iOS 并发编程指南

原著：Apple Inc.

翻译：Kevin

联系：support@gungyi.com

网站：http://www.gungyi.com

鸣谢：GungYi 移动应用开发

CocoaChina 社区

时间：2011-12-09

目录

1. 简介

iOS 和 Mac OS 传统的并发编程模型是线程，不过线程模型伸缩性不强，而且编写正确的线程代码也不容易。Mac OS 和 iOS 采取“**异步设计方式**”来解决并发的问题。

引入的异步技术有两个：

- **Grand Central Dispatch（GCD）**：系统管理线程，你不需要编写线程代码。只需定义想要执行的任务，然后添加到适当的 dispatch queue。GCD 会负责创建线程和调度你的任务。系统直接提供线程管理，比应用实现更加高效。
- **Operation Queue**：Objective-C 对象，类似于 dispatch queue。你定义想要执行的任务，并添加任务到 operation queue，后者负责调度和执行这些任务。和 GCD 一样，Operation Queue 也管理了线程，更加高效。

1.1. Dispatch Queue

基于 C 的执行自定义任务机制。dispatch queue 按先进先出的顺序，串行或并发地执行任务。serial dispatch queue 一次只能执行一个任务，

直接当前任务完成才开始出列并启动下一个任务。而 concurrent dispatch queue 则尽可能多地启动任务并发执行。

优点：

- 直观而简单的编程接口
- 提供自动和整体的线程池管理
- 提供汇编级调优的速度
- 更加高效地使用内存
- 不会 trap 内核 under load
- 异步分派任务到 dispatch queue 不会导致 queue 死锁
- 伸缩性强
- serial dispatch queue 比锁和其它同步原语更加高效

1.2. Dispatch Sources

Dispatch Sources 是基于 C 的系统事件异步处理机制。一个 Dispatch Source 封装了一个特定类型的系统事件，当事件发生时提交一个特定的 block 对象或函数到 dispatch queue。你可以使用 Dispatch Sources 监控以下类型的系统事件：

- 定时器
- 信号处理器

- 描述符相关的事件
- 进程相关的事件
- Mach port 事件
- 你触发的自定义事件

1.3. Operation Queues

Operation Queues 是 Cocoa 版本的并发 dispatch queue，由 NSOperationQueue 类实现。dispatch queue 总是按先进先出的顺序执行任务，而 Operation Queues 在确定任务执行顺序时，还会考虑其它因素。最主要的一个因素是指定任务是否依赖于另一个任务的完成。你在定义任务时配置依赖性，从而创建复杂的任务执行顺序图

提交到 Operation Queues 的任务必须是 NSOperation 对象，operation object 封装了你要执行的工作，以及所需的所有数据。由于 NSOperation 是一个抽象基类，通常你需要定义自定义子类来执行任务。不过 Foundation framework 自带了一些具体子类，你可以创建并执行相关的任务。

Operation objects 会产生 key-value observing（KVO）通知，对于监控任务的进程非常有用。虽然 operation queue 总是并发地执行任务，你可以使用依赖，在需要时确保顺序执行

1.4. 异步设计技术

通过确保主线程自由响应用户事件，并发可以很好地提高应用的响应性。通过将工作分配到多核，还能提高应用处理的性能。但是并发也带来一定的额外开销，并且使代码更加复杂，更难编写和调试代码。

因此在应用设计阶段，就应该考虑并发，设计应用需要执行的任务，及任务所需的数据结构。

2. Operation Queues

基于 Objective-C，因此基于 Cocoa 的应用通常会使用 Operation Queues

2.1. Operation Objects

operation object 是 NSOperation 类的实例，封装了应用需要执行的任务，和执行任务所需的数据。NSOperation 本身是抽象基类，我们必须实现子类。Foundation framework 提供了两个具体子类，你可以直接使用：

类	描述

NSInvocationOperation	可以直接使用的类，基于应用的一个对象和 selector 来创建 operation object。如果你已经有现有的方法来执行需要的任务，就可以使用这个类。
NSBlockOperation	可以直接使用的类，用来并发地执行一个或多个 block 对象。operation object 使用“组”的语义来执行多个 block 对象，所有相关的 block 都执行完成之后，operation object 才算完成。
NSOperation	基类，用来自定义子类 operation object。继承 NSOperation 可以完全控制 operation object 的实现，包括修改操作执行和状态报告的方式。

所有 operation objects 都支持以下关键特性：

- 支持建立基于图的 operation objects 依赖。可以阻止某个 operation 运行，直到它依赖的所有 operation 都已经完成。
- 支持可选的 completion block，在 operation 的主任务完成后调用。
- 支持应用使用 KVO 通知来监控 operation 的执行状态。
- 支持 operation 优先级，从而影响相对的执行顺序
- 支持取消，允许你中止正在执行的任务

2.2. 并发 VS 非并发Operations

通常我们通过将 operation 添加到 operation queue 中来执行该操作。但是我们也可以手动调用 start 方法来执行一个 operation 对象，这样做不保证 operation 会并发执行。NSOperation 类对象的 isConcurrent 方法告诉你这个 operation 相对于调用 start 方法的线程，是同步还是异步执行的。isConcurrent 方法默认返回 NO，表示 operation 与调用线程同步执行。

如果你需要实现并发 operation，也就是相对调用线程异步执行的操作。你必须添加额外的代码，来异步地启动操作。例如生成一个线程、调用异步系统函数，以确保 start 方法启动任务，并立即返回。

多数开发者从来都不需要实现并发 operation 对象，我们只需要将 operations 添加到 operation queue。当你提交非并发 operation 到 operation queue 时，queue 会创建线程来运行你的操作，因此也能达到异步执行的目的。只有你不希望使用 operation queue 来执行 operation 时，才需要定义并发 operations。

2.3. 创建一个 NSInvocationOperation 对象

如果已经现有一个方法，需要并发地执行，就可以直接创建 NSInvocationOperation 对象，而不需要自己继承 NSOperation。

```
@implementation MyCustomClass
- (NSOperation*)taskWithData:(id)data {
    NSInvocationOperation* theOp = [[[NSInvocationOperation alloc]
        initWithTarget:self
        selector:@selector(myTaskMethod:) object:data] autorelease];

    return theOp;
}

// This is the method that does the actual work of the task.
- (void)myTaskMethod:(id)data {
    // Perform the task.
}
@end
```

2.4. 创建一个 NSBlockOperation 对象

NSBlockOperation 对象用于封装一个或多个 block 对象，一般创建时会添加至少一个 block，然后再根据需要添加更多的 block。

当 NSBlockOperation 对象执行时，会把所有 block 提交到默认优先级的并发 dispatch queue。然后 NSBlockOperation 对象等待所有 block 完成执行，最后标记自己已完成。因此可以使用 block operation 来跟踪一组执行中的 block，有点类似于 thread join 等待多个线程的结果。区别在

于 block operation 本身也运行在一个单独的线程，应用的其它线程在等待 block operation 完成时可以继续工作。

```
NSBlockOperation* theOp = [NSBlockOperation blockOperationWithBlock: ^{
      NSLog(@"Beginning operation.\n");
      // Do some work.
   }];
```

使用 addExecutionBlock: 可以添加更多 block 到这个 block operation 对象。如果需要顺序地执行 block，你必须直接提交到所需的 dispatch queue。

2.5. 自定义Operation对象

如果 block operation 和 invocation operation 对象不符合应用的需求，你可以直接继承 NSOperation，并添加任何你想要的行为。NSOperation 类提供通用的子类继承点，而且实现了许多重要的基础设施来处理依赖和 KVO 通知。继承所需的工作量主要取决于你要实现非并发还是并发的 operation。

定义非并发 operation 要简单许多，只需要执行主任务，并正确地响应取消事件；NSOperation 处理了其它所有事情。对于并发 operation，你必须替换某些现有的基础设施代码。

执行主任务

每个 operation 对象至少需要实现以下方法：

- 自定义 initialization 方法：初始化，将 operation 对象设置为已知状态
- 自定义 main 方法：执行你的任务

你也可以选择性地实现以下方法：

- main 方法中需要调用的其它自定义方法
- Accessor 方法：设置和访问 operation 对象的数据
- dealloc 方法：清理 operation 对象分配的所有内存
- NSCoding 协议的方法：允许 operation 对象 archive 和 unarchive

```
@interface MyNonConcurrentOperation : NSOperation {
   id myData;
}
-(id)initWithData:(id)data;
@end

@implementation MyNonConcurrentOperation
- (id)initWithData:(id)data {
```

```
   if (self = [super init])
      myData = [data retain];
   return self;
}

- (void)dealloc {
   [myData release];
   [super dealloc];
}

-(void)main {
   @try {
      NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];
      // Do some work on myData and report the results.
      [pool release];
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}
@end
```

响应取消事件

operation 开始执行之后，会一直执行任务直到完成，或者显式地取消操作。取消可能在任何时候发生，甚至在 operation 执行之前。尽管 NSOperation 提供了一个方法，让应用取消一个操作，但是识别出取消事件则是你的事情。如果 operation 直接终止，可能无法回收所有已分配的内存或资源。因此 operation 对象需要检测取消事件，并优雅地退出执行。

operation 对象定期地调用 isCancelled 方法，如果返回 YES（表示已取消），则立即退出执行。不管是自定义 NSOperation 子类，还是使用系统提供的两个具体子类，都需要支持取消。isCancelled 方法本身非常轻量，可以频繁地调用而不产生大的性能损失。以下地方可能需要调用 isCancelled：

- 在执行任何实际的工作之前
- 在循环的每次迭代过程中，如果每个迭代相对较长可能需要调用多次
- 代码中相对比较容易中止操作的任何地方

```
- (void)main {
   @try {
      NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

```
        BOOL isDone = NO;

        while (![self isCancelled] && !isDone) {
            // Do some work and set isDone to YES when finished
        }
        [pool release];
    }
    @catch(...) {
        // Do not rethrow exceptions.
    }
}
```

注意你的代码还需要完成所有相关的资源清理工作

为并发执行配置operations

Operation 对象默认按同步方式执行，也就是在调用 start 方法的那个线程中直接执行。由于 operation queue 为非并发 operation 提供了线程支持，对应用来说，多数 operations 仍然是异步执行的。但是如果你希望手工执行 operations，而且仍然希望能够异步执行操作，你就必须采取适当的措施，通过定义 operation 对象为并发操作来实现。

方法	描述
start	（必须）所有并发操作都必须覆盖这个方法，以自定义的实现替换默认行为。手动执行一个操作时，你会调用 start 方法。因此你对这个方法的实现是操作的起点，设置一个线程或其它执行环境，来执行你的任务。你的实现在任何时候都绝对不能调用 super。
main	（可选）这个方法通常用来实现 operation 对象相关联的任务。尽管你可以在 start 方法中执行任务，使用 main 来实现任务可以让你的代码更加清晰地分离设置和任务代码
isExecuting isFinished	（必须）并发操作负责设置自己的执行环境，并向外部 client 报告执行环境的状态。因此并发操作必须维护某些状态信息，以知道是否正在执行任务，是否已经完成任务。使用这两个方法报告自己的状态。 这两个方法的实现必须能够在其它多个线程中同时调用。另外这些方法报告的状态变化时，还需要为相应的 key path 产生适当的 KVO 通知。
isConcurrent	（必须）标识一个操作是否并发 operation，覆盖这个方法并返回 YES

