**什么是GCD?**

GCD全称为Grand Central Dispatch，是**libdispatch**的市场名称，而**libdispatch**是Apple的一个库，其为并发代码在iOS和OS X的多核硬件上执行提供支持。确切地说GCD是一套低层级的C API，通过 GCD，开发者只需要向队列中添加一段代码块(block或C函数指针)，而不需要直接和线程打交道。GCD在后端管理着一个线程池，它不仅决定着你的代码块将在哪个线程被执行，还根据可用的系统资源对这些线程进行管理。这样通过GCD来管理线程，从而解决线程被创建的问题。

**GCD的优势**

* **易用:** GCD 提供一个易于使用的并发模型而不仅仅只是锁和线程，以帮助我们避开并发陷阱,而且因为基于block，它能极为简单得在不同代码作用域之间传递上下文。
* **灵活:** GCD 具有在常见模式上(比如锁、单例)，用更高性能的方法优化代码，而且GCD能提供更多的控制权力以及大量的底层函数。
* **性能:** GCD能自动根据系统负载来增减线程数量，这就减少了上下文切换以及增加了计算效率。

**GCD相关概念**

如果要深入了解GCD，还有一些概念是需要知道的。

* **Dispatch Objects**   
  尽管GCD是纯C语言的，但它被组建成面向对象的风格。GCD对象被称为dispatch object, 所有的dispatch objects都是OC对象.，就如其他OC对象一样，当开启了ARC(automatic reference counting)时,dispatch objects的retain和release都会自动执行。而如果是MRC的话，dispatch objects会使用dispatch\_retain和dispatch\_release这两个方法来控制引用计数。
* **Serial & Concurrent**  
  串行任务就是每次只有一个任务被执行，并发任务就是在同一时间可以有多个任务被执行。
* **Synchronous & Asynchronous**  
  **同步函数**意思是在完成了它预定的任务后才返回，在任务执行时会阻塞当前线程。而**异步函数**则是任务会完成但不会等它完成，所以异步函数不会阻塞当前线程，会继续去执行下一个函数。
* **Concurrency & Parallelism**  
  **并发**的意思就是同时运行多个任务。这些任务可能是以在单核 CPU 上以分时（时间共享）的形式同时运行，也可能是在多核 CPU 上以真正的并行方式来运行。然后为了使单核设备也能实现这一点，并发任务必须先运行一个线程，执行一个上下文切换，然后运行另一个线程或进程。**并行**则是真正意思上的多任务同时运行。
* **Context Switch**  
  **Context Switch**即上下文切换，一个上下文切换指当你在单个进程里切换执行不同的线程时存储与恢复执行状态的过程。这个过程在编写多任务应用时很普遍，但会带来一些额外的开销。
* **Dispatch Queues**  
  GCD dispatch queues是一个强大的执行多任务的工具。Dispatch queue是一个对象，它可以接受任务，并将任务以先进先出(FIFO)的顺序来执行。Dispatch queue可以并发的或串行的执行任意一个代码块，而且并发任务会像NSOperationQueue那样基于系统负载来合适地并发进行，串行队列同一时间则只执行单一任务。Dispatch queues内部使用的是线程，GCD 管理这些线程，并且使用Dispatch queues的时候，我们都不需要自己创建线程。Dispatch queues相对于和线程直接通信的代码优势是：Dispatch queues使用起来特别方便，执行任务更加有效率。
* **Queue Types**  
  GCD有三种队列类型：

| **类型** | **描述** |
| --- | --- |
| **Serial** | **串行队列**将任务以先进先出(FIFO)的顺序来执行，所以串行队列经常用来做访问某些特定资源的同步处理。你可以也根据需要创建多个队列，而这些队列相对其他队列都是并发执行的。换句话说，如果你创建了4个串行队列，每一个队列在同一时间都只执行一个任务，对这四个任务来说，他们是相互独立且并发执行的。如果需要创建串行队列，一般用**dispatch\_queue\_create**这个方法来实现。 |
| **Concurrent** | **并发队列**虽然是能同时执行多个任务，但这些任务仍然是按照先到先执行(FIFO)的顺序来执行的。**并发队列**会基于系统负载来合适地选择并发执行这些任务。在iOS5之前，并发队列一般指的就是**全局队列(Global queue)**，进程中存在四个全局队列：高、中(默认)、低、后台四个优先级队列，可以调用dispatch\_get\_global\_queue函数传入优先级来访问队列。而在iOS5之后，我们也可以用**dispatch\_queue\_create**，并指定队列类型**DISPATCH\_QUEUE\_CONCURRENT**，来自己创建一个并发队列。 |
| **Main dispatch queue** | 与主线程功能相同。实际上，提交至**main queue**的任务会在主线程中执行。**main queue**可以调用**dispatch\_get\_main\_queue()**来获得。因为**main queue**是与主线程相关的，所以这是一个串行队列。和其它串行队列一样，这个队列中的任务一次只能执行一个。它能保证所有的任务都在主线程执行，而主线程是唯一可用于更新 UI 的线程。 |

