSLog(@"result = %@", result);
}
```

将这个例子和例 5-2 作比较，2 个方法都通过 usingBlockObject 语法使用 Block Object；但是较早版本提及以前声明的 Block Objects 的名称（intToString），这样可以简单快速的创建一个 Block Object。在这段代码中，我们构建了一个内联 Block Object 作为第二个参数传输到 convertIntToString:usingBlockObject: 方法中。

5.1.4. 参考

XXX

5.2. 在 Block Object 中获取变量

5.2.1. 问题

你想要理解在 Objective-C 方法和在 Block Objects 中获取变量的区别。

5.2.2. 方案

这儿简单总结关于 Block Objects 变量你必须了解的特点：

- 局部变量在 Block Objects 和 Objective-C 方法中的工作原理非常相似。
- 对于内联 Block Objects,局部变脸不仅包含 Block 内部定义的变量，并且包含在 Block Objects 执行方法中定义的变量。(随后会有举例)
- 你不能参考 self；在 Objective-C 类中运行的独立 Block Objects，如果你需要访问 **self**，就必须把 Object 作为参数传递到 BlockObject，我们很快会看到举例。
- 只有当 self 出现在创建 Block Object 的词法范围内，你可以在内联 Block Object 内参

考 **self**。

- 对于内联 Block Objects，那些在 BlockObject 执行过程中定义的局部变量是可读写的，换句话说，对于 Block Objects 自身的局部变量来说，Block Objects 有个读写存取。
- 对于内联 Block Objects，实现 Object 的 Objective-C 方法的局部变量只能从中读取，不能写入。不过还有一个例外，如果定义它们通过 __block 存储类型定义的话，Block Object 可以写入此类的变量。对此我们也会有举例。
- •假设你有一个类 NSObject 的 Object，并且在这个 Object 的执行中你使用了一个 Block Object 与 GCD 相连，那么在这个 Block Object 内部，你会有一个存储来读取那个支持你的 Block 执行的 NSObject 内部的声明属性。

只有当你使用声明属性的 setter and getter 方法你才能获取独立 Block Objects 内部的 NSObject 的这些属性；在一个独立 Block Object 使用 Dot Notation 方法你无法获取一个 Object 的声明属性。

5.2.3. 讨论

我们先来看看如何在两个 Block Objects 的执行过程中使用变量；一个是内联 Block Object，另一个是独立 Block Objects：

```
void (^independentBlockObject)(void) = ^(void){
   NSInteger localInteger = 10;
   NSLog(@"local integer = %ld", (long)localInteger);
   localInteger = 20;
   NSLog(@"local integer = %ld", (long)localInteger);
};
```

我们赋予的值调用这个 Block Object 时，我们的赋值会打印到控制台窗口：

```
local integer = 10
local integer = 20
```

目前为止，一切都好。现在让我们看看内联 Block Objects 和它们的局部变量：

```
- (void) simpleMethod{
   NSUInteger outsideVariable = 10;
   NSMutableArray *array = [[NSMutableArray alloc]
                            initWithObjects:@"obj1",
                            @"obj2", nil];
   [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
      NSUInteger insideVariable = 20;
      NSLog(@"Outside variable = %lu", (unsigned long)outsideVariable);
      NSLog(@"Inside variable = %lu", (unsigned long)insideVariable);
      /* Return value for our block object */
      return NSOrderedSame;
   }];
}
```

NSMutableArray 的 sortUsingComparator:instance 方法试图寻找一个可变的数组，这个示例代码的目的只是为了说明局部变量的使用，所以你不需要知道这个方法的内部

原理。

Block Object 可以读写它自己的 insideVariable 局部变量。然而，对于默认的 outsideVariable 变量，Block Object 有一个只读接口。为了允许 Block Object 写入 outsideVariable，我们必须给 outside Variable 增加一个__block 存储类的前缀：

```
- (void) simpleMethod{
   __block NSUInteger outsideVariable = 10;
   NSMutableArray *array = [[NSMutableArray alloc]
                                    initWithObjects:@"obj1",
                                    @"obj2", nil];
   [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
       NSUInteger insideVariable = 20;
       outsideVariable = 30;
       NSLog(@"Outside variable = %lu", (unsigned long)outsideVariable);
       NSLog(@"Inside variable = %lu", (unsigned long)insideVariable);
       /* Return value for our block object */
       return NSOrderedSame;
    }];
}
```

在内联 Block Objects 获取 Self 和那些创建内联 Block Object 的词法范围内定义的 Self 完全一样。例如，在这个例子中，Block Object 将会获取 Self，因为 simpleMethod 是一个 Objective-C 类的实例：

```
- (void) simpleMethod{
   NSMutableArray *array = [[NSMutableArray alloc]
                                    initWithObjects:@"obj1",
                                    @"obj2", nil];
   [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
      NSLog(@"self = %@", self);
      /* Return value for our block object */
      return NSOrderedSame;
   }];
}
```

在 Block Object 的执行的没有变化的情况下，你不能从独立 Block Object 中获取 Self。试图编译这段代码时你将会收到一个编译时错误:

```
void (^incorrectBlockObject)(void) = ^{
   NSLog(@"self = %@", self); /* self is undefined here */
};
```

如果你想从独立 Block Objects 中获取 Self，只要把这个 Object 传递到你的 Block Object，其中 Self 是 Object 的一个参数：

```
void (^correctBlockObject)(id) = ^(id self){
   NSLog(@"self = %@", self);
};
- (void) callCorrectBlockObject{
   correctBlockObject(self);
}
```

你不需要把这个参数命名为 Self，可以将这个参数叫做其他任何东西。然而，如果你把这个参数称作 Self 的话，你随后会容易抓住 Block Object 的代码并把它放到一个 Objective-C 方法的执行过程中，在这个执行中为了使编译器能够理解，你不需要把变量名称的每个实例改成 Self。

我们来看看已声明的属性，看看 Block Object 如何获取它们。对于内联 Block Objects，你可以使用 dot notation 读写 Self 的已声明属性。例如，假设在我们类中有一个 NSString 的类已声明属性叫做 stringProperty：

```
#import <UIKit/UIKit.h>
@interface GCDAppDelegate : NSObject <UIApplicationDelegate>
@property (nonatomic, strong) NSString        *stringProperty;
@end
```

现在我们能够这样在内联 Block Object 中获取这个属性：

```
#import "GCDAppDelegate.h"
@implementation GCDAppDelegate
@synthesize stringProperty;
- (void) simpleMethod{
   NSMutableArray *array = [[NSMutableArray alloc]
                                    initWithObjects:@"obj1",
                                    @"obj2", nil];
   [array sortUsingComparator:^NSComparisonResult(id obj1, id obj2) {
      NSLog(@"self = %@", self);
      self.stringProperty = @"Block Objects";
      NSLog(@"String property = %@", self.stringProperty);
      /* Return value for our block object */
      return NSOrderedSame;
   }];
}
@end
```

然而在独立 Block Object 内部，你不能使用 dot notation 读写一个已声明属性：

```
void (^correctBlockObject)(id) = ^(id self){
   NSLog(@"self = %@", self);
   /* Should use setter method instead of this */
   self.stringProperty = @"Block Objects"; /* Compile-time Error */
   /* Should use getter method instead of this */
   NSLog(@"self.stringProperty = %@",
            self.stringProperty); /* Compile-time Error */
};
```

在这个场景中可以使用这个合成属性的 getter and setter 方法来代替 dot notation：

```
void (^correctBlockObject)(id) = ^(id self){
   NSLog(@"self = %@", self);
   /* This will work fine */
   [self setStringProperty:@"Block Objects"];
   /* This will work fine as well */
   NSLog(@"self.stringProperty = %@",
            [self stringProperty]);
```

```
};
```

当它出现在内联 Block Objects 中，有一条非常重要的规则你必须记住：内联 Block Objects 在其词法区域会为这些变量复制值。如果不明白这一点，不要担心，我们看个例子：

```
typedef void (^BlockWithNoParams)(void);
- (void) scopeTest{
   NSUInteger integerValue = 10;
   /*************** Definition of internal block object ***************/
   BlockWithNoParams myBlock = ^{
      NSLog(@"Integer value inside the block = %lu",
               (unsigned long)integerValue);
   };
   /*************** End definition of internal block object ***************/
   integerValue = 20;
   /* Call the block here after changing the
     value of the integerValue variable */
   myBlock();
   NSLog(@"Integer value outside the block = %lu",
            (unsigned long)integerValue);
}
```

我们声明一个局部整型变量并且将其原始值赋值为 10，然后我们执行 Block Object 但是没有调用 Block Object。在 Block Object 执行之后，我们简单改变局部整型变量的值，随后在我们调用时 Block Object 将试着读取这些值；随后把局部变量的值改为 20，我们再调用 Block Object，你期待 Block Object 将变量的值打印为 20，但是这个结果不会出现；它将打印为 10，和你在此处看到的一样：

```
Integer value inside the block = 10
Integer value outside the block = 20
```

此处发生的事情是在执行 Block 的地方 Block Object 自身一直有一个 integerValue 变量的只读复制。你可能想，为什么 Block Object 抓取了一个局部变量 integerValue 的只读值？答案很简单，我们已经在这个部分解答了这个问题。除非用存储类__block 前缀限定，Block Object 词法区域的局部变量传作为只读变量传递给 Block Object。所以，为了改变这个现象，我们可以把 scopeTest method 的执行改成带有__block 存储类做前缀的 integerValue 变量，就像这样：

```
- (void) scopeTest{
   __block NSUInteger integerValue = 10;
   /*************** Definition of internal block object ***************/
   BlockWithNoParams myBlock = ^{
      NSLog(@"Integer value inside the block = %lu",
               (unsigned long)integerValue);
   };
   /*************** End definition of internal block object ***************/
   integerValue = 20;
   /* Call the block here after changing the
     value of the integerValue variable */
   myBlock();
   NSLog(@"Integer value outside the block = %lu",
            (unsigned long)integerValue);
```

```
}
```

在调用了 scopeTest 方法之后，现在如果我们在控制台空口查看结果的话，我们会看到下面的结果:

```
Integer value inside the block = 20
Integer value outside the block = 20
```

关于在 Block Objects 上使用变量的问题，这个部分给了你足够的信息。我建议你写一些 Block Objects 并在其内部使用变量；给这些变量赋值并且读取，会让你更好的理解 Block Objects 如果使用变量。如果你忘了在 Block Objects 控制变量访问的规则就回过头在看看这个部分。

5.2.4. 参考

XXX

5.3. 调用 Block Object

5.3.1. 问题

已经了解如何构建 Block Objects，现在你想执行 Block Objects 来获取结果。

5.3.2. 方案

执行 Block Objects 和执行 C 函数的方法一样，这一点将在讨论部分展开。

5.3.3. 讨论

我们在 5.1 和 5.2 中看到过调用 Block Objects 的例子，在这一部分将会呈现更多具体的实例。

如果你有一个独立 Block Object，你可以想调用 C 函数那样简单调用它：

```
void (^simpleBlock)(NSString *) = ^(NSString    *paramString){
   /* Implement the block object here and use the
     paramString parameter */
};
- (void) callSimpleBlock{
   simpleBlock(@"O'Reilly");
}
```

如果你想在另一个独立 BlockObject 内部调用一个独立 Block Object，按照同样的方法就像调用一个 C 方法那样来调用一个新的 Block Object：

```
/*************** Definition of first block object ***************/
NSString *(^trimString)(NSString *) = ^(NSString *inputString){
   NSString *result = [inputString stringByTrimmingCharactersInSet:
                                 [NSCharacterSet whitespaceCharacterSet]];
   return result;
};
/*************** End definition of first block object ***************/
/*************** Definition of second block object ***************/
NSString *(^trimWithOtherBlock)(NSString *) = ^(NSString *inputString){
   return trimString(inputString);
};
```