```
@interface MyOperation : NSOperation {

    BOOL        executing;

    BOOL        finished;

}

- (void)completeOperation;

@end


@implementation MyOperation
```

```
- (id)init {
    self = [super init];
    if (self) {
        executing = NO;
        finished = NO;
    }
    return self;
}

- (BOOL)isConcurrent {
    return YES;
}

- (BOOL)isExecuting {
    return executing;
}

- (BOOL)isFinished {
    return finished;
}


- (void)start {
   // Always check for cancellation before launching the task.
   if ([self isCancelled])
   {
```

```
      // Must move the operation to the finished state if it is canceled.
      [self willChangeValueForKey:@"isFinished"];
      finished = YES;
      [self didChangeValueForKey:@"isFinished"];
      return;
   }

   // If the operation is not canceled, begin executing the task.
   [self willChangeValueForKey:@"isExecuting"];
   [NSThread detachNewThreadSelector:@selector(main) toTarget:self
withObject:nil];
   executing = YES;
   [self didChangeValueForKey:@"isExecuting"];
}

- (void)main {
   @try {
       NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

       // Do the main work of the operation here.

       [self completeOperation];
       [pool release];
   }
   @catch(...) {
      // Do not rethrow exceptions.
```

```
    }
}

- (void)completeOperation {
    [self willChangeValueForKey:@"isFinished"];
    [self willChangeValueForKey:@"isExecuting"];

    executing = NO;
    finished = YES;

    [self didChangeValueForKey:@"isExecuting"];
    [self didChangeValueForKey:@"isFinished"];
}
@end
```

即使操作被取消，你也应该通知 KVO observers，你的操作已经完成。当某个 operation 对象依赖于另一个 operation 对象的完成时，它会监测后者的 isFinished key path。只有所有依赖的对象都报告已经完成，第一个 operation 对象才会开始运行。如果你的 operation 对象没有产生完成通知，就会阻止其它依赖于你的 operation 对象运行。

维护KVO依从

NSOperation 类的 key-value observing（KVO）依从于以下 key paths：

- isCancelled
- isConcurrent
- isExecuting
- isFinished
- isReady
- dependencies
- queuePriority
- completionBlock

如果你覆盖 start 方法，或者对 NSOperation 对象的其它自定义运行（覆盖 main 除外），你必须确保自定义对象对这些 key paths 保留 KVO 依从。覆盖 start 方法时，需要关注 isExecuting 和 isFinished 两个 key paths。

如果你希望实现依赖于其它东西（非 operation 对象），你可以覆盖 isReady 方法，并强制返回 NO，直到你等待的依赖得到满足。如果你需要保留默认的依赖管理系统，确保你调用了[super isReady]。当你的 operation 对象的准备就绪状态发生改变时，生成一个 isReady 的 key path 的 KVO 通知。

除非你覆盖了 addDependency: 或 removeDependency: 方法，否则你不需要关注 dependencies key path

虽然你也可以生成 NSOperation 的其它 KVO 通知，但通常你不需要这样做。如果需要取消一个操作，你可以直接调用现有的 cancel 方法。类似地，你也很少需要修改 queue 优先级信息。最后，除非你的 operation 对象可以动态地改变并发状态，你也不需要提供 isConcurrent key path 的 KVO 通知。

2.6. 自定义一个Operation对象的执行行为

对 Operation 对象的配置发生在创建对象之后，将其添加到 queue 之前。

配置operation之间的依赖关系

依赖关系可以顺序地执行相关的 operation 对象，依赖于其它操作，则必须等到该操作完成之后自己才能开始。你可以创建一对一的依赖关系，也可以创建多个对象之间的依赖图。

使用 NSOperation 的 addDependency: 方法在两个 operation 对象之间建立依赖关系。表示当前 operation 对象将依赖于参数指定的目标 operation 对象。依赖关系不局限于相同 queue 中的 operations 对象，Operation 对象会管理自己的依赖，因此完全可以在不同的 queue 之间的 Operation 对象创建依赖关系。

唯一的限制是不能创建环形依赖，这是程序员的错误，所有受影响的 operations 都无法运行！

当一个 operation 对象依赖的所有其它对象都已经执行完成，该 operation 就变成准备执行状态（如果你自定义了 isReady 方法，则由你的方法确定是否准备好运行）。如果 operation 已经在一个 queue 中，queue 就可以在任何时候执行这个 operation。如果你需要手动执行该 operation，就自己调用 operation 的 start 方法。

配置依赖必须在运行 operation 和添加 operation 到 queue 之前进行，之后添加的依赖关系可能不起作用。

依赖要求每个 operation 对象在状态发生变化时必须发出适当的 KVO 通知。如果你自定义了 operation 对象的行为，就必须在自定义代码中生成适当的 KVO 通知，以确保依赖能够正确地执行。

修改Operation的执行优先级

对于添加到 queue 的 Operations，执行顺序首先由已入队列的 operations 是否准备好，然后再根据所有 operations 的相对优先级确定。是否准备好由对象的依赖关系确定，优先级等级则是 operation 对象本身的一个属性。默认所有 operation 都拥有“普通”优先级，不过你可以通过 setQueuePriority: 方法来提升或降低 operation 对象的优先级。

优先级只能应用于相同 queue 中的 operations。如果应用有多个 operation queue，每个 queue 的优先级等级是互相独立的。因此不同 queue 中的低优先级操作仍然可能比高优先级操作更早执行。

优先级不能替代依赖关系，优先级只是 queue 对已经准备好的 operations 确定执行顺序。先满足依赖关系，然后再根据优先级从所有准备好的操作中选择优先级最高的那个执行。

修改底层线程的优先级

Mac OS X 10.6 之后，我们可以配置 operation 底层线程的执行优先级，线程直接由内核管理，通常优先级高的线程会给予更多的执行机会。对于 operation 对象，你指定线程优先级为 0.0 到 1.0 之间的某个数值，0.0 表示最低优先级，1.0 表示最高优先级。默认线程优先级为 0.5

要设置 operation 的线程优先级，你必须在将 operation 添加到 queue 之前，调用 setThreadPriority: 方法进行设置。当 queue 执行该 operation 时，默认的 start 方法会使用你指定的值来修改当前线程的优先级。不过新的线程优先级只在 operation 的 main 方法范围内有效。其它所有代码仍然（包括 completion block）运行在默认线程优先级。

如果你创建了并发 operation，并覆盖了 start 方法，你必须自己配置线程优先级。

设置一个completion block

在 Mac OS X 10.6 之后，operation 可以在主任务完成之后执行一个 completion block。你可以使用这个 completion block 来执行任何不属于主任务的工作。例如你可以使用这个 block 来通知相关的 client，操作已经执行完成。而并发 operation 对象则可以使用这个 block 来产生最终的 KVO 通知。

调用 NSOperation 的 setCompletionBlock: 方法来设置一个 completion block，你传递的 block 应该没有参数和返回值。

2.7. 实现Operation对象的技巧

Operation对象的内存管理

Operation 对象需要良好的内存管理策略。

创建自己的AutoRelease Pool

operation 是 Objective-C 对象，你在实现任务的代码中应该创建一个 autorelease pool，这样可以保护那些 autorelease 对象得到尽快地释放。虽然你的自定义代码执行时可能已经有了一个 pool，但你不能依赖于这个行为，总是应该自己创建一个。

拥有自己的 autorelease pool 还能更加灵活地管理 operation 的内存。如果 operation 创建大量的临时对象，则可以考虑创建额外的 pool，来清理不再使用的临时对象。在 iOS 中特别需要注意，应迟早地清理不再使用的临时对象，避免内存警告。

```
- (void)main {
   @try {
      NSAutoreleasePool* pool = [[NSAutoreleasePool alloc] init];

      // Do the main work of the operation here.

      [pool release];
   }
   @catch(...) {
      // Do not rethrow exceptions.
   }
}
```

避免Per-Thread存储

虽然多数 operation 都在线程中执行，但对于非并发 operation，通常由 operation queue 提供线程，这时候 queue 拥有该线程，而你的应用不应该去动这个线程。特别是不要关联任何数据到不是你创建和拥有的线程。这些线程由 queue 管理，根据系统和应用的需求创建或销毁。

因此使用 Per-Thread storage 在 operations 之间传递数据是不可靠的，而且很有可能会失败。

对于 operation 对象，你完全没有理由使用 Per-Thread Storage，应该在创建对象的时候就给它需要的所有数据。所有输入和输出数据都应该存储在 operation 对象中，最后再整合到你的应用，或者最终释放掉。

根据需要保留Operation对象的引用

由于 operation 对象异步执行，你不能创建完以后就完全不管。它们也是对象，需要你来分配和释放它们管理的任何资源，特别是如果你需要在 operation 对象完成后获取其中的数据。

由于 queue 总是尽最大可能快速地调度和执行 operation，在你添加 operation 到 queue 时，可能立即就开始运行，当你稍后向 queue 请求 operation 对象的状态时，有可能 queue 已经执行完了相应的 operation 并从 queue 中删除了这个对象。因此你总是应该自己拥有 operation 对象的引用。

处理错误和异常

operation 本质上是应用中独立的实体，因此需要自己负责处理所有的错误和异常。NSOperation 默认的 start 方法并没有捕获异常。所以你自己的代码总是应该捕获并抑制异常。你还应该检查错误代码并适当地通知应用。如果你覆盖了 start 方法，你也必须捕获所有异常，阻止它离开底层线程的范围。

你需要准备好处理以下错误或异常：

- 检查并处理 UNIX errno 风格的错误代码
- 检查方法或函数显式返回的错误代码
- 捕获你的代码或系统 frameworks 抛出的异常
- 捕获 NSOperation 类自己抛出的异常，在以下情况 NSOperation 会抛出异常：
    - operation 没有准备好，但是调用了 start 方法
    - operation 正在执行或已经完成（可能被取消），再次调用了 start 方法。
    - 当你添加 completion block 到正在执行或已经完成的 operation
    - 当你试图获取已经取消 NSInvocationOperation 对象的结果

2.8. 为Operation对象确定一个适当的范围

和任何对象一样，NSOperation 对象也会消耗内存，执行时也会带来开销。因此如果 operation 对象只做很少的工作，但是却创建成千上万个小的 operation 对象，你就会发现更多的时间花在了调度 operations 而不是执行它们。

要高效地使用 Operations，关键是在 Operation 执行的工作量和保持计算机繁忙之间，找到最佳的平衡。确保每个 Operation 都有一定的工作量可以执行。例如 100 个 operations 执行 100 次相同任务，可以考虑换成 10 个 operations，每个执行 10 次。

你同样要避免向一个 queue 中添加过多的 operations，或者持续快速地向 queue 中添加 operation，超过 queue 所能处理的能力。这里可以考虑分批创建 operations 对象，在一批对象执行完之后，使用 completion block 告诉应用创建下一批 operations 对象。

2.9. 执行Operations

应用需要执行 Operations 来处理相关的工作，你有几种方法来执行 Operations 对象。

添加Operations到Operation Queue

执行 Operations 最简单的方法是添加到 operation queue，后者是 NSOperationQueue 对象。应用负责创建和维护自己使用的所有 NSOperationQueue 对象。