**Getting Started!**

好了，在了解了上述概念后，我们可以正式投入GCD的怀抱了！

**创建和管理队列**

当你决定添加一些任务到队列中时，你需要决定该用那种类型的队列，并且抉择该如何使用他们。**Dispatch queues**可以串行或并发地执行这些任务，当你脑海里有一个大概的思路去如何使用队列时，你可以额快速地设置好队列的属性。接下来的部分，本文将告诉大家如何创建队列和怎样设置队列的属性。

1. **获取一个全局队列**  
   当我们需要同时执行多个任务时，并发队列是非常有用的。并发队列其实仍然还是一个队列，它保留了队列中的任务按先进先出(FIFO)的顺序执行的特点。同时，一个并发队列可以移除t它多余的任务，甚至这些任务之前还有未完成的任务。一个并发队列中实际执行的任务数是由很多因素决定的，比如系统的内核数，其他串行队列中任务的优先级，以及其他进程的工作状态。

系统为每个程序提供了四种全局队列，这些队列中仅仅通过优先级加以区别，这四种类型分别是高、中(默认)、低、后台。因为这些队列是全局的，所以大家不能直接创建它们，取而代之的是我们可以通过**dispatch\_get\_global\_queue**这个方法来调用它们。

**代码示例：**

dispatch\_queue\_t aQueue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

//全局队列的四种类型

DISPATCH\_QUEUE\_PRIORITY\_HIGH

DISPATCH\_QUEUE\_PRIORITY\_DEFAULT

DISPATCH\_QUEUE\_PRIORITY\_LOW

DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND

正如大家所看到的，因为存在队列的优先级，所以那些在高优先级队列中的任务会比在默认或低优先级队列中的任务要先执行，而默认级别队列的优先级又高于低优先级队列。注意，这里有一个比较特殊的级别容易被忽视，**DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND**。被设置成后台级别的队列，它会等待所有比它级别高的队列中的任务执行完或CPU空闲的时候才会执行自己的任务。例如磁盘的读写操作非常耗时，如果我们不需要立即获取到磁盘的数据，我们可以把读写任务放到后台队列中，这样读写任务只会在恰当的时候去执行而不会影响需要更改优先级的其他任务，整个程序也会更加有效率。

**Note:** 尽管dispatch queues是引用计数对象，但是我们不需要用retain和release来管理全局的并发队列。因为全局队列对于程序来说是全局的，retain和release会被全局队列忽略。所以，我们不需要存储这些队列的引用数，仅仅只需要在任何要使用它们的地方，调用**dispatch\_get\_global\_queue**这个方法即可。

1. **创建串行队列**&**并发队列**  
   当我们需要某些任务以指定的顺序去执行时，串行队列是一个非常好的选择。一个串行队列在同一时间里只会执行一个任务，而且每次都只会从队列的头部把任务取出来执行。正因为如此，我们可以用串行队列来替代锁的操作，比如数据资源的同步或修改数据结构时。和锁不同的是，串行队列能保证任务都是在可预见的顺序里执行，而且一旦我们在一个串行队列里异步提交了任务，队列就能永远不发生死锁。怎么样，是不是很棒，不过不像并发队列，这些串行队列是需要我们自己创建和管理的。

我们还可以在程序里创建任意数量的队列，不过值得注意的是，我们要尽量避免创建大量的串行队列而目的仅仅是为了同时执行队列中的这些任务。虽然GCD 通过创建所谓的线程池来大致匹配 CPU 内核数量，但是线程的创建并不是无代价的。每个线程都需要占用内存和内核资源。所以如果需要创建大量的并发任务，我们只需要把这些任务放到并发队列中即可。

**代码示例：**

//dispatch\_queue\_t

//dispatch\_queue\_create(const char \*label, dispatch\_queue\_attr\_t attr);

//串行队列

dispatch\_queue\_t serialQueue;

serialQueue = dispatch\_queue\_create("com.example.SerialQueue", NULL);

//并发队列

dispatch\_queue\_t concurrentQueue;

concurrentQueue = dispatch\_queue\_create("com.example.ConcurrentQueue", DISPATCH\_QUEUE\_CONCURRENT);

**NOTE:** *dispatch\_queue\_attr\_t*设置成NULL的时候默认代表串行。

1. **获取Main Queue**  
   获取主队列的方法很简单，如下所示：
2. dispatch\_queue\_t mainQueue;

mainQueue = dispatch\_get\_main\_queue();

1. **创建队列的自定义上下文**  
   所有的dispatch objects(包括dispatch queues)允许我们关联自定义的上下文。我们可以通过使用 **dispatch\_set\_context**和**dispatch\_get\_context**这两个方法，来为objects设定和获取这些上下文数据。因为系统不会使用我们自定义的数据，所以我们需要在适当的时候生成和销毁这些数据。对于队列，我们可以使用上下文来为一个OC对象或其他数据结构存储一个指针，以此来作为某个队列的唯一标识。我们可以在队列销毁前并在队列最后执行的方法中去销毁上下文数据。

**代码示例：**

void myFinalizerFunction(void \*context)

{

MyDataContext\* theData = (MyDataContext\*)context;

// 清除这个数据的内容

myCleanUpDataContextFunction(theData);

// 释放数据.

free(theData);

}

dispatch\_queue\_t createMyQueue()

{

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

myInitializeDataContextFunction(data);

// 创建队列并设置上下文.