```
/*************** End definition of second block object ***************/
- (void) callTrimBlock{
   NSString *trimmedString = trimWithOtherBlock(@"     O'Reilly      ");
   NSLog(@"Trimmed string = %@", trimmedString);
}
In this example, go ahead and invoke the callTrimBlock Objective-C method:
[self callTrimBlock];
```

callTrimBlock 方法会调用 trimWithOtherBlock Block Object，trimWithOtherBlock Block Object 为了调整给定的字符串会调用 trimString Block Object。调整字符串很简单，可以在一行代码中完成，但是这个举例代码告诉了你如何在 Block Objects 内部调用 Block Objects。

5.3.4. 参考

XXX

5.4. 给 Grand Central Dispatch 分派任务

5.4.1. 问题

5.4.2. 方案

dispatch queues:有两个向调度队列提交任务的方法：

- Block Objects (见 5.1)
- C 函数

5.4.3. 讨论

Block Objects 是利用 GCD 及其巨大能量的最好途径。一些 GCD 函数已经扩展到允许程序员使用 Block Objects 的 C 函数。然而，事实是只有一部分受限制的 GCD 函数允许程序员使用 C 函数，所以在采取进一步行动之前要阅读下关于 Block Objects（5.1）的内容。那些必须提供给各种 GCD 函数的 C 函数应该属于类 dispatch_function_t，它是在下面这个 Apple 库里边定义的：

*typedef void (*dispatch_function_t)(void *);*

例如，如果我们要创建一个命名为 myGCDFunction 的函数，我们必须按照这样的方法执行它：

```
void myGCDFunction(void * paraContext){
   /* Do the work here */
}
```

当他们把任务分配到这些函数时 paraContext 的参数提到了允许程序员传递他们的 C 函数的语境。我们很快就会学习到这一点。

传递到 GCD 的 Block Objects 不会总是有相同的结构。一些必须接受参数，一些没有必要，但是没有一个 Block Objects 向 GCD 提交返回值。

在接下来的 3 节中，你会掌握如何向 GCD 提交要执行的任务，无论它们是 Block Objects 形式或者是 C 函数。

5.4.4. 参考

XXX

5.5. 用 GCD 执行 UI-Related 任务

5.5.1. 问题

为了并发你使用了 GCD 并且想知道使用 UI-related Api 的最佳方法是什么。

5.5.2. 方案

使用 dispatch_get_main_queue 函数。

5.5.3. 讨论

UI-related 任务必须在主线程中执行，所以主队列是 GCD 中执行 UI 任务的唯一候选对象。我们可以使用 dispatch_get_main_queue 函数得到处理主分派队列的句柄。

有 2 种向主队列分派任务的方法，两者都是异步的，即使在任务没有执行的时候也让你的程序继续：

dispatch_async function

在分派队列上执行一个 Block Object。

dispatch_async_f function

在分派队列上执行一个 C 函数。

Dispatch_sync 方法不能在主队列中调用，因为无限期的阻止线程并会导致你的应用死锁。所有通过 GCD 提交到主队列的任务必须异步提交。

我们看看 dispatch_async 函数的使用情况，它接受 2 个参数：

Dispatch queue handle

在这分派队列上任务将被执行；

Block object

为了异步执行 Block Object 会被发送到分派队列。

这儿有一个例子，在 iOS 中这段代码将使用主队列作为一个提醒来展示给用户：

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();
dispatch_async(mainQueue, ^(void) {

   [[[UIAlertView alloc] initWithTitle:@"GCD"
                                 message:@"GCD is amazing!"
                                delegate:nil
                       cancelButtonTitle:@"OK"
                       otherButtonTitles:nil, nil] show];

});
```

你已经注意到了，Dispatch_async GCD 函数没有参数和返回值。提交到这个函数的 Block Object 为了完成任务就必须收集它自己的数据。在我们刚看到的代码片段中，这个提醒视图有它完成任务时需要的所有值。然而，不可能总是这样。在这样的实例中你必须确保提交到 GCD 的 Block Object 需要的所有

值，在其范围内具有访问权限。

在 iOS 模拟器上运行这个 APP，用户将会获得与图 5-1 中所示类似的结果：

图 5-1. 一个使用异步 GCD 调用展示的提醒

这可能不那么令人印象深刻。事实上，如果你仔细想想确实一般。所以是什么使这个主队列真正有意思呢？答案很简单：当你需要从 GCD 获取最高性能来在并行或者串行线程上完成一些密集运算时，你可能想给你的用户展示结果或者想把一个组件移动到屏幕上；为了做到这些，你必须使用主队列，因为它是 UI-related 的工作。在使用 GCD 更新 UI 时，在这个部分展示的函数是离开一个串行或者并行队列的唯一办法，所以你能想到它有多重要了。

在向主队列提交 Block Object 的执行话，你可以提交 C 函数来代替。在 GCD 中向 dispatch_async_f 函数提交所有 UI-related C 函数的执行。用 C 函数来代替 Blokc Objects 时，只要对代码稍作调整，我们会得到和图 5-1 一样的结果。前面提到过，用 dispatch_async_f function 函数我们可以提交一个 application-defined 的上下文的指针，这个指针随后在调用 C 函数时用到。所以我们的计划是：我们先创建一个结构来保存值，这些值包括一个警报视图的标题、消息和取消键的标题。当 APP 运行时，我们把这些值放入结构并传递到 C 函数去展示。此处是我们如何定义我们的结构：

```
typedef struct{
   char *title;
   char *message;
   char *cancelButtonTitle;
} AlertViewData;
```

现在我们去执行稍后会用 GCD 呼叫的 C 函数。这个 C 函数应该期待类 void*的参数，然后转换到 AlertViewData*。换句话说，我们希望这个函数的调用者传递给我们一个 alert view 的数据的引用，这个数据被封装在 AlertViewData 结构里：

```
void displayAlertView(void *paramContext){

   AlertViewData *alertData = (AlertViewData *)paramContext;

   NSString *title =
      [NSString stringWithUTF8String:alertData->title];

   NSString *message =
      [NSString stringWithUTF8String:alertData->message];

   NSString *cancelButtonTitle =
      [NSString stringWithUTF8String:alertData->cancelButtonTitle];

   [[[UIAlertView alloc] initWithTitle:title
                                 message:message
                                delegate:nil
                       cancelButtonTitle:cancelButtonTitle
                       otherButtonTitles:nil, nil] show];

   free(alertData);

}
```

我们在此处而不是在调用方释放了传递给我们的 AlertViewData，原因在于调用方要异步执行这个 C 函数，但是不知道 C 函数何时执行结束。所以，调用方必须分配足够的空间给 AlertViewData，同时我们的 displayAlertView C 函数必须释放空间。现在我们在主队列调用 displayAlertView 函数并把 Context（保存 Alert View 数据的结构）传递给它：

```
- (BOOL)                  application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

   dispatch_queue_t mainQueue = dispatch_get_main_queue();

   AlertViewData *context = (AlertViewData *)
   malloc(sizeof(AlertViewData));

   if (context != NULL){
      context->title = "GCD";
      context->message = "GCD is amazing.";
      context->cancelButtonTitle = "OK";

      dispatch_async_f(mainQueue,
                       (void *)context,
                       displayAlertView);
   }

   self.window = [[UIWindow alloc] initWithFrame:
                  [[UIScreen mainScreen] bounds]];

   self.window.backgroundColor = [UIColor whiteColor];
   [self.window makeKeyAndVisible];
   return YES;
}
```

如果你调用了 NSThread 类的 currentThread 类方法，你会发现你分派到主队列的 Block Objects 或者 C 函数实际上在主线程上运行：

```
- (BOOL)                    application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

   dispatch_queue_t mainQueue = dispatch_get_main_queue();

   dispatch_async(mainQueue, ^(void) {
      NSLog(@"Current thread = %@", [NSThread currentThread]);
      NSLog(@"Main thread = %@", [NSThread mainThread]);
   });

   self.window = [[UIWindow alloc] initWithFrame:
                        [[UIScreen mainScreen] bounds]];

   self.window.backgroundColor = [UIColor whiteColor];
   [self.window makeKeyAndVisible];
   return YES;
}
```

这段代码的输出和下文看到的类似：

```
Current thread = <NSThread: 0x4b0e4e0>{name = (null), num = 1}
Main thread = <NSThread: 0x4b0e4e0>{name = (null), num = 1}
```

现在你知道了如何使用 GCD 完成 UI-related 任务。我们可以转换到其他主题，例如使用并发队列来并行执行任务（见 5.6 和 5.7）；如果有必要的话可以把代码和 UI-related 代码混合起来。

5.5.4. 参考

XXX

5.6. 用 GCD 同步执行 Non-UI-Related 任务

5.6.1. 问题

想要执行那些不包含任何 UI-related 代码的同步任务。

5.6.2. 方案

使用 dispatch_sync 函数。

5.6.3. 讨论

当执行那些与 UI 无关的任务，或者与 UI 交互的任务时，和执行其他任务一样，会需要大量时间，以上情况会经常出现。例如，你想下载一个图片并想在下载完成之后展现给用户。下载过程却和 UI 没有任何关系。对于任何与 UI 无关的任务，你可以使用 GCD 中的全局并发队列。它们允许同步和异步执行。但是同步执行并不是说你的程序在继续之前要等到代码编程结束。这里是说并发队列会一直等待到你的任务完成后，才会在任务继续队列中的下一个代码块。当你把一个 Block Object 放进并发队列，你自己的程序会会继续运行而不会等队列执行这段代码。这是因为并发队列，如同它们的名字暗示的那样，会在线程上而不是在主线程上运行自己的代码（此处有个例外：当一个任务通过 dispatch_sync 函数提交到并发队列或者一个串行队列时，如果可能的话，iOS 将在当前线程上运行这个任务，这个当前线程可能是主线程，主要取决于当时代码路径的位置。正如我们即将看到的，这是 GCD 上的一个优化程序。）如果你同步提交一个任务到一个并发队列，同时提交另一个同步任务到

另一个并发队列；相对而言这两个同步任务将异步运行，因为他们运行在两个不同的并发队列上。理解这一点很重要，正如我们将看到的那样，你想确定在 B 任务开始之前 A 任务完成了。为了保证这一点，把它们同时提交一个相同的队列。

你可以使用 dispatch_sync 函数在一个分派队列上执行同步任务。你必须做的事情就是提供一个此队列的句柄了，这个队列必须运行任务，并且一个代码块会在这个队列上执行。

我们看个例子，它输出整数 1 到 1000 两次，一个完整的序列紧跟着另一个，但是并不会阻止主线。我们可以创建一个为我们计数的 Block Object 并且把同一个 Block Object 调用两次：

```
void (^printFrom1To1000)(void) = ^{
    NSUInteger counter = 0;
    for (counter = 1;
            counter <= 1000;
            counter++){
        NSLog(@"Counter = %lu - Thread = %@",
                (unsigned long)counter,
                [NSThread currentThread]);
    }
};
```

现在我们用 GCD 来调用这个 Block Object：

```
dispatch_queue_t concurrentQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_sync(concurrentQueue, printFrom1To1000);
dispatch_sync(concurrentQueue, printFrom1To1000);
```

如果你运行这段代码，你会发现计数发生在主线程，即使你已经要求过并发队列执行这个任务。事实证明这是 GCD 的优化。Dispatch_sync 函数将使用当前线程——你分配任务时使用的线程——任何可能的情况下，作为优化的一部分会被编程到 GCD。关于这一点 Apple 这样说：

作为一个优化，这个函数调在可能的情况下在当前线程上调用了 Block。

—Grand Central Dispatch (GCD) Reference

执行时为了在一个分派队列上用 C 函数来同步代替 Block Object，使用 Dispatch_sync_f function。我们可以简单把这段我们刚才写的 printFrom1To1000 Block Object 翻译成 C 函数，就像这样：

```
void printFrom1To1000(void *paramContext){
    NSUInteger counter = 0;
    for (counter = 1;
            counter <= 1000;
            counter++){
        NSLog(@"Counter = %lu - Thread = %@",
                (unsigned long)counter,
                [NSThread currentThread]);
    }
}
```

现在我们用 Dispatch_sync_f function 在一个并发队列中来执行 PrintFrom1To1000 函数，结果如下所示：

```
dispatch_queue_t concurrentQueue =
    dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_sync_f(concurrentQueue,
```

```
                    NULL,
                    printFrom1To1000);
dispatch_sync_f(concurrentQueue,
                    NULL,
                    printFrom1To1000);
```

Dispatch_get_global_queue 函数的第一个参数说明了并发队列的优先级，这个属性 GCD 必须替程序员检索。优先级越高，将提供更多的 CPU Timeslice 来获取该队列执行的代码。你可以使用下面这些值作为 Dispatch_get_global_queue 函数的第一个参数：