```
NSOperationQueue* aQueue = [[NSOperationQueue alloc] init];
```

调用 addOperation: 方法添加一个 operation 到 queue，Mac OS X 10.6 之后可以使用 addOperations:waitUntilFinished: 方法一次添加一组 operations，或者也可以直接使用 addOperationWithBlock: 方法添加 block 对象到 queue。

```
[aQueue addOperation:anOp]; // Add a single operation

[aQueue addOperations:anArrayOfOps waitUntilFinished:NO]; // Add multiple operations

[aQueue addOperationWithBlock:^{

   /* Do something. */

}];
```

Operations 添加到 queue 后，通常短时间内就会得到运行。但是如果存在依赖，或者 Operations 挂起等原因，也可能需要等待。

注意 Operations 添加到 queue 之后，绝对不要再修改 Operations 对象。因为 Operations 对象可能会在任何时候运行，因此改变依赖或数据会产生不利的影响。你只能通过 NSOperation 的方法来查看操作的状态，是否正在运行、等待运行、已经完成等。

虽然 NSOperationQueue 类设计用于并发执行 Operations，你也可以强制单个 queue 一次只能执行一个 Operation。

setMaxConcurrentOperationCount: 方法可以配置 operation queue 的最大并发操作数量。设为 1 就表示 queue 每次只能执行一个操作。不过 operation 执行的顺序仍然依赖于其它因素，像操作是否准备好和优先级等。因此串行化的 operation queue 并不等同于 GCD 中的串行 dispatch queue。

手动执行Operations

手动执行 Operation，要求 Operation 已经准备好，isReady 返回 YES，此时你才能调用 start 方法来执行它。isReady 方法与 Operations 依赖是结合在一起的。

调用 start 而不是 main 来手动执行 Operation，因为 start 在执行你的自定义代码之前，会首先执行一些安全检查。而且 start 还会产生 KVO

通知，以正确地支持 Operations 的依赖机制。start 还能处理 Operations 已经被取消的情况，此时会抛出一个异常。

手动执行 Operation 对象之前，还需要调用 isConcurrent 方法，如果返回 NO，你的代码可以决定在当前线程同步执行这个 Operation，或者创建一个独立的线程以异步执行。

下面方法演示了手动执行 Operation，如果这个方法返回 NO，表示不能执行，你需要设置一个定时器，稍后再次调用本方法，直到这个方法返回 YES，表示已经执行 Operation。

```
- (BOOL)performOperation:(NSOperation*)anOp
{
   BOOL        ranIt = NO;

   if ([anOp isReady] && ![anOp isCancelled])
   {
      if (![anOp isConcurrent])
         [anOp start];
      else
         [NSThread detachNewThreadSelector:@selector(start)
                  toTarget:anOp withObject:nil];
      ranIt = YES;
   }
   else if ([anOp isCancelled])
```

```
    {
        // If it was canceled before it was started,
        //  move the operation to the finished state.
        [self willChangeValueForKey:@"isFinished"];
        [self willChangeValueForKey:@"isExecuting"];
        executing = NO;
        finished = YES;
        [self didChangeValueForKey:@"isExecuting"];
        [self didChangeValueForKey:@"isFinished"];

        // Set ranIt to YES to prevent the operation from
        // being passed to this method again in the future.
        ranIt = YES;
    }
    return ranIt;
}
```

取消Operations

一旦添加到 operation queue，queue 就拥有了这个对象并且不能被删除，唯一能做的事情是取消。你可以调用 Operation 对象的 cancel 方法取消单个操作，也可以调用 operation queue 的 cancelAllOperations 方法取消当前 queue 中的所有操作。

只有你确定不再需要 Operations 对象时，才应该取消它。发出取消命令会将 Operations 对象设置为"Canceled"状态，会阻止它被执行。由于取消也被认为是完成，依赖于它的其它 Operations 对象会收到适当的 KVO 通知，并清除依赖状态，然后得到执行。

因此常见的做法是当发生重大事件时，一次性取消 queue 中的所有操作，例如应用退出或用户请求取消操作。

等待Operations完成

为了最佳的性能，你应该尽量设计你的应用尽可能地异步操作，让应用在操作正在执行时可以去处理其它事情。

如果创建 operation 的代码需要处理 operation 完成后的结果，可以使用 NSOperation 的 waitUntilFinished 方法等待 operation 完成。通常我们应该避免编写这样的代码，阻塞当前线程可能是一种简便的解决方案，但是它引入了更多的串行代码，限制了整个应用的并发性，同时也降低了用户体验。

绝对不要在应用主线程中等待一个Operation，只能在第二或次要线程中等待。阻止主线程将导致应用无法响应用户事件，应用也将表现为无响应。

除了等待单个 Operation 完成，你也可以同时等待一个 queue 中的所有操作，使用 NSOperationQueue 的 waitUntilAllOperationsAreFinished 方法。注意在等待一个 queue 时，应用的其它线程仍然可以往 queue 中添加 Operation，因此可能加长你线程的等待时间。

挂起和继续Queue

如果你想临时挂起 Operations 的执行，可以使用 setSuspended: 方法暂停相应的 queue。不过挂起一个 queue 不会导致正在执行的 Operation 在任务中途暂停，只是简单地阻止调度新 Operation 执行。你可以在响应用户请求时，挂起一个 queue，来暂停等待中的任务。稍后根据用户的请求，可以再次调用 setSuspended: 方法继续 Queue 中操作的执行。

3. Dispatch Queues

3.1. 简介

GCD dispatch queues 是执行任务的强大工具，允许你同步或异步地执行任意代码 block。原先使用单独线程执行的所有任务都可以替换为使用 dispatch queues。而 dispatch queues 最大的优点在于使用简单，而且更加高效。

dispatch queues 任务的概念就是应用需要执行的一些工作，如计算、创建或修改数据结构、处理数据等等。我们使用函数或 block 对象来定义任务，并添加到 dispatch queue。

dispatch queue 是类似于对象的结构体，管理你提交给它的任务，而且都是先进先出的数据结构。因此 queue 中的任务总是以添加的顺序开始执行。GCD 提供了几种 dispatch queues，不过你也自己创建。

类型	描述
串行	也称为 private dispatch queue，每次只执行一个任务，按任务添加顺序执行。当前正在执行的任务在独立的线程中运行（不同任务的线程可能不同），dispatch queue 管理了这些线程。通常串行 queue 主要用于对特定资源的同步访问。 你可以创建任意数量的串行 queues，虽然每个 queue 本身每次只能执行一个任务，但是各个 queue 之间是并发执行的。
并发	也称为 global dispatch queue，可以并发执行一个或多个任务，但是任务仍然是以添加到 queue 的顺序启动。每个任务运行于独立的线程中，dispatch queue 管理所有线程。同时运行的任务数量随时都会变化，而且依赖于系统条件。 你不能创建并发 dispatch queues。相反应用只能使用三个已经定义好的全局并发 queues。
Main dispatch	全局可用的串行 queue，在应用主线程中执行任务。这个 queue 与应用的 run loop 交叉执行。由于它运行在应用的主

queue	线程，main queue 通常用于应用的关键同步点。 虽然你不需要创建 main dispatch queue，但你必须确保应用适当地回收

应用使用 dispatch queue，相比线程有很多优点，最直接的优点是简单，不用编写线程创建和管理的代码，让你集中精力编写实际工作的代码。另外系统管理线程更加高效，并且可以动态调控所有线程。

dispatch queue 比线程具有更强的可预测性，例如两个线程访问共享资源，你可能无法控制哪个线程先后访问；但是把两个任务添加到串行 queue，则可以确保两个任务对共享资源的访问顺序。同时基于 queue 的同步也比基于锁的线程同步机制更加高效。

应用有效地使用 dispatch queue，要求尽可能地设计自包含、可以异步执行的任务。

dispatch queues 的几个关键点：

- dispatch queues 相对其它 dispatch queues 并发地执行任务，串行化任务只能在同一个 dispatch queue 中实现。
- 系统决定了同时能够执行的任务数量，应用在 100 个不同的 queues 中启动 100 个任务，并不表示 100 个任务全部都在并发地执行（除非系统拥有 100 或更多个核）
- 系统在选择执行哪个任务时，会考虑 queue 的优先级。
- queue 中的任务必须在任何时候都准备好运行，注意这点和 Operation 对象不同。
- private dispatch queue 是引用计数的对象。你的代码中需要 retain 这些 queue，另外 dispatch source 也可能添加到一个 queue，从而增加 retain 的计数。因此你必须确保所有 dispatch source 都被取消，而且适当地调用 release。

3.2. Queue相关的技术

除了 dispatch queue，GCD 还提供几个相关的技术，使用 queue 来帮助你管理代码。

技术	描述
Dispatch group	用于监控一组 block 对象完成（你可以同步或异步地监控 block）。Group 提供了一个非常有用的同步机制，你的代码可以等待其它任务的完成
Dispatch semaphore	类似于传统的 semaphore（信号量），但是更加高效。只有当调用线程由于信号量不可用，需要阻塞时，Dispatch semaphore 才会去调用内核。如果信号量可用，就不会与内核进行交互。使用信号量可以实现对有限资源的访问控制
Dispatch source	Dispatch source 在特定类型的系统事件发生时，会产生通知。你可以使用 dispatch source 来监控各种事件，如：进程通知、信号、描述符事件、等等。当事件发生时，dispatch source 异步地提交你的任务到指定的 dispatch queue，来进行处理。

3.3. 使用Block实现任务

Block 可以非常容易地定义“自包含”的工作单元，尽管看上去非常类似于函数指针，block 实际上由底层数据结构来表示，由编译器负责创建和管理。编译器对你的代码（和所有相关的数据）进行打包，封装为可以存在于堆中的格式，并在你的应用各个地方传递。

Block 最关键的优点能够使用 own lexical scope 之外的变量，在函数或方法内部定义一个 block 时，block 可以直接读取父 scope 中的变量。block 访问的变量全部被拷贝到 block 在堆中的数据结构，这样 block 就能在稍后自由地访问这些变量。当 block 被添加到 dispatch queue 中时，这些变量通常是只读格式的。不过同步执行的 Block 对象，可以使用那些定义为__block 的变量，对这些变量的修改会影响到调用 scope。

Block 的简单用法：

```
int x = 123;

int y = 456;


// Block declaration and assignment

void (^aBlock)(int) = ^(int z) {

    printf("%d %d %d\n", x, y, z);

};


// Execute the block

aBlock(789);   // prints: 123 456 789
```

设计 Block 时需考虑以下关键指导方针：

- 对于使用 dispatch queue 的异步 Block，可以在 Block 中安全地捕获和使用父函数或方法中的 scalar 变量。但是 Block 不应该去捕获大型结构体或其它基于指针的变量，这些变量由 Block 的调用上下文分配和删除。在你的 Block 被执行时，这些指针引用的内存可能已经不存在。当然，你自己显式地分配内存（或对象），然后让 Block 拥有这些内存的所有权，是安全可行的。
- Dispatch queue 对添加的 Block 会进行复制，在完成执行后自动释放。换句话说，你不需要在添加 Block 到 Queue 时显式地复制
- 尽管 Queue 执行小任务比原始线程更加高效，仍然存在创建 Block 和在 Queue 中执行的开销。如果 Block 做的事情太少，可能直接执行比 dispatch 到 queue 更加有效。使用性能工具来确认 Block 的工作是否太少
- 绝对不要针对底层线程缓存数据，然后期望在不同 Block 中能够访问这些数据。如果相同 queue 中的任务需要共享数据，应该使用 dispatch queue 的 context 指针来存储这些数据。
- 如果 Block 创建了大量 Objective-C 对象，考虑创建自己的 autorelease pool，来处理这些对象的内存管理。虽然 GCD dispatch queue 也有自己的 autorelease pool，但不保证在什么时候会回收这些 pool。

3.4. 创建和管理Dispatch Queue

获得全局并发Dispatch Queue

并发 dispatch queue 可以同时并行地执行多个任务，不过并发 queue 仍然按先进先出的顺序来启动任务，并发 queue 会在之前任务完成之前就出列下一个任务并启动执行。并发 queue 同时执行的任务数量会根据应用和系统动态变化，各种因素包括：可用核数量、其它进程正在执行的工作数量、其它串行 dispatch queue 中优先任务的数量等。

系统给每个应用提供三个并发 dispatch queue，所有应用全局共享，三个 queue 的区别是优先级。你不需要显式地创建这些 queue，使用 dispatch_get_global_queue 函数来获取这三个 queue：

```
dispatch_queue_t aQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

除了默认优先级的并发 queue，你还可以获得高和低优先级的两个，分别使用 DISPATCH_QUEUE_PRIORITY_HIGH 和 DISPATCH_QUEUE_PRIORITY_LOW 常量来调用上面函数。

虽然 dispatch queue 是引用计数的对象，但你不需要 retain 和 release 全局并发 queue。因为这些 queue 对应用是全局的，retain 和 release 调用会被忽略。

你也不需要存储这三个 queue 的引用，每次都直接调用 dispatch_get_global_queue 获得 queue 就行了。

创建串行Dispatch Queue

应用的任务需要按特定顺序执行时，就需要使用串行 Dispatch Queue，串行 queue 每次只能执行一个任务。你可以使用串行 queue 来替代锁，保护共享资源或可变的数据结构。和锁不一样的是，串行 queue 确保任务按可预测的顺序执行。而且只要你异步地提交任务到串行 queue，就永远不会产生死锁。

你必须显式地创建和管理所有你使用的串行 queue，应用可以创建任意数量的串行 queue，但不要为了同时执行更多任务而创建更多的串行 queue。如果你需要并发地执行大量任务，应该把任务提交到全局并发 Queue。

创建串行 queue 时，你需要明确自己的目的，如保护共享资源，或同步应用的某些关键行为。

dispatch_queue_create 函数创建串行 queue，两个参数分别是 queue 名和一组 queue 属性。调试器和性能工具会显示 queue 的名字，便于你跟踪任务的执行。

```
dispatch_queue_t queue;

queue = dispatch_queue_create("com.example.MyQueue", NULL);
```

运行时获得公共Queue

GCD 提供函数，让应用访问几个公共 dispatch queue：

- 使用 dispatch_get_current_queue 函数作为调试用途，或者测试当前 queue 的标识。在 block 对象中调用这个函数会返回 block 提交到的 queue（这个时候 queue 应该正在执行中）。在 block 对象之外调用这个函数会返回应用的默认并发 queue。
- 使用 dispatch_get_main_queue 函数获得应用主线程关联的串行 dispatch queue。Cocoa 应用、调用了 dispatch_main 函数或配置了 run loop（CFRunLoopRef 类型 或一个 NSRunLoop 对象）的应用，会自动创建这个 queue。
- 使用 dispatch_get_global_queue 来获得共享的并发 queue

Dispatch Queue的内存管理

Dispatch Queue 和其它 dispatch 对象都是引用计数的数据类型。当你创建一个串行 dispatch queue 时，初始引用计数为 1，你可以使用 dispatch_retain 和 dispatch_release 函数来增加和减少引用计数。当引用计数到达 0 时，系统会异步地销毁这个 queue。

对 dispatch 对象（如 queue）retain 和 release 是很重要的，确保它们被使用时能够保留在内存中。和内存托管的 Cocoa 对象一样，通用的规则是如果你使用一个传递给你代码中的 queue，你应该在使用前 retain，使用完之后 release。

你不需要 retain 或 release 全局 dispatch queue，包括全局并发 dispatch queue 和 main dispatch queue。

即使你实现的是自动垃圾收集的应用，也需要 retain 和 release 你的 dispatch queue 和其它 dispatch 对象。GCD 不支持垃圾收集模型来回收内存。

在Queue中存储自定义上下文信息

所有 dispatch 对象（包括 dispatch queue）都允许你关联 custom context data。使用 dispatch_set_context 和 dispatch_get_context 函数来设置和获取对象的上下文数据。系统不会使用你的上下文数据，所以需要你自己在适当的时候分配和销毁这些数据。

对于 Queue，你可以使用上下文数据来存储一个指针，指向 Objective-C 对象或其它数据结构，协助标识这个 queue 或代码的其它用途。你可以使用 queue 的 finalizer 函数来销毁（或解除关联）上下文数据。

为Queue提供一个清理函数

在创建串行 dispatch queue 之后，可以附加一个 finalizer 函数，在 queue 被销毁之前执行自定义的清理操作。使用 dispatch_set_finalizer_f 函数为 queue 指定一个清理函数，当 queue 的引用计数到达 0 时，就会执行该清理函数。你可以使用清理函数来解除 queue 关联的上下文数据，而且只有上下文指针不为 NULL 时才会调用这个清理函数。

下面例子演示了自定义 finalizer 函数的使用，你需要自己提供 myInitializeDataContextFunction 和 myCleanUpDataContextFunction 函数，用于初始化和清理上下文数据。

```
void myFinalizerFunction(void *context)

{

    MyDataContext* theData = (MyDataContext*)context;



    // Clean up the contents of the structure

    myCleanUpDataContextFunction(theData);
```

```
    // Now release the structure itself.

    free(theData);

}

dispatch_queue_t createMyQueue()

{

    MyDataContext*  data = (MyDataContext*)
malloc(sizeof(MyDataContext));

    myInitializeDataContextFunction(data);



    // Create the queue and set the context data.

    dispatch_queue_t serialQueue =
dispatch_queue_create("com.example.CriticalTaskQueue", NULL);

    if (serialQueue)

    {

        dispatch_set_context(serialQueue, data);

        dispatch_set_finalizer_f(serialQueue, &myFinalizerFunction);

    }

    return serialQueue;

}
```

3.5. 添加任务到Queue

要执行一个任务，你需要将它 dispatch 到一个适当的 dispatch queue，你可以同步或异步地 dispatch 一个任务，也可以单个或按组来 dispatch。一旦进入到 queue，queue 会负责尽快地执行你的任务。

添加单个任务到Queue

你可以异步或同步地添加一个任务到 Queue，尽可能地使用 dispatch_async 或 dispatch_async_f 函数异步地 dispatch 任务。因为添加任务到 Queue 中时，

无法确定这些代码什么时候能够执行。因此异步地添加 block 或函数，可以让你立即调度这些代码的执行，然后调用线程可以继续去做其它事情。

特别是应用主线程一定要异步地 dispatch 任务，这样才能及时地响应用户事件。

少数时候你可能希望同步地 dispatch 任务，以避免竞争条件或其它同步错误。使用 dispatch_sync 和 dispatch_sync_f 函数同步地添加任务到 Queue，这两个函数会阻塞，直到相应任务完成执行。

绝对不要在任务中调用 dispatch_sync 或 dispatch_sync_f 函数，并同步 dispatch 新任务到当前正在执行的 queue。对于串行 queue 这一点特别重要，因为这样做肯定会导致死锁；而并发 queue 也应该避免这样做。

```
dispatch_queue_t myCustomQueue;

myCustomQueue = dispatch_queue_create("com.example.MyCustomQueue", NULL);


dispatch_async(myCustomQueue, ^{

    printf("Do some work here.\n");

});


printf("The first block may or may not have run.\n");


dispatch_sync(myCustomQueue, ^{

    printf("Do some more work here.\n");

});

printf("Both blocks have completed.\n");
```

任务完成时执行Completion Block

dispatch 到 queue 中的任务，通常与创建任务的代码独立运行。在任务完成时，应用可能希望得到通知并使用任务完成的结果数据。在传统的异步编程模型中，你可能会使用回调机制，不过 dispatch queue 允许你使用 Completion Block。

Completion Block 是你 dispatch 到 queue 的另一段代码，在原始任务完成时自动执行。调用代码在启动任务时通过参数提供 Completion Block。任务代码只需要在完成工作时提交指定的 Block 或函数到指定的 queue。

下面代码使用 block 实现了平均数，最后两个参数允许调用方指定一个 queue 和报告结果的 block。在平均数函数完成计算后，会传递结果到指定的 block，并 dispatch 到指定的 queue。为了防止 queue 被过早地释放，必须首先 retain 这个 queue，然后在 dispatch 这个 Completion Block 之后，再 release 这个 queue。

```
void average_async(int *data, size_t len,
   dispatch_queue_t queue, void (^block)(int))
{
   // Retain the queue provided by the user to make
   // sure it does not disappear before the completion
   // block can be called.
   dispatch_retain(queue);

   // Do the work on the default concurrent queue and then
   // call the user-provided block with the results.
   dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFA
ULT, 0), ^{
      int avg = average(data, len);
      dispatch_async(queue, ^{ block(avg);});
```

```
      // Release the user-provided queue when done

      dispatch_release(queue);

   });

}
```