DISPATCH_QUEUE_PRIORITY_LOW

您的任务比正常任务用到更少的 Timeslice。

DISPATCH_QUEUE_PRIORITY_DEFAULT

执行代码的默认系统优先级将应用于您的任务。

DISPATCH_QUEUE_PRIORITY_HIGH

和正常任务相比，更多的 Timeslices 会应用到你的任务中。

Dispatch_get_global_queue 函数的第二个参数已经保存了，你只要一直给它输入数值 0 就可以了。

在这一部分你了解到如果在并行队列分派同步执行的任务。在下个部分将介绍并发队列上的一步执行。在 5.11 中将介绍在你创建 APP 的串行队列上来同步和异步执行任务。

5.6.4. 参考

XXX

5.7. 在 GCD 上异步执行 Non-UI 相关任务

5.7.1. 问题

你想要在 GCD 的帮助下能够异步执行 Non-UI 相关任务。

5.7.2. 方案

在主队列、串行队列和并发队列上异步执行代码块才能见识到 GCD 的真正实力。我敢肯定，在这一部分结束的时候，你将会完全相信 GCD 是多线程应用的未来，并将完全取代现代应用中的线程。

要在分派队列上执行异步任务，你必须使用下面这些函数中的其中一个：

dispatch_async

为了异步执行向分派队列提交一个 Block Object（2 个都通过函数指定）；

dispatch_async_f

为了异步执行向分派队列提交一个 C 函数和一个上下文引用（3 项通过函数指定）

5.7.3. 讨论

我们来看个实例。我们来写一个 iOS APP，它能从网络的 URL 上下载一个图片。下载完成之后，APP 应该将图片展示给用户。这个是我们的计划，那么为了达到这个目的我们该如何使用目前为止掌握的关于 GCD 的技能:

1. 我们要启动一个异步并发队列上的 Block Object。
2. 一旦在这个 Block 中使用了 Dispatch_sync 函数，我们将启动另一个 Block Object 来从 URL 上同步下载图片。从一个异步代码块上同步下载一个 URL 可以保持这个队列运行同步函数，而不是主线程。当我们从主线程角度来看的话整个运作仍是异步的。我们关心的问题是在下载图片的过程中没有阻塞主线程。

3. 图片下载完毕后，为了在 UI 上将图片展示给用户，我们会在主线程上同步执行一个 Block Object(见 5.5)。我们计划的框架就是这么简单：
4.

```
dispatch_queue_t concurrentQueue =
   dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_async(concurrentQueue, ^{
   __block UIImage *image = nil;
   dispatch_sync(concurrentQueue, ^{
      /* Download the image here */
   });
   dispatch_sync(dispatch_get_main_queue(), ^{
      /* Show the image to the user here on the main queue*/
   });
});
```

在第一个下载图片的同步调用结束之后，第二个 dispatch_sync 调用会在队列中执行，这个调用负责展示图片；这就是我们确切需要的东西，因为在图片展示给用户之前我们必须等图片完全下载。所以在图片下载完成之后，我们执行第二个 Block Object，但这一次是在主队列上。

我们现在来下载图片并将其展示给用户。我们将通过 iPhone App 的一个视图控制器的 DidAppear:方法来实现：

```
- (void) viewDidAppear:(BOOL)paramAnimated{

   dispatch_queue_t concurrentQueue =
   dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

   dispatch_async(concurrentQueue, ^{

      __block UIImage *image = nil;

      dispatch_sync(concurrentQueue, ^{
         /* Download the image here */

         /* iPad's image from Apple's website. Wrap it into two
           lines as the URL is too long to fit into one line */
         NSString *urlAsString = @"http://images.apple.com/mobileme/features"\
                                 "/images/ipad_findyouripad_20100518.jpg";

         NSURL *url = [NSURL URLWithString:urlAsString];

         NSURLRequest *urlRequest = [NSURLRequest requestWithURL:url];

         NSError *downloadError = nil;
         NSData *imageData = [NSURLConnection
                              sendSynchronousRequest:urlRequest
                              returningResponse:nil
                              error:&downloadError];

         if (downloadError == nil &&
               imageData != nil){

            image = [UIImage imageWithData:imageData];
            /* We have the image. We can use it now */

         }
         else if (downloadError != nil){
```

```
            NSLog(@"Error happened = %@", downloadError);
        } else {
            NSLog(@"No data could get downloaded from the URL.");
        }

    });

    dispatch_sync(dispatch_get_main_queue(), ^{
        /* Show the image to the user here on the main queue*/

        if (image != nil){
            /* Create the image view here */
            UIImageView *imageView = [[UIImageView alloc]
                                      initWithFrame:self.view.bounds];

            /* Set the image */
            [imageView setImage:image];

            /* Make sure the image is not scaled incorrectly */
            [imageView setContentMode:UIViewContentModeScaleAspectFit];

            /* Add the image to this view controller's view */
            [self.view addSubview:imageView];

        } else {
            NSLog(@"Image isn't downloaded. Nothing to display.");
        }

    });

  });

}
```

和你在图 5-2 中看到的一样，我们已经成功下载了图片并在 UI 上创建了一个图片视图将其展现给用户。

图 5-2. 使用 GCD 来下载并展现图片给用户

我们来看下一个例子。假设我们在磁盘的一个文件夹内存储了 10,000 个随机数字的数组，我们想把这个数组下载到内存，让数字按照升序排列（最小的数字先出现在列表中），然后展现给用户。

用于显示的控件主要取决于你的代码是用于 iOS（理想情况下，你会使用一个 UITableView 实例）还是 Mac OSX（NSTableView 是不错的选择）。由于我们没有数组，那为什么我们不先创建一个数组然后在下载下来，最后展示出来呢？

此处有 2 个方法可以帮助我们在设备的磁盘上找到要保存 10,000 个随机数字数组的位置：

```
- (NSString *) fileLocation{

   /* Get the document folder(s) */
   NSArray *folders =
   NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                       NSUserDomainMask,
                                       YES);

   /* Did we find anything? */
   if ([folders count] == 0){
      return nil;
   }

   /* Get the first folder */
   NSString *documentsFolder = [folders objectAtIndex:0];

   /* Append the file name to the end of the documents path */
   return [documentsFolder
           stringByAppendingPathComponent:@"list.txt"];
```

```
}
- (BOOL) hasFileAlreadyBeenCreated{

   BOOL result = NO;

   NSFileManager *fileManager = [[NSFileManager alloc] init];
   if ([fileManager fileExistsAtPath:[self fileLocation]]){
      result = YES;
   }

   return result;
}
```

现在的重点部分是：当且仅当我们之前没有在磁盘上创建这个数组，我们才能把这个 10,000 随时数字组成的数组保存到磁盘上。如果我们已经有了数组，我们会马上将其下从磁盘上下载。如果我们之前在磁盘上创建数组，我们先要创建一个然后从磁盘上下载。最后，如果数组能从磁盘上成功读出的话，我们会找到一个升序排列的数组并通过 UI 将结果展现给用户。我会把展示结果的环节留给你来完成：

```
dispatch_queue_t concurrentQueue =
   dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
/* If we have not already saved an array of 10,000
 random numbers to the disk before, generate these numbers now
 and then save them to the disk in an array */
dispatch_async(concurrentQueue, ^{

   NSUInteger numberOfValuesRequired = 10000;
   if ([self hasFileAlreadyBeenCreated] == NO){
      dispatch_sync(concurrentQueue, ^{

         NSMutableArray *arrayOfRandomNumbers =
         [[NSMutableArray alloc] initWithCapacity:numberOfValuesRequired];

         NSUInteger counter = 0;
         for (counter = 0;
               counter < numberOfValuesRequired;
               counter++){
            unsigned int randomNumber =
            arc4random() % ((unsigned int)RAND_MAX + 1);

            [arrayOfRandomNumbers addObject:
              [NSNumber numberWithUnsignedInt:randomNumber]];
         }

         /* Now let's write the array to disk */
         [arrayOfRandomNumbers writeToFile:[self fileLocation]
                                atomically:YES];

      });
   }

   __block NSMutableArray *randomNumbers = nil;

   /* Read the numbers from disk and sort them in an
    ascending fashion */
   dispatch_sync(concurrentQueue, ^{
```

```
    /* If the file has now been created, we have to read it */
    if ([self hasFileAlreadyBeenCreated]){
      randomNumbers = [[NSMutableArray alloc]
                       initWithContentsOfFile:[self fileLocation]];

      /* Now sort the numbers */
      [randomNumbers sortUsingComparator:
       ^NSComparisonResult(id obj1, id obj2) {

         NSNumber *number1 = (NSNumber *)obj1;
         NSNumber *number2 = (NSNumber *)obj2;
         return [number1 compare:number2];

       }];
    }
  });


  dispatch_async(dispatch_get_main_queue(), ^{
    if ([randomNumbers count] > 0){
      /* Refresh the UI here using the numbers in the
       randomNumbers array */
    }
  });
});
```

与同步和异步 Block 或者函数执行相比，关于 GCD 的内容更多。在 5.10 你会了解如何将 Block Objects 归类，如何准备其在一个分派队列中执行。我同时建议你看看 5.8 和 5.9 中关于 GCD 能够提供给程序员的其他功能。

5.7.4. 参考

XXX

5.8. 利用 GCD 延時後執行任務

5.8.1. 问题

你想能够在你想指定一定数量的延迟之后，使用 GCD 来执行代码。

5.8.2. 方案

使用 Dispatch_after 和 Dispatch_after_f 函数。

5.8.3. 讨论

通过核心基础，您可以在一段给定的时间之后调用对象中的选择器，这个调用可以使用 NSObject 类的 performSelector:withObject:afterDelay:方法。这儿有一个例子：

```
- (void) printString:(NSString *)paramString{
  NSLog(@"%@", paramString);
}
- (BOOL)                  application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  [self performSelector:@selector(printString:)
```

```
                withObject:@"Grand Central Dispatch"
                afterDelay:3.0];
  self.window = [[UIWindow alloc] initWithFrame:
                    [[UIScreen mainScreen] bounds]];

  // Override point for customization after application launch.
  self.window.backgroundColor = [UIColor whiteColor];
  [self.window makeKeyAndVisible];
  return YES;
}
```

在这个例子中我们要求运行库在 3 秒钟的延迟之后调用了 printString:方法。在 GCD 中我们可以使用 dispatch_after 和 dispatch_after_f 函数达到一样的目的，两个函数分别描述如下：

dispatch_after

在一段给定的、以纳秒为单位的时间段之后，将 Block Object 分派到一个分派队列。这个函数需要的参数有：

Delay in nanoseconds

在执行指定的 Block Object(由第三个参数指定)之前 GCD 必须在一个给定的分派队列(由第二个参数指定) 上等待的纳秒数。

Dispatch queue

Block Object (由第三个参数指定)在给定的延迟(由第一个参数指定)之后必须执行在其上执行的派送队列。

Block object

在指定的分派队列上等待一定纳秒后 Block Object 会被调用；它没有返回值并且不接受参数 (见 5.1)。

dispatch_after_f

分派一个 C 函数到 GCD 在限定的、以纳秒限定的时间之后执行，这个函数接受 4 个参数：

Delay in nanoseconds

在执行给定函数之前 (由第四个参数指定)GCD 必须在指定派送队列(由第二个参数指定)上等待的纳秒数。

Dispatch queue

C 函数 (由第四个参数指定) 必须在给定延迟(由第一个参数指定)之后在其上执行分派送队列。

Context

一个值在堆中传递到了 C 函数的内存地址 (举例见 5.5)。

C 函数

在特定时间段(由第一个参数指定)之后必须执行的 C 函数在给定分派队列中(由第二个参数指定)的地址。

虽然延迟是纳秒级，但主要由 iOS 决定派送延迟的粒度。当你指定纳秒值时，这个延迟可能没有你希望那么准确按。

我们先来看一个 dispatch_after 的例子：

```
double delayInSeconds = 2.0;
```

```
dispatch_time_t delayInNanoSeconds =
   dispatch_time(DISPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC);
dispatch_queue_t concurrentQueue =
   dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_after(delayInNanoSeconds, concurrentQueue, ^(void){
   /* Perform your operations here */
});
```

像你看到的那样，对于 dispatch_after 和 dispatch_after_f 函数纳秒级延迟参数必须是类 dispatch_time_t，它是绝对时间的抽象表示形式。要得到这个参数的值，你要用此代码中演示的 dispatch_time 函数。这里是您可以传递给 dispatch_time 函数的参数：