并发地执行Loop Iteration

如果你使用循环执行固定次数的迭代，并发 dispatch queue 可能会提高性能。例如下面 for 循环：

```
for (i = 0; i < count; i++) {

   printf("%u\n",i);

}
```

如果每次迭代执行的任务与其它迭代独立无关，而且循环迭代执行顺序也无关紧要的话，你可以调用 dispatch_apply 或 dispatch_apply_f 函数来替换循环。这两个函数为每次循环迭代将指定的 block 或函数提交到 queue。当 dispatch 到并发 queue 时，就有可能同时执行多个循环迭代。

调用 dispatch_apply 或 dispatch_apply_f 时你可以指定串行或并发 queue。并发 queue 允许同时执行多个循环迭代，而串行 queue 就没太大必要使用了。

和普通 for 循环一样，dispatch_apply 和 dispatch_apply_f 函数也是在所有迭代完成之后才会返回。因此在 queue 上下文执行的代码中再次调用这两个函数时，必须非常小心。如果你传递的参数是串行 queue，而且正是执行当前代码的 Queue，就会产生死锁。

另外这两个函数还会阻塞当前线程，因此在主线程中调用这两个函数同样必须小心，可能会阻止事件处理循环并无法响应用户事件。所以如果循环代码需要一定的时间执行，你可以考虑在另一个线程中调用这两个函数。

下面代码使用 dispatch_apply 替换了 for 循环，你传递的 block 必须包含一个参数，用来标识当前循环迭代。第一次迭代这个参数值为 0，第二次时为 1，最后一次值为 count - 1。

```
dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
```

```
dispatch_apply(count, queue, ^(size_t i) {
   printf("%u\n",i);
});
```

循环迭代执行的工作量需要仔细平衡，太多的话会降低响应性；太少则会影响整体性能，因为调度的开销大于实际执行代码。

在主线程中执行任务

GCD 提供一个特殊 dispatch queue，可以在应用的主线程中执行任务。应用主线程设置了 run loop（由 CFRunLoopRef 类型或 NSRunLoop 对象管理），就会自动创建这个 queue，并且自动 drain。非 Cocoa 应用如果不显式地设置 run loop，就必须显式地调用 dispatch_main 函数来显式地 drain 这个 dispatch queue。否则虽然你可以添加任务到 queue，但任务永远不会被执行。

调用 dispatch_get_main_queue 函数获得应用主线程的 dispatch queue。添加到这个 queue 的任务由主线程串行化执行，因此你可以在应用的某些地方使用这个 queue 作为同步点。

任务中使用Objective-C对象

GCD 支持 Cocoa 内存管理机制，因此可以在提交到 queue 的 block 中自由地使用 Objective-C 对象。每个 dispatch queue 维护自己的 autorelease pool 确保释放 autorelease 对象，但是 queue 不保证这些对象实际释放的时间。在自动垃圾收集的应用中，GCD 会在垃圾收集系统中注册自己创建的每个线程。

如果应用消耗大量内存，并且创建大量 autorelease 对象，你需要创建自己的 autorelease pool，用来及时地释放不再使用的对象。

3.6. 挂起和继续queue

我们可以暂停一个 queue 以阻止它执行 block 对象，使用 dispatch_suspend 函数挂起一个 dispatch queue；使用 dispatch_resume 函数继续 dispatch queue。调用 dispatch_suspend 会增加 queue 的引用计数，调用 dispatch_resume 则减少

queue 的引用计数。当引用计数大于 0 时，queue 就保持挂起状态。因此你必须对应地调用 suspend 和 resume 函数。

挂起和继续是异步的，而且只在执行 block 之间生效。挂起一个 queue 不会导致正在执行的 block 停止。

3.7. 使用Dispatch Semaphore控制有限资源的使用

如果提交到 dispatch queue 中的任务需要访问某些有限资源，可以使用 dispatch semaphore 来控制同时访问这个资源的任务数量。dispatch semaphore 和普通的信号量类似，唯一的区别是当资源可用时，需要更少的时间来获得 dispatch semaphore。

使用 dispatch semaphore 的过程如下：

1. 使用 dispatch_semaphore_create 函数创建 semaphore，指定正数值表示资源的可用数量。
2. 在每个任务中，调用 dispatch_semaphore_wait 来等待 Semaphore
3. 当上面调用返回时，获得资源并开始工作
4. 使用完资源后，调用 dispatch_semaphore_signal 函数释放和 signal 这个 semaphore

```
// Create the semaphore, specifying the initial pool size

dispatch_semaphore_t fd_sema = dispatch_semaphore_create(getdtablesize()
/ 2);



// Wait for a free file descriptor

dispatch_semaphore_wait(fd_sema, DISPATCH_TIME_FOREVER);

fd = open("/etc/services", O_RDONLY);



// Release the file descriptor when done

close(fd);

dispatch_semaphore_signal(fd_sema);
```

3.8. 等待queue中的一组任务

Dispatch group 用来阻塞一个线程，直到一个或多个任务完成执行。有时候你必须等待任务完成的结果，然后才能继续后面的处理。dispatch group 也可以替代线程 join。

基本的流程是设置一个组，dispatch 任务到 queue，然后等待结果。你需要使用 dispatch_group_async 函数，会关联任务到相关的组和 queue。使用 dispatch_group_wait 等待一组任务完成。

```
dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_group_t group = dispatch_group_create();


// Add a task to the group

dispatch_group_async(group, queue, ^{

   // Some asynchronous work

});


// Do some other work while the tasks execute.


// When you cannot make any more forward progress,

// wait on the group to block the current thread.

dispatch_group_wait(group, DISPATCH_TIME_FOREVER);


// Release the group when it is no longer needed.

dispatch_release(group);
```

3.9. Dispatch Queue和线程安全性

使用 Dispatch Queue 实现应用并发时，也需要注意线程安全性：

- Dispatch queue 本身是线程安全的。换句话说，你可以在应用的任意线程中提交任务到 dispatch queue，不需要使用锁或其它同步机制。
- 不要在执行任务代码中调用 dispatch_sync 函数调度相同的 queue，这样做会死锁这个 queue。如果你需要 dispatch 到当前 queue，需要使用 dispatch_async 函数异步调度
- 避免在提交到 dispatch queue 的任务中获得锁，虽然在任务中使用锁是安全的，但在请求锁时，如果锁不可用，可能会完全阻塞串行 queue。类似的，并发 queue 等待锁也可能阻止其它任务的执行。如果代码需要同步，就使用串行 dispatch queue。
- 虽然可以获得运行任务的底层线程的信息，最好不要这样做。

4. Dispatch Sources

4.1. 关于Dispatch Source

现代系统通常提供异步接口，允许应用向系统提交请求，然后在系统处理请求时应用可以继续处理自己的事情。GCD 正是基于这个基本行为而设计，允许你提交请求，并通过 block 和 dispatch queue 报告结果。

dispatch source 是基础数据类型，协调特定底层系统事件的处理。GCD 支持以下 dispatch source：

- **Timer dispatch source**：定期产生通知
- **Signal dispatch source**：UNIX 信号到达时产生通知
- **Descriptor dispatch source**：各种文件和 socket 操作的通知
    - 数据可读
    - 数据可写
    - 文件在文件系统中被删除、移动、重命名
    - 文件元数据信息改变
- **Process dispatch source**：进程相关的事件通知
    - 当进程退出时
    - 当进程发起 fork 或 exec 等调用
    - 信号被递送到进程
- **Mach port dispatch source**：Mach 相关事件的通知
- **Custom dispatch source**：你自己定义并自己触发

Dispatch source 替代了异步回调函数，来处理系统相关的事件。当你配置一个 dispatch source 时，你指定要监测的事件、dispatch queue、以及处理事件的代码（block 或函数）。当事件发生时，dispatch source 会提交你的 block 或函数到指定的 queue 去执行

和手工提交到 queue 的任务不同，dispatch source 为应用提供连续的事件源。除非你显式地取消，dispatch source 会一直保留与 dispatch queue 的关联。只要相应的事件发生，就会提交关联的代码到 dispatch queue 去执行。

为了防止事件积压到 dispatch queue，dispatch source 实现了事件合并机制。如果新事件在上一个事件处理器出列并执行之前到达，dispatch source 会将新旧事件的数据合并。根据事件类型的不同，合并操作可能会替换旧事件，或者更新旧事件的信息。

4.2. 创建Dispatch Source

创建 dispatch source 需要同时创建事件源和 dispatch source 本身。事件源是处理事件所需要的 native 数据结构，例如基于描述符的 dispatch source，你需要打开描述符；基于进程的事件，你需要获得目标程序的进程 ID。

然后可以如下创建相应的 dispatch source：

1. 使用 dispatch_source_create 函数创建 dispatch source
2. 配置 dispatch source：
    - 为 dispatch source 设置一个事件处理器
    - 对于定时器源，使用 dispatch_source_set_timer 函数设置定时器信息
3. 为 dispatch source 赋予一个取消处理器（可选）
4. 调用 dispatch_resume 函数开始处理事件

由于 dispatch source 必须进行额外的配置才能被使用，dispatch_source_create 函数返回的 dispatch source 将处于挂起状态。此时 dispatch source 会接收事件，但是不会进行处理。这时候你可以安装事件处理器，并执行额外的配置。

编写和安装一个事件处理器

你需要定义一个事件处理器来处理事件，可以是函数或 block 对象，并使用 dispatch_source_set_event_handler 或 dispatch_source_set_event_handler_f 安装事件处理器。事件到达时，dispatch source 会提交你的事件处理器到指定的 dispatch queue，由 queue 执行事件处理器。

事件处理器的代码负责处理所有到达的事件。如果事件处理器已经在 queue 中并等待处理已经到达的事件，如果此时又来了一个新事件，dispatch source 会合并这两个事件。事件处理器通常只能看到最新事件的信息，不过某些类型的 dispatch source 也能获得已经发生以及合并的事件信息。

如果事件处理器已经开始执行，一个或多个新事件到达，dispatch source 会保留这些事件，直到前面的事件处理器完成执行。然后以新事件再次提交处理器到 queue。

函数事件处理器有一个 context 指针指向 dispatch source 对象，没有返回值。Block 事件处理器没有参数，也没有返回值。

```
// Block-based event handler

void (^dispatch_block_t)(void)


// Function-based event handler

void (*dispatch_function_t)(void *)
```

在事件处理器中，你可以从 dispatch source 中获得事件的信息，函数处理器可以直接使用参数指针，Block 则必须自己捕获到 dispatch source 指针，一般 block 定义时会自动捕获到外部定义的所有变量。

```
dispatch_source_t source =
dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,

                                myDescriptor, 0, myQueue);

dispatch_source_set_event_handler(source, ^{

   // Get some data from the source variable, which is captured

   // from the parent context.

   size_t estimated = dispatch_source_get_data(source);


   // Continue reading the descriptor...

});
```

```
dispatch_resume(source);
```