Base time

假设这个值为 B，Delta parameter 值为 D，通过这个函数结束的时间就等于 B+D。您可以设置此参数的值到 DISPATCH_TIME_NOW 来把现在作为基时间，然后从现在使用 Delta 参数来确定 Delta。

Delta to add to base time

这个参数是要增加到计算时间参数来获取函数结果的纳秒。

例如，表示一个从现在开始 3 秒的时间，你可以这样来写你的代码：

```
dispatch_time_t delay =
dispatch_time(DISPATCH_TIME_NOW, 3.0f * NSEC_PER_SEC);
```

或者表示从现在开始半秒的时间：

```
dispatch_time_t delay =
dispatch_time(DISPATCH_TIME_NOW, (1.0 / 2.0f) * NSEC_PER_SEC);
```

现在我们看看如何使用 dispatch_after_f 函数：

```
void processSomething(void *paramContext){
   /* Do your processing here */
   NSLog(@"Processing...");
}
- (BOOL)                    application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
   double delayInSeconds = 2.0;

   dispatch_time_t delayInNanoSeconds =
      dispatch_time(DISPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC);

   dispatch_queue_t concurrentQueue =
      dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

         dispatch_after_f(delayInNanoSeconds,
                          concurrentQueue,
                          NULL,
                          processSomething);
         self.window = [[UIWindow alloc] initWithFrame:
                        [[UIScreen mainScreen] bounds]];

         // Override point for customization after application launch.
         self.window.backgroundColor = [UIColor whiteColor];
         [self.window makeKeyAndVisible];
         return YES;
      }
```

5.8.4. 参考

XXX

5.9. 在 GCD 上一个任务最多执行一次

5.9.1. 问题

在 APP 的生命周期内你想确保每段代码只执行一次，即使它在代码的不同地方多次调用（比如单例的初始化）。

5.9.2. 方案

使用 dispatch_once 函数。

5.9.3. 讨论

分配和初始化一个单例是在 APP 的生命周期内必须发生的其中一个任务。我相信你知道在其他情况下，你必须确保在您的应用程序的生命周期内一段代码只被执行一次。当您尝试执行一段代码时，GCD 让你指定它的一个标示符。如果 GCD 检测到这个标识符以前已经传递到框架，它不会再次执行该代码块。允许你这么做的函数是 dispatch_once，它接受 2 个参数：

Token

一个类 dispatch_once_t 的 Token 持有 GCD 是代码块第一次执行时生成的 Token。如果你想最多一次执行一段代码，那么无论什么时候这个代码在 APP 中被调用你必须指定在这个方法中指定同样的 Token。我们很快会看到一个例子。

Block Object

Block Object 最多执行一次。这个 Block Object 返回时没有值并且没有参数。dispatch_once 总是被发出呼叫的代码用来在当前队列上执行任务，它可能是一个串行队列、一个并发队列或者一个主队列。这儿有一个例子：

```
static dispatch_once_t onceToken;
void (^executedOnlyOnce)(void) = ^{

   static NSUInteger numberOfEntries = 0;
   numberOfEntries++;
   NSLog(@"Executed %lu time(s)", (unsigned long)numberOfEntries);

};
- (BOOL)                    application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

   dispatch_queue_t concurrentQueue =
      dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

   dispatch_once(&onceToken, ^{
      dispatch_async(concurrentQueue,
                     executedOnlyOnce);
   });

   dispatch_once(&onceToken, ^{
      dispatch_async(concurrentQueue,
                     executedOnlyOnce);
   });

   self.window = [[UIWindow alloc] initWithFrame:
                  [[UIScreen mainScreen] bounds]];
```

```
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;

}
```

和你看到的一样，虽然我们使用 dispatch_once 两次尝试调 executedOnlyOnce Block Object，事实上 GCD 对于这个 Block Object 只执行了一次，因为两次传递到 dispatch_once 函数的标识符都一样。

Apple 在其 Cocoa Fundamentals Guide 向程序员展示了如何创建一个单例。这个源代码非常老，还没有更新到使用 GCD 或者 ARC（自动引用计数）。

为了初始化一个对象的共享实例，我们可以改变这种模式来使用 GCD 和 dispatch_once 函数，就像这样：

```
#import "MySingleton.h"
@implementation MySingleton
- (id) sharedInstance{
    static MySingleton *SharedInstance = nil;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        SharedInstance = [MySingleton new];
    });
    return SharedInstance;
}
@end
```

5.9.4. 参考

XXX

5.10. 用 GCD 将任务分组

5.10.1. 问题

由于彼此之间的依赖关系，你想将代码块分组来确保它们被 GCD 逐个执行

5.10.2. 方案

使用 dispatch_group_create 函数在 GCD 上创建一个组。

5.10.3. 讨论

GCD 让我们创建组，这些组允许你把任务放到一个位置，然后全部运行，运行结束后会从 GCD 收到一个通知。这一点有很多有价值的用途。例如，假设你有一个 UI-Base APP，想在 UI 上重新加载组件。你有一个表格视图，一个滚动视图，一个图片视图，就要用这些方法加载这些组建的内容：

```
- (void) reloadTableView{
    /* Reload the table view here */
    NSLog(@"%s", __FUNCTION__);
}
- (void) reloadScrollView{
    /* Do the work here */
    NSLog(@"%s", __FUNCTION__);
}
- (void) reloadImageView{
```

```
    /* Reload the image view here */
    NSLog(@"%s", __FUNCTION__);
}
```

此刻，这些方法是空白，但是稍后你会把 UI 相关代码放入其中，现在要依次调用 3 个方法。我们想知道什么时候 GCD 会完成调用，那样就可以把信息展示给用户了。为了达到这个目的，我们应该使用一个组。在 GCD 中使用组的时候你应该知道 4 个函数：

dispatch_group_create

创建一个组句柄。一旦你使用完了这个组句柄，应该使用 dispatch_release 函数将其释放。

dispatch_group_async

在一个组内提交一个代码块来执行。必须明确这个代码块属于哪个组，必须在哪个派送队列上执行。

dispatch_group_notify

允许你提交一个 Block Object。一旦添加到这个组的任务完成执行之后，这个 Block Object 应该被执行。这个函数也允许你明确执行 Block Object 的分派队列。

dispatch_release

这个函数释放那任何一个你通过 dispatch_group_create 函数创建的分派小组。

我们看个例子，正如前面解释的，在这个例子中我们想依次调用 reloadTableView，reloadScrollView 和 reloadImageView 方法并且一旦调用完成来向用户展示信息。要做到这一点我们可以使用 GCD 强大的分组功能：

```
dispatch_group_t taskGroup =    dispatch_group_create();
dispatch_queue_t mainQueue =    dispatch_get_main_queue();
/* Reload the table view on the main queue */
dispatch_group_async(taskGroup, mainQueue, ^{
    [self reloadTableView];
});
/* Reload the scroll view on the main queue */
dispatch_group_async(taskGroup, mainQueue, ^{
    [self reloadScrollView];
});
/* Reload the image view on the main queue */
dispatch_group_async(taskGroup, mainQueue, ^{
    [self reloadImageView];
});
/* At the end when we are done, dispatch the following block */
dispatch_group_notify(taskGroup, mainQueue, ^{
    /* Do some processing here */
    [[[UIAlertView alloc] initWithTitle:@"Finished"
                                message:@"All tasks are finished"
                               delegate:nil
                      cancelButtonTitle:@"OK"
                      otherButtonTitles:nil, nil] show];

});
/* We are done with the group */
dispatch_release(taskGroup);
```

除了 dispatch_group_async，你也可以分派异步的 C 函数到一个分派组来使用 dispatch_group_async_f 函数。

GCDAppDelegate 是获取这个例子的类的名称。为了转换上下文对象我们必须使用这个类名称，从而使编译器理解我们的指令。
就像这样：

```
void reloadAllComponents(void *context){

   Grouping_Tasks_Together_with_GCDAppDelegate *self =
      (__bridge Grouping_Tasks_Together_with_GCDAppDelegate *)context;

   [self reloadTableView];
   [self reloadScrollView];
   [self reloadImageView];

}
- (BOOL)                    application:(UIApplication *)application
      didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

   dispatch_group_t taskGroup =    dispatch_group_create();
   dispatch_queue_t mainQueue =    dispatch_get_main_queue();

   dispatch_group_async_f(taskGroup,
                          mainQueue,
                          (__bridge void *)self,
                          reloadAllComponents);

   /* At the end when we are done, dispatch the following block */
   dispatch_group_notify(taskGroup, mainQueue, ^{
      /* Do some processing here */
      [[[UIAlertView alloc] initWithTitle:@"Finished"
                                  message:@"All tasks are finished"
                                 delegate:nil
                        cancelButtonTitle:@"OK"
                        otherButtonTitles:nil, nil] show];

   });

   /* We are done with the group */
   dispatch_release(taskGroup);

   self.window = [[UIWindow alloc] initWithFrame:
                  [[UIScreen mainScreen] bounds]];

   self.window.backgroundColor = [UIColor whiteColor];
   [self.window makeKeyAndVisible];
   return YES;
}
```

由于 dispatch_group_async_f 接受 C 函数作为一个代码块来执行，那么 C 函数必须有一个引用到 Self，这个 Self 能够调用当前对象的实例方法，其中 C 函数就在这个当前对象中被执行。这就是把 self 作为上下文指针传递到 dispatch_group_async_f 函数中背后的原因。关于上下文背景和 C 函数的更多内容请参考 5.5。一旦给定的任务全部完成，用户将看到一个和图 5-3 展示的内容类似的结果。

图 5-3. 用 GCD 管理一组任务

5.10.4. 参考

XXX

5.11. 用 GCD 构建自己的分派队列

5.11.1. 问题

想要创建你自己的、独特命名的分派队列。

5.11.2. 方案

使用 dispatch_queue_create 函数。

5.11.3. 讨论

利用 GCD，你可以创建你自己的串行分派队列 (见 5.0 中串行队列介绍)，串行调度队列按照先入先出（FIFO）的原则运行它们的任务。然而，串行队列上的异步任务不会在主线程上执行，这就使得串行队列极需要并发 FIFO 任务。所有提交到一个串行队列的同步任务会在当前线程上执行，在任何可能的情况下这个线程会被提交任务的代码使用。但是提交到串行队列的异步任务总是在主线程以外的线程上执行。

我们将使用 dispatch_queue_create 函数创建串行队列。这个函数的第一个参数是 C 字符串（char *），它将唯一标识系统中的串行队列。我强调系统的原因是这个标识符是一个全系统标识符，意味着你的 APP 创建了一个新的带有 serialQueue1 标识符的串行队列，和别人的 APP 的标识符相同，GCD 无法确定创建一个新的有相同命名的串行队列的结果。因为这一点，Apple 强烈推荐你使用反向 DNS 格式的标识符，反向 DNS 标识符通常按照这样的方

法构建：com.COMPANY.PRODUCT.IDENTIFIER。例如，我可以创建 2 个串行队列并分别命名为：

```
com.pixolity.GCD.serialQueue1
com.pixolity.GCD.serialQueue2
```

创建了串行队列之后，你可以开始使用本书中学到的各种 GCD 功能向它分配任务。一旦你自己创建的串行队列使用完成工作后，你必须使用 dispatch_release 释放它。还想看个例子吗？我想是的!

```
dispatch_queue_t firstSerialQueue =
dispatch_queue_create("com.pixolity.GCD.serialQueue1", 0);
dispatch_async(firstSerialQueue, ^{
    NSUInteger counter = 0;
    for (counter = 0;
            counter < 5;
            counter++){
        NSLog(@"First iteration, counter = %lu", (unsigned long)counter);
    }
});
dispatch_async(firstSerialQueue, ^{
    NSUInteger counter = 0;
    for (counter = 0;
            counter < 5;
            counter++){
        NSLog(@"Second iteration, counter = %lu", (unsigned long)counter);
    }
});
dispatch_async(firstSerialQueue, ^{
    NSUInteger counter = 0;
    for (counter = 0;
            counter < 5;
            counter++){
        NSLog(@"Third iteration, counter = %lu", (unsigned long)counter);
    }
});
dispatch_release(firstSerialQueue);
```

如果你运行这段代码，在控制台窗口看下打印的输出，你会看到类似于下面的结果：

```
First iteration, counter = 0
First iteration, counter = 1
First iteration, counter = 2
First iteration, counter = 3
First iteration, counter = 4
Second iteration, counter = 0
Second iteration, counter = 1
Second iteration, counter = 2
Second iteration, counter = 3
Second iteration, counter = 4
Third iteration, counter = 0
Third iteration, counter = 1
Third iteration, counter = 2
Third iteration, counter = 3
Third iteration, counter = 4
```

很明显，虽然我们把 Block Objects 异步分派到了串行队列上，这个队列还是按照 FIFO

原则执行它的代码。我们可以修正同样的模板代买来使用 dispatch_async_f 函数而不是 dispatch_async 函数，就像这样：

```
void firstIteration(void *paramContext){

   NSUInteger counter = 0;
   for (counter = 0;
         counter < 5;
         counter++){
      NSLog(@"First iteration, counter = %lu", (unsigned long)counter);
   }
}
void secondIteration(void *paramContext){

   NSUInteger counter = 0;
   for (counter = 0;
         counter < 5;
         counter++){
      NSLog(@"Second iteration, counter = %lu", (unsigned long)counter);
   }
}
void thirdIteration(void *paramContext){

   NSUInteger counter = 0;
   for (counter = 0;
         counter < 5;
         counter++){
      NSLog(@"Third iteration, counter = %lu", (unsigned long)counter);
   }
}
- (BOOL)                    application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

   dispatch_queue_t firstSerialQueue =
   dispatch_queue_create("com.pixolity.GCD.serialQueue1", 0);

   dispatch_async_f(firstSerialQueue, NULL, firstIteration);
   dispatch_async_f(firstSerialQueue, NULL, secondIteration);
   dispatch_async_f(firstSerialQueue, NULL, thirdIteration);

   dispatch_release(firstSerialQueue);

   self.window = [[UIWindow alloc] initWithFrame:
                        [[UIScreen mainScreen] bounds]];
   self.window.backgroundColor = [UIColor whiteColor];
   [self.window makeKeyAndVisible];
   return YES;
}
```

5.11.4. 参考

XXX

5.12. 使用 Operation 同步运行任务

5.12.1. 问题

你想要执行一系列的同步任务。

5.12.2. 方案

创建 operations 并手动启动它们。

```
Create operations and start them manually:
#import <UIKit/UIKit.h>
@interface Running_Tasks_Synchronously_with_OperationsAppDelegate
                : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) NSInvocationOperation *simpleOperation;
@end
```

应用程序 delegate 的实现如下代码所示：

```
- (void) simpleOperationEntry:(id)paramObject{

   NSLog(@"Parameter Object = %@", paramObject);
   NSLog(@"Main Thread = %@", [NSThread mainThread]);
   NSLog(@"Current Thread = %@", [NSThread currentThread]);

}
- (BOOL)                    application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

   NSNumber *simpleObject = [NSNumber numberWithInteger:123];

   self.simpleOperation = [[NSInvocationOperation alloc]
                                    initWithTarget:self
                                    selector:@selector(simpleOperationEntry:)
                                    object:simpleObject];

   [self.simpleOperation start];

   self.window = [[UIWindow alloc] initWithFrame:
                        [[UIScreen mainScreen] bounds]];
   self.window.backgroundColor = [UIColor whiteColor];
   [self.window makeKeyAndVisible];
   return YES;
}
```

这个程序在控制台窗口输出的内容与下面类似：

```
Parameter Object = 123
Main Thread = <NSThread: 0x6810280>{name = (null), num = 1}
Current Thread = <NSThread: 0x6810280>{name = (null), num = 1}
```

就如这个类(NSInvocationOperation)的名字一样，这种类型对象的主要职责就是调用某个对象里面的一个方法。使用 operations 调用一个对象函数是最直接的方法。

5.12.3. 讨论

一个 invocation operation，就如在本章介绍部分描述一样，可以调用某个对象里面的一个方法。你或许要问“它特别之处是什么？”。当一个 invocation operation 被添加到 operation 队列中时，invocation operation 的作用会很明显。在 operation 队列中，invocation operation 可以异步的调用某个目标对象的一个方法，并且在并行的线程中启动 operation。如果你看控制台窗口(本章中的方案小节里面)，你会注意到被 invocation operation 调用的方法里面，当前线程与主线程是相同的，这是因为主线程里面，application:didFinishLaun chingWithOptions:方法调用了 operation 的 start 方法。在 5.13 节里面，我们将学习如何利用 operation 队列的高级特性来运行异步任务。

除了上面的 invocation operation，还可以使用 block 或者 plain operations 来执行同步任务。下面给出的示例是使用 block operation，从 0 到 999 进行计数(application delegate 的.h 文件代码如下):

```
#import <UIKit/UIKit.h>
@interface Running_Tasks_Synchronously_with_OperationsAppDelegate
                : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic, strong) NSBlockOperation *simpleOperation;
@end
```

application delegate 的实现如下(.m 文件)：

```
- (BOOL)                application:(UIApplication *)application
  didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

  self.simpleOperation = [NSBlockOperation blockOperationWithBlock:^{
    NSLog(@"Main Thread = %@", [NSThread mainThread]);
    NSLog(@"Current Thread = %@", [NSThread currentThread]);
    NSUInteger counter = 0;
    for (counter = 0;
         counter < 1000;
         counter++){
      NSLog(@"Count = %lu", (unsigned long)counter);
    }
  }];

  /* Start the operation */
  [self.simpleOperation start];

  /* Print something out just to test if we have to wait
    for the block to execute its code or not */
  NSLog(@"Main thread is here");

  self.window = [[UIWindow alloc] initWithFrame:
                 [[UIScreen mainScreen] bounds]];
  self.window.backgroundColor = [UIColor whiteColor];
  [self.window makeKeyAndVisible];
  return YES;
}
```

如果运行程序，我们会看到 0 至 100 会按顺序打印到屏幕上，后面跟着打印出“Main thread is here”，如下：

```
Main Thread = <NSThread: 0x6810280>{name = (null), num = 1}
Current Thread = <NSThread: 0x6810280>{name = (null), num = 1}
...
Count = 991
Count = 992
Count = 993
Count = 994
Count = 995
Count = 996
Count = 997
Count = 998
Count = 999
Main thread is here
```

通过上面的示例，可以证明 block operation 运行在 main 线程中，block 里面的代码同样运行在 main 线程中，因为 block operation 是在 application:didFinish LaunchingWithOptions:方法中被启动的。从日志信息里面可以看出重要的一点，就是我们的 operation 阻塞了 main 线程，必须在 block operation 执行完毕之后，main 线程才能继续执行后面的任务。这是非常糟糕的编程实践。实际上，iOS 开发者必须利用任何办法和技术，确保 main 线程能随时能响应，因为 main 线程的主要任务是处理用户的输入。关于这方面，苹果是这样说的：

你应该注意在 main 线程中执行你的任务。Main 线程是程序处理触摸事件和用户输入的地方。为了确保程序总是能对用户的操作做出响应，你应该永远都不要使用 main 线程执行 long-running 任务，或者执行一个潜在的无限运行的任务，例如网络的访问。而是应该总是把这类任务移到后台线程中。具体执行方法是把每一个任务封装到 operation 对象中，并将其添加到一个 operation 队列中，当然，你也可以自己创建一个确定的线程。

要想了解更多的这个主题内容，浏览 iOS Reference Library 中的文档” Tuning for Performance and Responsiveness”即可，可用 URL 是：http://developer.apple.com/library/ios/#documentation/iphone/conceptual/iphoneosprogrammingguide/Performance/Performance.html

此外，调用和阻塞操作，可以继承 NSOperation 并在子类中执行你的任务。开始之前，你必须知道几点关于继承 NSOperation 需要注意的事情：

- 如果你不打算使用 operation 队列，你必须在 operation 的 start 方法中自己新建一个线程。如果你不想使用 operation 队列，以及不想让你的 operation 与其他 operation 进行异步操作，你可以手动开始，在 operation 的 start 方法里面简单的调用 main 方法。
- 在你自己实现的 operation 中，NSOperation 实例有两个重要的方法必须进行 overridden：isExecuting 和 isFinished。他们可以被其它的任意对象调用。在这两个方法中，必须返回一个线程安全值，这个值可以在 operation 中进行操作。一旦你的 operation 开始了，必须通过 KVO，告诉所有的监听者，你改变了这这两个方法返回的值。我们将在后面给出的示例代码中看到。
- 在 operation 的 main 方法里面，必须提供 autorelease pool，因为你的 operation 在未来的某个时刻会被添加到一个 operation 队列中。你必须确保你的 operation 运行，这里有两种方法：手动启动 operation 或者通过 operation 队列启动。
- 必须为你的 operation 提供一个初始化方法。每个 operation 必须指定一个初始化方法。所有 operation 的其它初始化方法，包括默认的 init 方法，必须调用指定的初始化方法，指定的初始化方法一般具有多个参数。其它的初始化方法必须确保传递的任何参数符合指定的初始化方法。

下面给出我们的 operation 对象的声明(.h 文件)：

```
#import <Foundation/Foundation.h>
@interface CountingOperation : NSOperation
/* Designated Initializer */
- (id) initWithStartingCount:(NSUInteger)paramStartingCount
endingCount:(NSUInteger)paramEndingCount;
@end
```

为了容易理解，给出的实现文件(.m 文件)代码比较长，如下：

```
#import "CountingOperation.h"
@implementation CountingOperation
NSUInteger startingCount;
NSUInteger endingCount;
BOOL finished;
BOOL executing;
- (id) init {
return([self initWithStartingCount:0
endingCount:1000]);
}
- (id) initWithStartingCount:(NSUInteger)paramStartingCount
endingCount:(NSUInteger)paramEndingCount{
self = [super init];
if (self != nil){
/* Keep these values for the main method */
startingCount = paramStartingCount;
endingCount = paramEndingCount;
}
return(self);
}
- (void) start {
/* If we are cancelled before starting, then
we have to return immediately and generate the
required KVO notifications */
if ([self isCancelled]){
/* If this operation *is* cancelled */
/* KVO compliance */
[self willChangeValueForKey:@"isFinished"];
finished = YES;
[self didChangeValueForKey:@"isFinished"];
return;
} else {
/* If this operation is *not* cancelled */
/* KVO compliance */
[self willChangeValueForKey:@"isExecuting"];
executing = YES;
/* Call the main method from inside the start method */
[self main];
[self didChangeValueForKey:@"isExecuting"];
}
}
- (void) main {
@try {
/* Here is our autorelease pool */
@autoreleasepool {
/* Keep a local variable here that must get set to YES
```

```
whenever we are done with the task */
BOOL taskIsFinished = NO;
/* Create a while loop here that only exists
if the taskIsFinished variable is set to YES or
the operation has been cancelled */
while (taskIsFinished == NO &&
[self isCancelled] == NO){
/* Perform the task here */
NSLog(@"Main Thread = %@", [NSThread mainThread]);
NSLog(@"Current Thread = %@", [NSThread currentThread]);
NSUInteger counter = startingCount;
for (counter = startingCount;
counter < endingCount;
counter++){
NSLog(@"Count = %lu", (unsigned long)counter);
}
/* Very important. This way we can get out of the
loop and we are still complying with the cancellation
rules of operations */
taskIsFinished = YES;
}
/* KVO compliance. Generate the
required KVO notifications */
[self willChangeValueForKey:@"isFinished"];
[self willChangeValueForKey:@"isExecuting"];
finished = YES;
executing = NO;
[self didChangeValueForKey:@"isFinished"];
[self didChangeValueForKey:@"isExecuting"];
}
}
@catch (NSException * e) {
NSLog(@"Exception %@", e);
}
}
- (BOOL) isFinished{
/* Simply return the value */
return(finished);
}
- (BOOL) isExecuting{
/* Simply return the value */
return(executing);
}
@end
```

我们可以按照下面的方法启动这个 operation：

```
#import "Running_Tasks_Synchronously_with_OperationsAppDelegate.h"
#import "CountingOperation.h"
@implementation Running_Tasks_Synchronously_with_OperationsAppDelegate
@synthesize window = _window;
@synthesize simpleOperation;
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
self.simpleOperation = [[CountingOperation alloc] initWithStartingCount:0
endingCount:1000];
[self.simpleOperation start];
NSLog(@"Main thread is here");
self.window = [[UIWindow alloc] initWithFrame:
```

```
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
@end
```