Block 捕获外部变量允许更大的灵活性和动态性。当然，在 Block 中这些变量默认是只读的，虽然可以使用__block 来修改捕获的变量，但是你最好不要在事件处理器中这样做。因为 Dispatch source 异步执行事件处理器，当事件处理器修改原始外部变量时，有可能这些变量已经不存在了。

下面是事件处理器能够获得的事件信息：

函数	描述
dispatch_source_get_handle	这个函数返回 dispatch source 管理的底层系统数据类型。 对于描述符 dispatch source，函数返回一个 int，表示关联的描述符 对于信号 dispatch source，函数返回一个 int，表示最新事件的信号数值 对于进程 dispatch source，函数返回一个 pid_t 数据结构，表示被监控的进程 对于 Mach port dispatch source，函数返回一个 mach_port_t 数据结构 对于其它 dispatch source，函数返回的值未定义
dispatch_source_get_data	这个函数返回事件关联的所有未决数据。 对于从文件中读取数据的描述符 dispatch source，这个函数返回可以读取的字节数 对于向文件中写入数据的描述符 dispatch

	source，如果可以写入，则返回正数值 对于监控文件系统活动的描述符 dispatch source，函数返回一个常量，表示发生的事件类型，参考 dispatch_source_vnode_flags_t 枚举类型 对于进程 dispatch source，函数返回一个常量，表示发生的事件类型，参考 dispatch_source_proc_flags_t 枚举类型 对于 Mach port dispatch source，函数返回一个常量，表示发生的事件类型，参考 dispatch_source_machport_flags_t 枚举类型 对于自定义 dispatch source，函数返回从现有数据创建的新数据，以及传递给 dispatch_source_merge_data 函数的新数据。
dispatch_source_get_mask	这个函数返回用来创建 dispatch source 的事件标志 对于进程 dispatch source，函数返回 dispatch source 接收到的事件掩码，参考 dispatch_source_proc_flags_t 枚举类型 对于发送权利的 Mach port dispatch source，函数返回期望事件的掩码，参考 dispatch_source_mach_send_flags_t 枚举类型 对于自定义“或”的 dispatch source，函数返回用来合并数据值的掩码。

安装一个取消处理器

取消处理器在 dispatch soruce 释放之前执行清理工作。多数类型的 dispatch source 不需要取消处理器，除非你对 dispatch source 有自定义行为需要在释放时执行。但是使用描述符或 Mach port 的 dispatch source 必须设置取消处理器，用来关闭描述符或释放 Mach port。否则可能导致微妙的 bug，这些结构体会被系统其它部分或你的应用在不经意间重用。

你可以在任何时候安装取消处理器，但通常我们在创建 dispatch source 时就会安装取消处理器。使用 dispatch_source_set_cancel_handler 或 dispatch_source_set_cancel_handler_f 函数来设置取消处理器。

下面取消处理器关闭描述符：

```
dispatch_source_set_cancel_handler(mySource, ^{

   close(fd); // Close a file descriptor opened earlier.

});
```

修改目标Queue

在创建 dispatch source 时可以指定一个 queue，用来执行事件处理器和取消处理器。不过你也可以使用 dispatch_set_target_queue 函数在任何时候修改目标 queue。修改 queue 可以改变执行 dispatch source 事件的优先级。

修改 dispatch source 的目标 queue 是异步操作，dispatch source 会尽可能快地完成这个修改。如果事件处理器已经进入 queue 并等待处理，它会继续在原来的 Queue 中执行。随后到达的所有事件的处理器都会在后面修改的 queue 中执行。

关联自定义数据到dispatch source

和 GCD 的其它类型一样，你可以使用 dispatch_set_context 函数关联自定义数据到 dispatch source。使用 context 指针存储事件处理器需要的任何数据。如果你在 context 指针中存储了数据，你就应该安装一个取消处理器，在 dispatch source 不再需要时释放这些 context 自定义数据。

如果你使用 block 实现事件处理器，你也可以捕获本地变量，并在 Block 中使用。虽然这样也可以代替 context 指针，但是你应该明智地使用 Block 捕获变量。因为 dispatch source 长时间存在于应用中，Block 捕获指针变量时必须非常小心，因为指针指向的数据可能会被释放，因此需要复制数据或 retain。不管使用哪种方法，你都应该提供一个取消处理器，在最后释放这些数据。

Dispatch Source的内存管理

Dispatch Source 也是引用计数的数据类型，初始计数为 1，可以使用 dispatch_retain 和 dispatch_release 函数来增加和减少引用计数。引用计数到达 0 时，系统自动释放 dispatch source 数据结构。

dispatch source 的所有权可以由 dispatch source 内部或外部进行管理。外部所有权时，另一个对象拥有 dispatch source，并负责在不需要时释放它。内部所有权时，dispatch source 自己拥有自己，并负责在适当的时候释放自己。虽然外部所有权很常用，当你希望创建自主 dispatch source，并让它自己管理自己的行为时，可以使用内部所有权。例如 dispatch source 应用单一全局事件时，可以让它自己处理该事件，并立即退出。

4.3. Dispatch Source示例

创建一个定时器

定时器 dispatch source 定时产生事件，可以用来发起定时执行的任务，如游戏或其它图形应用，可以使用定时器来更新屏幕或动画。你也可以设置定时器，并在固定间隔事件中检查服务器的新信息。

所有定时器 dispatch source 都是间隔定时器，一旦创建，会按你指定的间隔定期递送事件。你需要为定时器 dispatch source 指定一个期望的定时器事件精度，也就是 leeway 值，让系统能够灵活地管理电源并唤醒内核。例如系统可以使用 leeway 值来提前或延迟触发定时器，使其更好地与其它系统事件结合。创建自己的定时器时，你应该尽量指定一个 leeway 值。

就算你指定 leeway 值为 0，也不要期望定时器能够按照精确的纳秒来触发事件。系统会尽可能地满足你的需求，但是无法保证完全精确的触发时间。

当计算机睡眠时，定时器 dispatch source 会被挂起，稍后系统唤醒时，定时器 dispatch source 也会自动唤醒。根据你提供的配置，暂停定时器可能会影响定

时器下一次的触发。如果定时器 dispatch source 使用 dispatch_time 函数或 DISPATCH_TIME_NOW 常量设置，定时器 dispatch source 会使用系统默认时钟来确定何时触发，但是默认时钟在计算机睡眠时不会继续。

如果你使用 dispatch_walltime 函数来设置定时器 dispatch source，则定时器会根据挂钟时间来跟踪，这种定时器比较适合触发间隔相对比较大的场合，可以防止定时器触发间隔出现太大的误差。

下面是定时器 dispatch source 的一个例子，每 30 秒触发一次，leeway 值为 1，因为间隔相对较大，使用 dispatch_walltime 来创建定时器。定时器会立即触发第一次，随后每 30 秒触发一次。 MyPeriodicTask 和 MyStoreTimer 是自定义函数，用于实现定时器的行为，并存储定时器到应用的数据结构。

```
dispatch_source_t CreateDispatchTimer(uint64_t interval,

             uint64_t leeway,

             dispatch_queue_t queue,

             dispatch_block_t block)

{

   dispatch_source_t timer =
dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER,

                                                    0, 0, queue);

   if (timer)

   {

      dispatch_source_set_timer(timer, dispatch_walltime(NULL, 0),
interval, leeway);

      dispatch_source_set_event_handler(timer, block);

      dispatch_resume(timer);

   }

   return timer;

}



void MyCreateTimer()
```

```
{
   dispatch_source_t aTimer = CreateDispatchTimer(30ull * NSEC_PER_SEC,
                              1ull * NSEC_PER_SEC,
                              dispatch_get_main_queue(),
                              ^{ MyPeriodicTask(); });

   // Store it somewhere for later use.
    if (aTimer)
    {
        MyStoreTimer(aTimer);
    }
}
```

虽然定时器 dispatch source 是接收时间事件的主要方法，你还可以使用其它选择。如果想在指定时间间隔后执行一个 block，可以使用 dispatch_after 或 dispatch_after_f 函数。这两个函数非常类似于 dispatch_async，但是只允许你指定一个时间值，时间一到就自动提交 block 到 queue 中执行，时间值可以指定为相对或绝对时间。

从描述符中读取数据

要从文件或 socket 中读取数据，需要打开文件或 socket，并创建一个 DISPATCH_SOURCE_TYPE_READ 类型的 dispatch source。你指定的事件处理器必须能够读取和处理描述符中的内容。对于文件，需要读取文件数据，并为应用创建适当的数据结构；对于网络 socket，需要处理最新接收到的网络数据。

读取数据时，你总是应该配置描述符使用非阻塞操作，虽然你可以使用 dispatch_source_get_data 函数查看当前有多少数据可读，但在你调用它和实际读取数据之间，可用的数据数量可能会发生变化。如果底层文件被截断，或发生网络错误，从描述符中读取会阻塞当前线程，停止在事件处理器中间并阻止 dispatch queue 去执行其它任务。对于串行 queue，这样还可能会死锁，即使是并发 queue，也会减少 queue 能够执行的任务数量。

下面例子配置 dispatch source 从文件中读取数据，事件处理器读取指定文件的全部内容到缓冲区，并调用一个自定义函数来处理这些数据。调用方可以使用返回的 dispatch source 在读取操作完成之后，来取消这个事件。为了确保 dispatch queue 不会阻塞，这里使用了 fcntl 函数，配置文件描述符执行非阻塞操作。dispatch source 安装了取消处理器，确保最后关闭了文件描述符。

```
dispatch_source_t ProcessContentsOfFile(const char* filename)

{

   // Prepare the file for reading.

   int fd = open(filename, O_RDONLY);

   if (fd == -1)

      return NULL;

   fcntl(fd, F_SETFL, O_NONBLOCK);  // Avoid blocking the read operation



   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

   dispatch_source_t readSource =
dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,

                                  fd, 0, queue);

   if (!readSource)

   {

      close(fd);

      return NULL;

   }



   // Install the event handler

   dispatch_source_set_event_handler(readSource, ^{

      size_t estimated = dispatch_source_get_data(readSource) + 1;

      // Read the data into a text buffer.
```

```
      char* buffer = (char*)malloc(estimated);
      if (buffer)
      {
         ssize_t actual = read(fd, buffer, (estimated));
         Boolean done = MyProcessFileData(buffer, actual);  // Process the
data.

         // Release the buffer when done.
         free(buffer);

         // If there is no more data, cancel the source.
         if (done)
            dispatch_source_cancel(readSource);
      }
    });

   // Install the cancellation handler
   dispatch_source_set_cancel_handler(readSource, ^{close(fd);});

   // Start reading the file.
   dispatch_resume(readSource);
   return readSource;
}
```

在这个例子中，自定义的 MyProcessFileData 函数确定读取到足够的数据，返回 YES 告诉 dispatch source 读取已经完成，可以取消任务。通常读取描述符的 dispatch source 在还有数据可读时，会重复调度事件处理器。如果 socket 连接关

闭或到达文件末尾，dispatch source 自动停止调度事件处理器。如果你自己确定不再需要 dispatch source，也可以手动取消它。

向描述符写入数据

向文件或 socket 写入数据非常类似于读取数据，配置描述符为写入操作后，创建一个 DISPATCH_SOURCE_TYPE_WRITE 类型的 dispatch source，创建好之后，系统会调用事件处理器，让它开始向文件或 socket 写入数据。当你完成写入后，使用 dispatch_source_cancel 函数取消 dispatch source。

写入数据也应该配置文件描述符使用非阻塞操作，虽然 dispatch_source_get_data 函数可以查看当前有多少可用写入空间，但这个值只是建议性的，而且在你执行写入操作时可能会发生变化。如果发生错误，写入数据到阻塞描述符，也会使事件处理器停止在执行中途，并阻止 dispatch queue 执行其它任务。串行 queue 会产生死锁，并发 queue 则会减少能够执行的任务数量。

下面是使用 dispatch source 写入数据到文件的例子，创建文件后，函数传递文件描述符到事件处理器。MyGetData 函数负责提供要写入的数据，在数据写入到文件之后，事件处理器取消 dispatch source，阻止再次调用。此时 dispatch source 的拥有者需负责释放 dispatch source。

```
dispatch_source_t WriteDataToFile(const char* filename)

{

    int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC,

                    (S_IRUSR | S_IWUSR | S_ISUID | S_ISGID));

    if (fd == -1)

        return NULL;

    fcntl(fd, F_SETFL); // Block during the write.



    dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

    dispatch_source_t writeSource =
dispatch_source_create(DISPATCH_SOURCE_TYPE_WRITE,

                           fd, 0, queue);
```

```
    if (!writeSource)
    {
        close(fd);
        return NULL;
    }

    dispatch_source_set_event_handler(writeSource, ^{
        size_t bufferSize = MyGetDataSize();
        void* buffer = malloc(bufferSize);

        size_t actual = MyGetData(buffer, bufferSize);
        write(fd, buffer, actual);

        free(buffer);

        // Cancel and release the dispatch source when done.
        dispatch_source_cancel(writeSource);
    });

    dispatch_source_set_cancel_handler(writeSource, ^{close(fd);});
    dispatch_resume(writeSource);
    return (writeSource);
}
```

监控文件系统对象

如果需要监控文件系统对象的变化，可以设置一个 DISPATCH_SOURCE_TYPE_VNODE 类型的 dispatch source，你可以从这个 dispatch source 中接收文件删除、写入、重命名等通知。你还可以得到文件的特定元数据信息变化通知。

在 dispatch source 正在处理事件时，dispatch source 中指定的文件描述符必须保持打开状态。

下面例子监控一个文件的文件名变化，并在文件名变化时执行一些操作（自定义的 MyUpdateFileName 函数）。由于文件描述符专门为 dispatch source 打开，dispatch source 安装了取消处理器来关闭文件描述符。这个例子中的文件描述符关联到底层的文件系统对象，因此同一个 dispatch source 可以用来检测多次文件名变化。

```
dispatch_source_t MonitorNameChangesToFile(const char* filename)

{

   int fd = open(filename, O_EVTONLY);

   if (fd == -1)

      return NULL;



   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

   dispatch_source_t source =
dispatch_source_create(DISPATCH_SOURCE_TYPE_VNODE,

               fd, DISPATCH_VNODE_RENAME, queue);

   if (source)

   {

      // Copy the filename for later use.

      int length = strlen(filename);
```

```
        char* newString = (char*)malloc(length + 1);
        newString = strcpy(newString, filename);
        dispatch_set_context(source, newString);

        // Install the event handler to process the name change
        dispatch_source_set_event_handler(source, ^{
            const char*  oldFilename =
(char*)dispatch_get_context(source);
            MyUpdateFileName(oldFilename, fd);
        });

        // Install a cancellation handler to free the descriptor
        // and the stored string.
        dispatch_source_set_cancel_handler(source, ^{
            char* fileStr = (char*)dispatch_get_context(source);
            free(fileStr);
            close(fd);
        });

        // Start processing events.
        dispatch_resume(source);
    }
    else
        close(fd);

    return source;
```

```
}
```

监测信号

应用可以接收许多不同类型的信号，如不可恢复的错误（非法指令）、或重要信息的通知（如子进程退出）。传统编程中，应用使用 sigaction 函数安装信号处理器函数，信号到达时同步处理信号。如果你只是想信号到达时得到通知，并不想实际地处理该信号，可以使用信号 dispatch source 来异步处理信号。

信号 dispatch source 不能替代 sigaction 函数提供的同步信号处理机制。同步信号处理器可以捕获一个信号，并阻止它中止应用。而信号 dispatch source 只允许你监测信号的到达。此外，你不能使用信号 dispatch source 获取所有类型的信号，如 SIGILL, SIGBUS, SIGSEGV 信号。

由于信号 dispatch source 在 dispatch queue 中异步执行，它没有同步信号处理器的一些限制。例如信号 dispatch source 的事件处理器可以调用任何函数。灵活性增大的代价是，信号到达和 dispatch source 事件处理器被调用的延迟可能会增大。

下面例子配置信号 dispatch source 来处理 SIGHUP 信号，事件处理器调用 MyProcessSIGHUP 函数，用来处理信号。

```
void InstallSignalHandler()

{

   // Make sure the signal does not terminate the application.

   signal(SIGHUP, SIG_IGN);



   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

   dispatch_source_t source =
dispatch_source_create(DISPATCH_SOURCE_TYPE_SIGNAL, SIGHUP, 0, queue);
```

```
   if (source)

   {

      dispatch_source_set_event_handler(source, ^{

         MyProcessSIGHUP();

      });


      // Start processing signals

      dispatch_resume(source);

   }

}
```

监控进程

进程 dispatch source 可以监控特定进程的行为，并适当地响应。父进程可以使用 dispatch source 来监控自己创建的所有子进程，例如监控子进程的死亡；类似地，子进程也可以使用 dispatch source 来监控父进程，例如在父进程退出时自己也退出。

下面例子安装了一个进程 dispatch source，监控父进程的终止。当父进程退出时，dispatch source 设置一些内部状态信息，告知子进程自己应该退出。MySetAppExitFlag 函数应该设置一个适当的标志，允许子进程终止。由于 dispatch source 自主运行，因此自己拥有自己，在程序关闭时会取消并释放自己。

```
void MonitorParentProcess()

{

   pid_t parentPID = getppid();



   dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

   dispatch_source_t source =
dispatch_source_create(DISPATCH_SOURCE_TYPE_PROC,
```

```
                                                    parentPID,
DISPATCH_PROC_EXIT, queue);
   if (source)
   {
      dispatch_source_set_event_handler(source, ^{
         MySetAppExitFlag();
         dispatch_source_cancel(source);
         dispatch_release(source);
      });
      dispatch_resume(source);
   }
}
```

4.4. 取消一个Dispatch Source

除非你显式地调用 dispatch_source_cancel 函数，dispatch source 将一直保持活动，取消一个 dispatch source 会停止递送新事件，并且不能撤销。因此你通常在取消 dispatch source 后立即释放它：

```
void RemoveDispatchSource(dispatch_source_t mySource)
{
   dispatch_source_cancel(mySource);
   dispatch_release(mySource);
}
```

取消一个 dispatch source 是异步操作，调用 dispatch_source_cancel 之后，不会再有新的事件被处理，但是正在被 dispatch source 处理的事件会继续被处理完成。在处理完最后的事件之后，dispatch source 会执行自己的取消处理器。

取消处理器是你最后的执行机会，在那里执行内存或资源的释放工作。例如描述符或 mach port 类型的 dispatch source，必须提供取消处理器，用来关闭描述符或 mach port

4.5. 挂起和继续Dispatch Source

你可以使用 dispatch_suspend 和 dispatch_resume 临时地挂起和继续 dispatch source 的事件递送。这两个函数分别增加和减少 dispatch 对象的挂起计数。因此，你必须每次 dispatch_suspend 调用之后，都需要相应的 dispatch_resume 才能继续事件递送。

挂起一个 dispatch source 期间，发生的任何事件都会被累积，直到 dispatch source 继续。但是不会递送所有事件，而是先合并到单一事件，然后再一次递送。例如你监控一个文件的文件名变化，就只会递送最后一次的变化事件。

5. Migrating Away from Threads

从现有的线程代码迁移到 GCD 和 Operation 对象有许多方法，尽管可能不是所有线程代码都能够执行迁移，但是迁移可能提升性能，并简化你的代码。

使用 dispatch queue 和 Operaiton queue 相比线程拥有许多优点：

- 应用不再需要存储线程栈到内存空间
- 消除了创建和配置线程的代码
- 消除了管理和调度线程工作的代码
- 简化了你要编写的代码

5.1.使用Dispatch Queue替代线程

首先考虑应用可能使用线程的几种方式：

- **单一任务线程**：创建一个线程执行单一任务，任务完成时释放线程
- **工作线程（Worker）**：创建一个或多个工作线程执行特定的任务，定期地分配任务给每个线程

- **线程池**：创建一个通用的线程池，并为每个线程设置 run loop，当你需要执行一个任务时，从池中抓取一个线程，并分配任务给它。如果没有空闲线程可用，任务进入等待队列。

虽然这些看上去是完全不同的技术，但实际上只是相同原理的变种。应用都是使用线程来执行某些任务，区别在于管理线程和任务排队的代码。使用 dispatch queue 和 operation queue，你可以消除所有线程、及线程通信的代码，集中精力编写处理任务的代码。

如果你使用了上面的线程模型，你应该已经非常了解应用需要执行的任务类型，只需要封装任务到 Operation 对象或 Block 对象，然后 dispatch 到适当的 queue，就一切搞定！

对于那些不使用锁的任务，你可以直接使用以下方法来进行迁移：

- 单一任务线程，封装任务到 block 或 operation 对象，并提交到并发 queue
- 工作线程，首先你需要确定使用串行 queue 还是并发 queue，如果工作线程需要同步特定任务的执行，就应该使用串行 queue。如果工作线程只是执行任意任务，任务之间并无关联，就应该使用并发 queue

- 线程池，封装任务到 block 或 operation 对象，并提交到并发 queue 中执行

当然，上面只是简单的情况。如果任务会争夺共享资源，理想的解决方案当然是消除或最小化共享资源的争夺。如果有办法重构代码，消除任务彼此对共享资源的依赖，这是最理想的。

如果做不到消除共享资源依赖，你仍然可以使用 queue，因为 queue 能够提供可预测的代码执行顺序。可预测意味着你不需要锁或其它重量级的同步机制，就可以实现代码的同步执行。