如果运行我们的代码，在控制台窗口会看到如下结果，跟我们使用 block operation 一样：

```
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Current Thread = <NSThread: 0x6810260>{name = (null), num = 1}
...
Count = 993
Count = 994
Count = 995
Count = 996
Count = 997
Count = 998
Count = 999
Main thread is here
```

5.12.4. 参考

XXX

5.13. 使用 Operation 异步运行任务

5.13.1. 问题

你想要执行一系列的异步(并发)任务。

5.13.2. 方案

使用 operation 队列，以及 NSOperation 的子类在 main 方法中分离出一个新的线程。

5.13.3. 讨论

如 5.12 节中所述，默认情况下，调用 operation 的 start 方法使其在线程上运行。通常我们在主线程中启动 operations。但是，在同一时间，我们希望 operations 运行在它们自己的线程中，而不是占用主线程的时间片。最好的解决方案是使用 operation 队列。不过，如果你想手动管理 operations（不建议），可以继承 NSOperation，并在 main 方法中新启一个线程。关于更多新启线程信息请参考 5.16 节。

我们继续介绍，使用一个 operation 队列并在该队列中添加两个简单的 invocation operations。（更多关于 invocation operations 信息，请参考本章的介绍章节。相关的示例代码，请参考 5.12 章节）下面在 application delegate 头文件(.h)中声明一个 operation 队列和两个 invocation operations：

```
#import <UIKit/UIKit.h>
@interface Running_Tasks_Asynchronously_with_OperationsAppDelegate
                : UIResponder <UIApplicationDelegate>
@property (nonatomic, strong) UIWindow *window;
@property (nonatomic, strong) NSOperationQueue          *operationQueue;
@property (nonatomic, strong) NSInvocationOperation *firstOperation;
@property (nonatomic, strong) NSInvocationOperation *secondOperation;
@end
```

application delegate 的实现文件(.m 文件)如下：

```
#import "Running_Tasks_Asynchronously_with_OperationsAppDelegate.h"
@implementation Running_Tasks_Asynchronously_with_OperationsAppDelegate
@synthesize window = _window;
@synthesize firstOperation;
@synthesize secondOperation;
@synthesize operationQueue;
- (void) firstOperationEntry:(id)paramObject{

    NSLog(@"%s", __FUNCTION__);
    NSLog(@"Parameter Object = %@", paramObject);
    NSLog(@"Main Thread = %@", [NSThread mainThread]);
    NSLog(@"Current Thread = %@", [NSThread currentThread]);

}
- (void) secondOperationEntry:(id)paramObject{

    NSLog(@"%s", __FUNCTION__);
    NSLog(@"Parameter Object = %@", paramObject);
    NSLog(@"Main Thread = %@", [NSThread mainThread]);
    NSLog(@"Current Thread = %@", [NSThread currentThread]);

}
- (BOOL)                    application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

    NSNumber *firstNumber = [NSNumber numberWithInteger:111];
    NSNumber *secondNumber = [NSNumber numberWithInteger:222];

    self.firstOperation =[[NSInvocationOperation alloc]
                                    initWithTarget:self
                                    selector:@selector(firstOperationEntry:)
                                    object:firstNumber];
self.secondOperation = [[NSInvocationOperation alloc]
                                      initWithTarget:self
                                      selector:@selector(secondOperationEntry:)
                                      object:secondNumber];

    self.operationQueue = [[NSOperationQueue alloc] init];

    /* Add the operations to the queue */
    [self.operationQueue addOperation:self.firstOperation];
    [self.operationQueue addOperation:self.secondOperation];

    NSLog(@"Main thread is here");

    self.window = [[UIWindow alloc] initWithFrame:
                           [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
@end
```

在实现的代码里面都发生了什么呢：

- 有两个方法：firstOperationEntry:和 secondOperationEntry:，每个方法都接收一个对象

作为参数，并且在控制台窗口打印出当前线程、主线程和参数。这两个入口函数的 invocation operation 将被添加到一个 operation 队列中。

- 我们初始化两个 NSInvocationOperation 类型对象，并给每个 operation 设置目标 selector 入口点，如之前所述。
- 然后我们初始化一个 NSOperationQueue 类型对象。（当然也可以在入口方法前面创建）队列对象将负责管理 operation 对象的并发。
- 我们调用 NSOperationQueue 的实例方法 addOperation:把每个 invocation operation 添加到 operation 队列中。在这里，operation 队列可能会也可能不会立即通过 nvocation operation 的 start 方法启动 invocation operation。但是，需要牢记重要的一点：添加 operations 至 operation 队列后，你不能手动启动 operations，必须交由 operation 队列负责。

现在，我们运行一次示例代码，在控制台窗口可以看到如下结果：

```
[Running_Tasks_Asynchronously_with_OperationsAppDelegate firstOperationEntry:]
Main thread is here
Parameter Object = 111
[Running_Tasks_Asynchronously_with_OperationsAppDelegate secondOperationEntry:]
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Parameter Object = 222
Current Thread = <NSThread: 0x6805c20>{name = (null), num = 3}
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Current Thread = <NSThread: 0x6b2d1d0>{name = (null), num = 4}
```

上面的结果说明我们的 invocation operations 运行在它们自己的线程中，并且与主线程是并行的，不会阻塞主线程。现在我们再次运行上面的代码，并注意观察控制台窗口的输出。将会看到这次的输出信息与之前的完全不同，如下：

```
Main thread is here
[Running_Tasks_Asynchronously_with_OperationsAppDelegate firstOperationEntry:]
[Running_Tasks_Asynchronously_with_OperationsAppDelegate secondOperationEntry:]
Parameter Object = 111
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Current Thread = <NSThread: 0x68247c0>{name = (null), num = 3}
Parameter Object = 222
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Current Thread = <NSThread: 0x6819b00>{name = (null), num = 4}
```

你可以清晰的看到主线程并没有被阻塞，那两个 invocation operations 与主线程并行的运行。这就证明了当把两个非并发的 operation 添加到 operation 队列中，就会并发运行了。Operation 队列管理需要运行的 operations。

如果我们子类化了一个 NSOperation 类，并且把这个子类的实例对象添加到了 operation 队列，我们需要做稍微的改动。记住以下几点：

- 由于当把 NSOperation 的子类对象添加到一个 operation 队列中，该对象会异步运行。由此，你必须 overrideNSOperation 的实例方法 isConcurrent，在该方法中返回 YES。
- 在 start 方法里面执行 main 任务之前，需要定期的调用 isCancelled 方法来检测该函数的返回值，以确定是退出 start 方法还是开始运行 operation。在这里，当 operation 添加到队列中后，operation 的 start 方法将会被 operation 队列调用，start 方法中，调用 isCanelled 方法确定 operation 是否被取消。如果 operation 被取消了，只需要从 start

方法中简单的返回即可。如果没被取消，会在 start 方法中调用 main 方法。

- 在 main task 实现部分中 override main 函数，main 函数将被 operation 执行。在这个函数里面确保分配和初始化 autorelease pool，并且在返回之前释放这个 pool。
- 重载 operation 的 isFinished 和 isExecuting 方法，这两个函数返回对应的 BOOL 值，代表 operation 是执行完毕还是在执行中。

下面是我们的 operation 声明(.h 文件)：

```
#import <Foundation/Foundation.h>
@interface SimpleOperation : NSOperation
/* Designated Initializer */
- (id) initWithObject:(NSObject *)paramObject;
@end
The implementation of the operation is as follows:
#import "SimpleOperation.h"
@implementation SimpleOperation
NSObject *givenObject;
BOOL finished;
BOOL executing;
- (id) init {
NSNumber *dummyObject = [NSNumber numberWithInteger:123];
return([self initWithObject:dummyObject]);
}
- (id) initWithObject:(NSObject *)paramObject{
self = [super init];
if (self != nil){
/* Keep these values for the main method */
givenObject = paramObject;
}
return(self);
}
- (void) start {
/* If we are cancelled before starting, then
we have to return immediately and generate the
required KVO notifications */
if ([self isCancelled]){
/* If this operation *is* cancelled */
/* KVO compliance */
[self willChangeValueForKey:@"isFinished"];
finished = YES;
[self didChangeValueForKey:@"isFinished"];
return;
} else {
/* If this operation is *not* cancelled */
/* KVO compliance */
[self willChangeValueForKey:@"isExecuting"];
executing = YES;
/* Call the main method from inside the start method */
[self main];
[self didChangeValueForKey:@"isExecuting"];
}
}
- (void) main {
@try {
@autoreleasepool {
/* Keep a local variable here that must get set to YES
whenever we are done with the task */
```

```
BOOL taskIsFinished = NO;
/* Create a while loop here that only exists
if the taskIsFinished variable is set to YES or
the operation has been cancelled */
while (taskIsFinished == NO &&
[self isCancelled] == NO){
/* Perform the task here */
NSLog(@"%s", __FUNCTION__);
NSLog(@"Parameter Object = %@", givenObject);
NSLog(@"Main Thread = %@", [NSThread mainThread]);
NSLog(@"Current Thread = %@", [NSThread currentThread]);
/* Very important. This way we can get out of the
loop and we are still complying with the cancellation
rules of operations */
taskIsFinished = YES;
}
/* KVO compliance. Generate the
required KVO notifications */
[self willChangeValueForKey:@"isFinished"];
[self willChangeValueForKey:@"isExecuting"];
finished = YES;
executing = NO;
[self didChangeValueForKey:@"isFinished"];
[self didChangeValueForKey:@"isExecuting"];
}
}
@catch (NSException * e) {
NSLog(@"Exception %@", e);
}
}
- (BOOL) isConcurrent{
return YES;
}
- (BOOL) isFinished{
/* Simply return the value */
return finished;
}
- (BOOL) isExecuting{
/* Simply return the value */
return executing;
}
@end
```

现在可以在其他任何类中使用上面定义的这个 operation 类了，比如在 application delegate 中。下面是 application delegate 的声明，使用了新的 operation 类，并将其添加到了新的 operation 队列中：

```
#import <UIKit/UIKit.h>
@class SimpleOperation;
@interface Running_Tasks_Asynchronously_with_OperationsAppDelegate
: UIResponder <UIApplicationDelegate>
@property (nonatomic, strong) UIWindow *window;
@property (nonatomic, strong) NSOperationQueue *operationQueue;
@property (nonatomic, strong) SimpleOperation *firstOperation;
@property (nonatomic, strong) SimpleOperation *secondOperation;
@end
```

application delegate 的实现部分如下：

```
#import "Running_Tasks_Asynchronously_with_OperationsAppDelegate.h"
#import "SimpleOperation.h"
@implementation Running_Tasks_Asynchronously_with_OperationsAppDelegate
@synthesize window = _window;
@synthesize firstOperation;
@synthesize secondOperation;
@synthesize operationQueue;
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
NSNumber *firstNumber = [NSNumber numberWithInteger:111];
NSNumber *secondNumber = [NSNumber numberWithInteger:222];
self.firstOperation = [[SimpleOperation alloc]
initWithObject:firstNumber];
self.secondOperation = [[SimpleOperation alloc]
initWithObject:secondNumber];
self.operationQueue = [[NSOperationQueue alloc] init];
/* Add the operations to the queue */
[self.operationQueue addOperation:self.firstOperation];
[self.operationQueue addOperation:self.secondOperation];
NSLog(@"Main thread is here");
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
@end
```

打印到控制台窗口的结果与之前使用并发 invocation operation 类似：

```
Main thread is here
-[SimpleOperation main]
-[SimpleOperation main]
Parameter Object = 222
Parameter Object = 222
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Main Thread = <NSThread: 0x6810260>{name = (null), num = 1}
Current Thread = <NSThread: 0x6a10b90>{name = (null), num = 3}
Current Thread = <NSThread: 0x6a13f50>{name = (null), num = 4}
```

5.13.4. 参考

XXX

5.14. 创建 Operations 之间的依赖

5.14.1. 问题

你想在某个 task 运行结束后才开始运行另外一个 task。

5.14.2. 方案

如果 operation B 在能运行之前必须等待 operation A 运行完毕，operation B 必须把添加

operation A 为自己的依赖，使用 NSOperation 的实例方法 addDependency:。如下所示：

```
[self.firstOperation addDependency:self.secondOperation];
```

firstOperation 和 secondOperation 都是 NSInvocationOperation 类型，我们会在“讨论”部分看到。在这行实例代码中，直到 secondOperation task 运行完毕，firstOperation 才会在 operation 队列中被运行。