你可以使用 queue 来取代锁执行以下任务：

- 如果任务必须按特定顺序执行，提交到串行 dispatch queue；如果你想使用 Operation queue，就使用 Operation 对象依赖来确保这些对象的执行顺序。
- 如果你已经使用锁来保护共享资源，创建一个串行 queue 来执行任务并修改该资源。串行 queue 可以替换现有的锁，直接作为同步机制使用。

- 如果你使用线程 join 来等待后台任务完成，考虑使用 dispatch group；也可以使用一个 NSBlockOperation 对象，或者 Operation 对象依赖，同样可以达到 group-completion 的行为。
- 如果你使用“生产者-消费者”模型来管理有限资源池，考虑使用 dispatch queue 来简化“生产者-消费者”
- 如果你使用线程来读取和写入描述符，或者监控文件操作，使用 dispatch source

记住 queue 不是替代线程的万能药!queue 提供的异步编程模型适合于延迟无关紧要的场合。虽然 queue 提供配置任务执行优先级的方法，但更高的优先级也不能确保任务一定能在特定时间得到执行。因此线程仍然是实现最小延迟的适当选择，例如音频和视频 playback 等场合。

5.2.消除基于锁的代码

在线程代码中，锁是传统的多个线程之间同步资源的访问机制。但是锁的开销本身比较大，线程还需等待锁的释放。

使用 queue 替代基于锁的线程代码，消除了锁带来的开销，并且简化了代码编写。你可以将任务放到串行 queue，来控制任务对共享资源的访问。queue 的开销要远远小于锁，因为将任务放入 queue 不需要陷入内核来获得 mutex

将任务放入 queue 时，你做的主要决定是同步还是异步，异步提交任务到 queue 让当前线程继续运行；同步提交任务则阻塞当前线程，直到任务执行完成。两种机制各有各的用途，不过通常异步优先于同步。

实现异步锁

异步锁可以保护共享资源，而又不阻塞任何修改资源的代码。当代码的部分工作需要修改一个数据结构时，可以使用异步锁。使用传统的线程，你的实现方式是：获得共享资源的锁，做必要的修改，释放锁，继续任务的其它部分工作。但是使用 dispatch queue，调用代码可以异步修改，无需等待这些修改操作完成。

下面是异步锁实现的一个例子，受保护的资源定义了自己的串行 dispatch queue。调用代码提交一个 block 到这个 queue，在 block 中执行对资源的修改。由于 queue 串行的执行所有 block，对这个资源的修改可以确保按顺序进行；而且由于任务是异步执行的，调用线程不会阻塞。

```
dispatch_async(obj->serial_queue, ^{
   // Critical section
});
```

同步执行临界区

如果当前代码必须等到指定任务完成，你可以使用 dispatch_sync 函数同步的提交任务，这个函数将任务添加到 dispatch queue，并阻塞当前线程直到任务完成执行。dispatch queue 本身可以是串行或并发 queue，你可以根据具体的需要来选择使用。由于 dispatch_sync 函数会阻塞当前线程，你只应该在确实需要的时候才使用。

下面是使用 dispatch_sync 实现临界区的例子：

```
dispatch_sync(my_queue, ^{
   // Critical section
});
```

如果你已经使用串行 queue 保护一个共享资源，同步提交到串行 queue，并不能比异步提交提供更多的保护。同步提交的唯一理由是，阻止当前代码在临界区完成之前继续执行。如果当前代码不需要等待临界区完成，或者可以简单的提交接下来的任务到相同的串行 queue，就应该使用异步提交。

5.3.改进循环代码

如果循环每次迭代执行的工作互相独立，可以考虑使用 dispatch_apply 或 dispatch_apply_f 函数来重新实现循环。这两个函数将循环的每次迭代提交到 dispatch queue 进行处理。结合并发 queue 使用时，可以并发地执行迭代以提高性能。

dispatch_apply 和 dispatch_apply_f 是同步函数，会阻塞当前线程直到所有循环迭代执行完成。当提交到并发 queue 时，循环迭代的执行顺序是不确定的。因此你用来执行循环迭代的 Block 对象（或函数）必须可重入（reentrant）。

下面例子使用 dispatch 来替换循环，你传递给 dispatch_apply 或 dispatch_apply_f 的 Block 或函数必须有一个整数参数，用来标识当前的循环迭代：

```
queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_apply(count, queue, ^(size_t i) {
   printf("%u\n", i);
});
```

你需要明智地使用这项技术，因为 dispatch queue 的开销虽然非常小，但仍然存在，你的循环代码必须拥有足够的工作量，才能忽略掉 dispatch queue 的这些开销。

提升每次循环迭代工作量最简单的办法是 striding（跨步），重写 block 代码执行多个循环迭代。从而减少了 dispatch_apply 函数指定的 count 值。

```
int stride = 137;

dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);


dispatch_apply(count / stride, queue, ^(size_t idx){

    size_t j = idx * stride;

    size_t j_stop = j + stride;

    do {

       printf("%u\n", (unsigned int)j++);

    }while (j < j_stop);

});

// 执行剩余的循环迭代

size_t i;

for (i = count - (count % stride); i < count; i++)
```

```
printf("%u\n", (unsigned int)i);
```

如果循环迭代次数非常多，使用 stride 可以提升性能。

5.4. 替换线程Join

线程 join 允许你生成多个线程，然后让当前线程等待所有线程完成。线程创建子线程时指定为 joinable，如果父线程在子线程完成之前不能继续处理，就可以 join 子线程。join 会阻塞父线程直到子线程完成任务并退出，这时候父线程可以获得子线程的结果状态，并继续自己的工作。父线程可以一次性 join 多个子线程。

Dispatch Group 提供了类似于线程 join 的语义，但拥有更多优点。dispatch group 可以让线程阻塞直到一个或多个任务完成。和线程 join 不一样的是，dispatch goup 同时等待所有子任务完成。而且由于 dispatch group 使用 dispatch queue 来执行任务，更加高效。

以下步骤可以使用 dispatch group 替换线程 join：

1. 使用 dispatch_group_create 函数创建一个新的 dispatch group
2. 使用 dispatch_group_async 或 dispatch_group_async_f 函数添加任务到 Group，这些是你要等待完成的任务

3. 如果当前线程不能继续处理任何工作，调用 dispatch_group_wait 函数等待这个 group，会阻塞当前线程直到 group 中的所有任务执行完成。

如果你使用 Operation 对象来实现任务，可以使用依赖来实现线程 join。不过这时候不是让父线程等待所有任务完成，而是将父代码移到一个 Operation 对象，然后设置父 Operation 对象依赖于所有子 Operation 对象。这样父 Operation 对象就会等到所有子 Operation 执行完成后才开始执行。

5.5.修改“生产者-消费者”实现

生产者-消费者 模型可以管理有限数量动态生产的资源。生产者生成新资源，消费者等待并消耗这些资源。实现生产者-消费者模型的典型机制是条件或信号量。

使用条件（Condition）时，生产者线程通常如下：

1. 锁住与 condition 关联的 mutex（使用 pthread_mutex_lock）
2. 生产资源（或工作）
3. Signal 条件变量，通知有资源（或工作）可以消费（使用 pthread_cond_signal）
4. 解锁 mutex（使用 pthread_mutex_unlock）

对应的消费者线程则如下：

1. 锁住 condition 关联的 mutex（使用 pthread_mutex_lock）
2. 设置一个 while 循环
    1. 检查是否有资源（或工作）
    2. 如果没有资源（或工作），调用 pthread_cond_wait 阻塞当前线程，直到相应的 condition 触发
3. 获得生产者提供的资源（或工作）
4. 解锁 mutex（使用 pthread_mutex_unlock）
5. 处理资源（或工作）

使用 dispatch queue，你可以简化生产者-消费者为一个调用：

```
dispatch_async(queue, ^{
   // Process a work item.
});
```

当生产者有工作需要做时，只需要将工作添加到 queue，并让 queue 去处理该工作。唯一需要确定的是 queue 的类型，如果生产者生成的任

务需要按特定顺序执行，就使用串行 queue；否则使用并发 Queue，让系统尽可能多地同时执行任务。

5.6. 替换Semaphore代码

使用信号量可以限制对共享资源的访问，你应该考虑使用 dispatch semaphore 来替换普通信号量。传统的信号量需要陷入内核，而 dispatch semaphore 可以在用户空间快速地测试状态，只有测试失败调用线程需要阻塞时才会陷入内核。这样 dispatch semaphore 拥有比传统 semaphore 快得多的性能。两者的行为是一致的。

5.7. 替换Run-Loop代码

如果你使用 run loop 来管理一个或多个线程执行的工作，你会发现使用 queue 来实现和维护任务会简单许多。设置自定义 run loop 需要同时设置底层线程和 run loop 本身。run-loop 代码则需要设置一个或多个 run loop source，并编写回调来处理这些 source 事件到达。你可以创建一个串行 queue，并 dispatch 任务到 queue 中，这样一行代码就能够替换原有的 run-loop 创建代码：

```
dispatch_queue_t myNewRunLoop = dispatch_queue_create("com.apple.MyQueue",
NULL);
```

由于 queue 自动执行添加进来的任务，不需要编写额外的代码来管理 queue。你也不需要创建和配置线程，更不需要创建或附加任何 run-loop source。此外，你可以通过简单地添加任务就能让 queue 执行其它类型的任务，而 run loop 要实现这一点，必须修改现有 run loop source，或者创建一个新的 run loop source。

run loop 的一个常用配置是处理网络 socket 异步到达的数据，现在你可以附加 dispatch source 到需要的 queue 中，来实现这个行为。dispatch source 还能提供更多处理数据的选项，支持更多类型的系统事件处理。

5.8.与POSIX线程的兼容性

GCD 管理了任务和运行线程之间的关系，通常你应该避免在任务代码中使用 POSIX 线程函数，如果一定要使用，请小心。

应用不能删除或 mutate 不是自己创建的数据结构。使用 dispatch queue 执行的 block 对象不能调用以下函数：

```
pthread_detach
pthread_cancel
pthread_join
```

```
pthread_kill

pthread_exit
```

任务运行时修改线程状态是可以的，但你必须还原线程原来的状态。只要你记得还原线程的状态，下面函数是安全的：

```
pthread_setcancelstate

pthread_setcanceltype

pthread_setschedparam

pthread_sigmask

pthread_setspecific
```

特定 block 的执行线程可以在多次调用间会发生变化，因此应用不应该依赖于以下函数返回的信息：

```
pthread_self

pthread_getschedparam

pthread_get_stacksize_np

pthread_get_stackaddr_np

pthread_mach_thread_np

pthread_from_mach_thread_np

pthread_getspecific
```

Block 必须捕获和禁止任何语言级的异常，Block 执行期间的其它错误也应该由 block 处理，或者通知应用。