5.14.3. 讨论

一个 operation 在它的所有依赖没有成功执行完毕前，是不会开始运行他的 task 的。默认情况下，刚刚初始化的一个 operation，是没有依赖的。

为了介绍依赖，我们对 5.13 中代码的 app delegate 实现进行稍微的改动，使用 addDependency：让第一个 operation 等待第二个 operation：

```
#import "Creating_Dependency_Between_OperationsAppDelegate.h"
@implementation Creating_Dependency_Between_OperationsAppDelegate
@synthesize window = _window;
@synthesize firstOperation;
@synthesize secondOperation;
@synthesize operationQueue;
- (void) firstOperationEntry:(id)paramObject{

    NSLog(@"First Operation - Parameter Object = %@",
          paramObject);

    NSLog(@"First Operation - Main Thread = %@",
          [NSThread mainThread]);

    NSLog(@"First Operation - Current Thread = %@",
          [NSThread currentThread]);

}
- (void) secondOperationEntry:(id)paramObject{

    NSLog(@"Second Operation - Parameter Object = %@",
          paramObject);

    NSLog(@"Second Operation - Main Thread = %@",
          [NSThread mainThread]);

    NSLog(@"Second Operation - Current Thread = %@",
          [NSThread currentThread]);

}
- (BOOL)                    application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{

    NSNumber *firstNumber = [NSNumber numberWithInteger:111];
    NSNumber *secondNumber = [NSNumber numberWithInteger:222];

    self.firstOperation = [[NSInvocationOperation alloc]
                           initWithTarget:self
                           selector:@selector(firstOperationEntry:)
                           object:firstNumber];
```

```
    self.secondOperation = [[NSInvocationOperation alloc]
                                    initWithTarget:self
                                    selector:@selector(secondOperationEntry:)
                                    object:secondNumber];

    [self.firstOperation addDependency:self.secondOperation];

    self.operationQueue = [[NSOperationQueue alloc] init];

    /* Add the operations to the queue */
    [self.operationQueue addOperation:self.firstOperation];
    [self.operationQueue addOperation:self.secondOperation];

    NSLog(@"Main thread is here");
self.window = [[UIWindow alloc] initWithFrame:
                        [[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
@end
```

现在来运行程序，你会在控制台窗口看到类似如下结果：

```
Second Operation - Parameter Object = 222
Main thread is here
Second Operation - Main Thread = <NSThread: 0x6810250>{name = (null), num = 1}
Second Operation - Current Thread = <NSThread: 0x6836ab0>{name = (null), num = 3}
First Operation - Parameter Object = 111
First Operation - Main Thread = <NSThread: 0x6810250>{name = (null), num = 1}
First Operation - Current Thread = <NSThread: 0x6836ab0>{name = (null), num = 3}
```

很明显可以看出，及时 operation 队列尝试同时运行两个 operation，但是 first operation 有一个依赖----second operation，因此必须要等到 second operation 结束以后 first operation 才能开始运行。

在任意时刻，如果你想要终止两个 operation 间的依赖关系，可以使用 removeDependency：operation 对象的一个实例方法。

5.14.4. 参考

XXX

5.15. 创建并发计时器

5.15.1. 问题

你想重复的执行一个特定任务，这个任务具有一定的延时。例如：只要你的程序在运行，你想每秒钟更新一次屏幕中的视图。

5.15.2. 方案

使用计时器：

```
- (void) paint:(NSTimer *)paramTimer{
/* Do something here */
```

```
NSLog(@"Painting");
}
```

```
- (void) startPainting{
self.paintingTimer = [NSTimer
scheduledTimerWithTimeInterval:1.0
target:self
selector:@selector(paint:)
userInfo:nil
repeats:YES];
}
```

```
- (void) stopPainting{
if (self.paintingTimer != nil){
[self.paintingTimer invalidate];
}
}
```

```
- (void)applicationWillResignActive:(UIApplication *)application{
[self stopPainting];
}
```

```
- (void)applicationDidBecomeActive:(UIApplication *)application{
[self startPainting];
}
```

Invalidate 方法同样会 release 计时器，我们就不用手动 release 了。正如你所看到的，我已经在头文件(.h 文件)中定义了一个叫 paintingTimer 的属性，如下：

```
#import <UIKit/UIKit.h>
@interface Creating_Concurrency_with_TimersAppDelegate
: UIResponder <UIApplicationDelegate>
@property (nonatomic, strong) UIWindow *window;
@property (nonatomic, strong) NSTimer *paintingTimer;
@end
```

5.15.3. 讨论

计时器在指定的时间间隔里面会出发一个事件。计时器必须在某个处理循环中被调度。用 NSTimer 定义的是一个没有被调度的计时器，它不会做任何事情，但是当你想要调度它时，是可行的。一旦你调用了这样的函数：scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:，计时器会变成一个被调度的计时器，就会根据你的请求触发事件。一个被调度的计时器也就是这个计时器被添加到了一个事件处理循环中。要想得到任意计时器触发的事件，我们必须在事件处理循环中调度它。稍后将通过一个例子来演示：创建一个没有被调度的计时器，然后手动在程序的主事件处理循环中调度它。

一旦创建了计时器，并添加到了循环处理中，无论是含蓄的还是明确的，每隔 n 秒钟，n 是由编程者指定的，计时器都将会调用由编程者指定的某个目标对象的方法。由于 n 是一个 floating 类型，可以指定几分之一秒。

这里有多种方法来创建、初始化和调度计时器。最简单的方法就是使用 NSTimer 的类方法 scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:，下面是该方法的参数介绍：

scheduledTimerWithTimeInterval

这是计时器在触发事件之前需要等到的秒数。例如，如果想让计时器每秒钟调目标对象的方法两次，则必须把这个参数设置为 0.5(把 1 秒钟分为两份)；如果想让目标对象方法每秒钟被调用四次，这个参数则需要设置为 0.25(把 1 秒钟分为四份)。

target

Target 是接收事件的目标对象。

selector

这是目标对象里接收事件的方法。

userInfo

这个对象是为了以后被使用(在目标对象的函数里面)，计时器拥有它。

repeats

指定计时器是重复的调用目标对象的方法(这种情况需要把参数设置为 YES)，还是只需要一次然后就停止(这种情况需要把参数设置为 NO)。

一旦创建了计时器并添加到了事件循环中，可以使用 NSTimer 的实例方法 invalidate 来停止和释放计时器。这不仅仅会释放计时器，如果有对象传递给了计时器(例如，传递给 NSTimer 的类方法 scheduledTimerWithTimeInterval:target:selector:userInfo:repeats: 的 userInfo 参数的对象)，同样计时器会释放这个对象，注：该对象在计时器的生命周期里面都被计时器所拥有。如果给 repeats 参数传递的是 NO，计时器将会在第一次事件触发以后失效，随后会释放掉它拥有的对象(如果有)。

你也可以使用其他的方法来创建被调度的计时器。NSTimer 的类方法 scheduledTimerWithTimeInterval:invocation:repeats:就是其中之一：

```
- (void) paint:(NSTimer *)paramTimer{
/* Do something here */
NSLog(@"Painting");
}
- (void) startPainting{
/* Here is the selector that we want to call */
SEL selectorToCall = @selector(paint:);
/* Here we compose a method signature out of the selector. We
know that the selector is in the current class so it is easy
to construct the method signature */
NSMethodSignature *methodSignature =
[[self class] instanceMethodSignatureForSelector:selectorToCall];
/* Now base our invocation on the method signature. We need this
invocation to schedule a timer */
NSInvocation *invocation =
[NSInvocation invocationWithMethodSignature:methodSignature];
[invocation setTarget:self];
[invocation setSelector:selectorToCall];
/* Start a scheduled timer now */
self.paintingTimer = [NSTimer scheduledTimerWithTimeInterval:1.0
invocation:invocation
repeats:YES];
}
- (void) stopPainting{
if (self.paintingTimer != nil){
[self.paintingTimer invalidate];
}
}
```

```
- (void)applicationWillResignActive:(UIApplication *)application{
[self stopPainting];
}
- (void)applicationDidBecomeActive:(UIApplication *)application{
[self startPainting];
}
```

我们可以把调度一个计时器与启动汽车的引擎相比较。别调度的计时器就是运行中的引擎。没有被调度的计时器就是一个已经准备好启动但是还没有运行的引擎。我们在程序里面，无论何时，都可以调度和取消调度计时器，就像根据我们所处的环境，决定汽车的引擎室启动还是停止。如果你想要在程序中，手动的在某一个确定时间点调度计时器，可以使用 NSTimer 的类方法 timerWithTimeInterval:target:selector:userInfo:repeats:，当你准备好的时候，可以把计时器添加到事件处理循环中：

```
- (void) startPainting{
self.paintingTimer = [NSTimer timerWithTimeInterval:1.0
target:self
selector:@selector(paint:)
userInfo:nil
repeats:YES];
/* Do your processing here and whenever you are ready,
use the addTimer:forMode instance method of the NSRunLoop class
in order to schedule the timer on that run loop */
[[NSRunLoop currentRunLoop] addTimer:self.paintingTimer
forMode:NSDefaultRunLoopMode];
}
```

NSRunLoop 的类方法 currentRunLoop 和 mainRunLoop 将各自返回程序的当前和主要的事件处理循环，就如函数名一样。

如同可以通过 NSInvocation 使用 scheduledTimerWithTimeInterval:invocation:repeats: 方法创建被调度的计时器一样，你通过 NSInvocation，同样可以使用 NSTimer 的类方法 timerWithTimeInterval:invocation:repeats:来创建一个未被调度的计时器。

```
- (void) paint:(NSTimer *)paramTimer{
/* Do something here */
NSLog(@"Painting");
}
- (void) startPainting{
/* Here is the selector that we want to call */
SEL selectorToCall = @selector(paint:);
/* Here we compose a method signature out of the selector. We
know that the selector is in the current class so it is easy
to construct the method signature */
NSMethodSignature *methodSignature =
[[self class] instanceMethodSignatureForSelector:selectorToCall];
/* Now base our invocation on the method signature. We need this
invocation to schedule a timer */
NSInvocation *invocation =
[NSInvocation invocationWithMethodSignature:methodSignature];
[invocation setTarget:self];
```

```
[invocation setSelector:selectorToCall];
self.paintingTimer = [NSTimer timerWithTimeInterval:1.0
invocation:invocation
repeats:YES];;
/* Do your processing here and whenever you are ready,
use the addTimer:forMode instance method of the NSRunLoop class
in order to schedule the timer on that run loop */
[[NSRunLoop currentRunLoop] addTimer:self.paintingTimer
forMode:NSDefaultRunLoopMode];
}
- (void) stopPainting{
if (self.paintingTimer != nil){
[self.paintingTimer invalidate];
}
}
- (void)applicationWillResignActive:(UIApplication *)application{
[self stopPainting];
}
- (void)applicationDidBecomeActive:(UIApplication *)application{
[self startPainting];
}
```

计时器会调用事件接收者---目标对象的方法，并把计时器本身当做参数传递过去。例如，之前介绍过的 paint 方法，下面演示它如何获得计时器，作为目标对象的方法，只有一个参数：

```
- (void) paint:(NSTimer *)paramTimer{
/* Do something here */
NSLog(@"Painting");
}
```

这个参数是事件触发计时器的一个指针。如果需要的话，你可以在此调用 invalidate 方法阻止计时器再次运行。你也可以调用 NSTimer 实例的 userInfo 方法取得被计时器拥有的对象(如果需要)。这个对象就是在 NSTimer 的初始化方法中传递进去的，它是直接传递到计时器中的，以供将来使用。

5.15.4. 参考

XXX

5.16. 创建并发线程

5.16.1. 问题

你想在程序中运行单独的任务时，拥有最大的控制权。例如，你想要根据用户要求，来运行一个长计算请求，同时，主线程 UI 可以自由的与用户交互和做别的事情。

5.16.2. 方案

在程序中使用线程：

```
- (void) downloadNewFile:(id)paramObject{
@autoreleasepool {
NSString *fileURL = (NSString *)paramObject;
NSURL *url = [NSURL URLWithString:fileURL];
```

```
NSURLRequest *request = [NSURLRequest requestWithURL:url];
NSURLResponse *response = nil;
NSError *error = nil;
NSData *downloadedData =
[NSURLConnection sendSynchronousRequest:request
returningResponse:&response
error:&error];
if ([downloadedData length] > 0){
/* Fully downloaded */
} else {
/* Nothing was downloaded. Check the Error value */
}
}
}
```

```
- (void)viewDidLoad {
[super viewDidLoad];
NSString *fileToDownload = @"http://www.OReilly.com";
[NSThread detachNewThreadSelector:@selector(downloadNewFile:)
toTarget:self
withObject:fileToDownload];
}
```

5.16.3. 讨论

任何一个 iOS 应用程序都是由一个或者多个线程构成的。在 iOS5 中，具有一个 view controller 的普通应用程序，刚开始的线程数目可以达到 4 个或者 5 个，这些线程是由系统库为了应用程序的链接而创建的。在应用程序中，无论你使用多线程或不使用，至少有 1 个线程被创建。该线程叫做”main UI 线程”，被附加到主事件处理循环中(main run loop)。

为了理解线程的作用，我们来做一个实验，假设我们有 3 个循环：

```
- (void) firstCounter{
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"First Counter = %lu", (unsigned long)counter);
}
}
```

```
- (void) secondCounter{
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"Second Counter = %lu", (unsigned long)counter);
}
}
```

```
- (void) thirdCounter{
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"Third Counter = %lu", (unsigned long)counter);
```

```
}
}
```

代码逻辑非常简单，对吧？他们都是从 0 至 1000，打印出他们各自的计数器数字。现在假设你想要运行这些计数器，通常会这样做：

```
- (void) viewDidLoad{
[super viewDidLoad];
[self firstCounter];
[self secondCounter];
[self thirdCounter];
}
```

上面的代码没必要放在 view controller 的 viewDidLoad 函数中。

现在打开控制台窗口并运行这个程序。你会看到第一个计时器运行完毕，然后是第二个计时器，最后是第三个计时器。也就是说这些循环是在同一个线程运行的。线程代码中被执行的每一块代码一直在运行，直到循环结束。

如果我们想要在同一时间里面运行所有的计数器，该如何办呢？当然，我们可以为每一个计数器创建单独的线程。但是稍等！我们已经知道程序已经有一个主线程可以使用了。所以，我们只需要创建 2 个线程，分别给计时器一和计时器二，把计时器三要做的事情放在主线程中即可。

```
- (void) firstCounter{
@autoreleasepool {
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"First Counter = %lu", (unsigned long)counter);
}
}
}
```

```
- (void) secondCounter{
@autoreleasepool {
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"Second Counter = %lu", (unsigned long)counter);
}
}
}
```

```
- (void) thirdCounter{
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"Third Counter = %lu", (unsigned long)counter);
}
}
```

```
- (void)viewDidLoad {
[super viewDidLoad];
[NSThread detachNewThreadSelector:@selector(firstCounter)
toTarget:self
withObject:nil];
[NSThread detachNewThreadSelector:@selector(secondCounter)
toTarget:self
withObject:nil];
/* Run this on the main thread */
[self thirdCounter];
}
```

由于 thirdCounter 函数没有运行在单独的线程中，所以不需要自动释放池(autorelease pool)。这个方法将在应用程序的主线程中运行，每一个 Cocoa Touch 程序都会自动的给该主线程创建一个自动释放池。一定要记住，我们是在 viewDidLoad 方法中创建的独立线程，我们必须在相同的 view controller 中的方法 viewDidUnload 中销毁这些线程。当系统发送了一个 low-memeory 警告时，视图会运行 viewDidUnload，一旦未能销毁在 viewDidLoad 方法中创建的独立线程，会引起内存泄露。为了解决这个问题，你需要需要有一个指针，在创建和初始化线程对象时指向该对象。请使用 NSTread 的实例方法 initWithTarget:selector:object:来初始化一个 NSThread 对象。这样，你将获得这个线程的一个指针，并且你可以使用线程的实例方法 start 来手动启动该线程。

在代码的最后通过调用 detachNewThreadSelector，把将第一个计数器和第二个计数器运行在独立的线程中。现在，如果你运行程序，将会在控制台窗口看到如下信息：

```
Second Counter = 921
Third Counter = 301
Second Counter = 922
Second Counter = 923
Second Counter = 924
First Counter = 956
Second Counter = 925
Counter = 957
Second Counter = 926
First Counter = 958
Third Counter = 302
Second Counter = 927
Third Counter = 303
Second Counter = 928
```

可以看出，这三个计时器是同时运行的，他们输出的内容是随机交替的。

每一个线程必须创建一个 autorelease pool。在 autorelease pool 被 release 之前，autorelease pool 会一直持有被 autoreleased 的对象的引用。在引用计数内存管理环境中这是一个非常重要的机制，例如 Cocoa Touch 中的对象就能够被 autoreleased。无论何时，在创建一个对象实例时，该对象的引用计数是 1，但是当创建的 autorelease pool 对象被 release 了，那么 autorelease 的对象同样会发送一个 release 消息，如果此时，它的引用计数仍然是 1，那么该对象将被销毁。

每一个线程都需要创建一个 autorelease pool，当做是该线程第一个被创建的对象。如果不这样做，如果不这样做，当线程退出的时候，你分配在线程中的对象会发生内存泄露。为了更好的理解，我们来看看下面的代码：

```
- (void) autoreleaseThread:(id)paramSender{

   NSBundle *mainBundle = [NSBundle mainBundle];
   NSString *filePath = [mainBundle pathForResource:@"AnImage"
                                                ofType:@"png"];

   UIImage *image = [UIImage imageWithContentsOfFile:filePath];

   /* Do something with the image */
   NSLog(@"Image = %@", image);

}
- (void)viewDidLoad {

   [super viewDidLoad];

   [NSThread detachNewThreadSelector:@selector(autoreleaseThread:)
                                toTarget:self
                              withObject:self];

}
```

如果你运行这段代码，在控制台窗口，会看到类似如下的信息：

```
*** __NSAutoreleaseNoPool(): Object 0x5b2c990 of
class NSCFString autoreleased with no pool in place - just leaking
*** __NSAutoreleaseNoPool(): Object 0x5b2ca30 of
class NSPathStore2 autoreleased with no pool in place - just leaking
*** __NSAutoreleaseNoPool(): Object 0x5b205c0 of
class NSPathStore2 autoreleased with no pool in place - just leaking
*** __NSAutoreleaseNoPool(): Object 0x5b2d650 of
class UIImage autoreleased with no pool in place - just leaking
```

上面的信息显示了我们创建的 autorelease 的 UIImage 实例产生了一个内存泄露，另外，FilePath 和其他的对象也产生了泄露。这是因为在我们的线程中，没有在开始的时候创建和初始化一个 autorelease pool。下面是正确的代码，你可以测试一下，确保它没有内存泄露：

```
- (void) autoreleaseThread:(id)paramSender{

   @autoreleasepool {
      NSBundle *mainBundle = [NSBundle mainBundle];
      NSString *filePath = [mainBundle pathForResource:@"AnImage"
                                                   ofType:@"png"];

      UIImage *image = [UIImage imageWithContentsOfFile:filePath];

      /* Do something with the image */
      NSLog(@"Image = %@", image);
   }
}
```

5.16.4. 参考

XXX

5.17. 调用后台方法

5.17.1. 问题

你想知道一个最简单的方法，来创建一个线程，而不需要直接处理线程。

5.17.2. 方案

使用 NSObject 的实例方法 performSelectorInBackground:withObject:

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
[self performSelectorInBackground:@selector(firstCounter)
withObject:nil];
[self performSelectorInBackground:@selector(secondCounter)
withObject:nil];
[self performSelectorInBackground:@selector(thirdCounter)
withObject:nil];
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

按照下面的方法实现计数方法：

```
- (void) firstCounter{
@autoreleasepool {
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"First Counter = %lu", (unsigned long)counter);
}
}
}
```

```
- (void) secondCounter{
@autoreleasepool {
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"Second Counter = %lu", (unsigned long)counter);
}
}
}
```

```
- (void) thirdCounter{
@autoreleasepool {
NSUInteger counter = 0;
for (counter = 0;
counter < 1000;
counter++){
NSLog(@"Third Counter = %lu", (unsigned long)counter);
```

```
}
}
}
```

5.17.3. 讨论

performSelectorInBackground:withObject:方法为我们在后台创建了一个线程。这等同于我们为 selectors 创建一个新的线程。最重要的事情是我们必须记住通过此方法为 selector 创建的线程，selector 必须有一个 autorelease pool，就想其它线程一样，在引用计数内存环境中。

5.17.4. 参考

XXX

5.18. 退出线程和计时器

5.18.1. 问题

你想停止线程或计时器的运行，或者防止再次触发。

5.18.2. 方案

对于计时器，使用 NSTimer 的实例方法 invalidate。而对于线程，使用 cancel 方法。在线程中避免使用 exit 方法，因为当调用了 exit 之后，线程就没有机会做清理工作，当你的应用程序结束时，会发生资源泄漏。

```
NSThread *thread = /* Get the reference to your thread here */;
[thread cancel];
NSTimer *timer = /* Get the reference to your timer here */;
[timer invalidate];
```

5.18.3. 讨论

退出一个计时器非常简单；只需要简单的调用计时器的 invalidate 实例方法即可。调用了 invalidate 方法之后，计时器不会再对它的目标对象触发任何时间。

然而，线程的退出有点复杂。当线程处于休眠状态时，调用了它的 cancel 方法，线程的循环在退出之前仍然会执行完它的全部任务。

下面我给你演示一下：

```
- (void) threadEntryPoint{
@autoreleasepool {
NSLog(@"Thread Entry Point");
while ([[NSThread currentThread] isCancelled] == NO){
[NSThread sleepForTimeInterval:4];
NSLog(@"Thread Loop");
}
NSLog(@"Thread Finished");
}
}
```

```
- (void) stopThread{
NSLog(@"Cancelling the Thread");
[self.myThread cancel];
NSLog(@"Releasing the thread");
self.myThread = nil;
}
```

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
self.myThread = [[NSThread alloc]
initWithTarget:self
selector:@selector(threadEntryPoint)
object:nil];
[self performSelector:@selector(stopThread)
withObject:nil
afterDelay:3.0f];
[self.myThread start];
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

这段代码创建了一个 NSThread 实例，并立即启动。我们的线程在执行它的任务之前，每次循环都会休眠 4 秒钟。然而，在线程开始之前，我们调用了 stopThread 方法，该方法会在 3 秒钟后执行，stopThread 方法调用了当前线程的 cancel 方法，试图让线程退出它的循环。现在让我们允许程序，可以看到控制台窗口打印出如下信息：

```
...
Thread Entry Point
Cancelling the Thread
Releasing the thread
Thread Loop
Thread Finished
```

你可以清晰的看到，即使在循环中间 cancel 请求已经触发了，但是我们的线程在退出之前，仍会完成当前的循环。这是一个非常普遍的陷阱，在执行任务之前，检查线程是否被 cancel 了，可以简单的避免外部影响内部线程的循环。我们通过重写上面的代码，在执行任务前，先检查外部影响，确定线程是否被 cancel 了。如下代码：

```
- (void) threadEntryPoint{
@autoreleasepool {
NSLog(@"Thread Entry Point");
while ([[NSThread currentThread] isCancelled] == NO){
[NSThread sleepForTimeInterval:0.2];
if ([[NSThread currentThread] isCancelled] == NO){
NSLog(@"Thread Loop");
}
}
NSLog(@"Thread Finished");
}
}
```

```
- (void) stopThread{
NSLog(@"Cancelling the Thread");
[self.myThread cancel];
NSLog(@"Releasing the thread");
self.myThread = nil;
}
```

```
- (BOOL) application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
self.myThread = [[NSThread alloc]
initWithTarget:self
selector:@selector(threadEntryPoint)
object:nil];
[self performSelector:@selector(stopThread)
withObject:nil
afterDelay:3.0f];
[self.myThread start];
self.window = [[UIWindow alloc] initWithFrame:
[[UIScreen mainScreen] bounds]];
self.window.backgroundColor = [UIColor whiteColor];
[self.window makeKeyAndVisible];
return YES;
}
```

5.18.4. 参考

XXX

点击这里访问：DevDiv.com 移动开发论坛