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;

}

**添加任务到队列**

GCD有两种方式来把任务添加到队列中：异步和同步。一般情况下，使用**dispatch\_async**和**dispatch\_async\_f**来执行异步操作，是比同步操作更好的选择。当我们添加一个block对象或C函数到一个队列中后就立即返回了，任务会在之后由 GCD 决定执行，以及任务什么时候执行完我们是无法知道确定的。这样的好处是，如果我们需要在后台执行一个基于网络或 CPU 紧张的任务时就使用异步方法 ，这样就不会阻塞当前线程。

尽管一般情况下，我们会优先选择异步操作，但是在某些情况下，我们还是需要任务同步来执行。比如需要用同步操作来防止资源竞争或其他同步问题。这时，我们可以用 **dispatch\_sync**和**dispatch\_sync\_f**方法来把任务添加到队列中，这样被添加的任务会阻塞当前线程，直到这些任务执行完。

**代码示例：**

//代码示例：

//异步执行

dispatch\_queue\_t myCustomQueue;

myCustomQueue = dispatch\_queue\_create("com.example.MyCustomQueue", NULL);

dispatch\_async(myCustomQueue, ^{

NSLog("Do some work here.");

});

//同步执行

dispatch\_sync(myCustomQueue, ^{

NSLog("Do some more work here.");

});

**NOTE：**dispatch\_async在不同队列类型执行的情况

* 自定义串行队列：当你想串行执行后台任务并追踪它时就是一个好选择。这消除了资源争用，因为你知道一次只有一个任务在执行。
* 主队列：这是在一个并发队列上完成任务后更新 UI 的一般选择。
* 并发队列：这是在后台执行非 UI 工作的一般选择

**任务执行完后添加一个完成块(Completion Block)**

通常来说，我们把任务添加到队列后，一旦任务执行完，我们希望能得到通知并及时处理任务完成的结果。安装传统的异步开发流程，我们可以使用回调机制，或者在队列中使用**完成块(Completion Block)**。

一个**Completion Block**是在原任务完成后，我们给队列添加的一个代码块。回调代码的经典做法一般是在任务开始时，把**completion block**当成一个参数。需要我们做的只是把一个指定的block或函数，在指定的队列完成时，提交给这个队列即可。

下面是在一个计算平均值的函数，其利用了block方法来作为运算结果的回调。这个函数的最后参数**queue**、**block**，指定了一个queue和一个block，其在计算完数值结果后会把结果值传给这个block，然后再把block分发到这个队列(queue)中。注意，为了避免queue被提前释放掉了，我们可以在函数执行开始阶段为队列retain，然后在completion block完成后再release队列。

**代码示例：**

void average\_async(int \*data, size\_t len,

dispatch\_queue\_t queue, void (^block)(int))

{

// Retain the queue 以此确保在completion block

// 完成前不会被释放掉

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\_DEFAULT, 0),

^{

int avg = average(data, len);

dispatch\_async(queue, ^{ block(avg);});

// Release the queue

dispatch\_release(queue);

});

}

**并发执行迭代循环**

在开发中，并发队列能很好地提高效率，特别是当我们需要执行一个数据庞大的循环操作时。打个比方来说吧，我们需要执行一个for循环，每一次循环操作如下：

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

NSLog("%d",i);

}

GCD提供了一个简化方法叫做dispatch\_apply，当我们把这个方法放到并发队列中执行时，这个函数会调用单一block多次，并平行运算，然后等待所有运算结束。

**代码示例：**

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

dispatch\_apply(count, queue, ^(size\_t i) {

NSLog("%d",i);

});

怎么样，是不是很棒，但是需要异步怎么办？dispatch\_apply函数是没有异步版本的。解决的方法是只要用dispatch\_async函数将所有代码推到后台就行了。

**代码示例：**

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

dispatch\_async(queue, ^{

dispatch\_apply(count, queue, ^(size\_t i) {

NSLog("%d",i);

});

});

**挂起和恢复队列**

有时候，我们不想让队列中的某些任务马上执行，这时我们可以通过挂起操作来阻止一个队列中将要执行的任务。当需要挂起队列时，使用**dispatch\_suspend**方法；恢复队列时，使用**dispatch\_resume**方法。调用**dispatch\_suspend**会增加队列挂起的引用计数，而调用**dispatch\_resume**则会减少引用计数，当引用计数大于0时，队列会保持挂起状态。因此，这队列的挂起和恢复中，我们需要小心使用以避免引用计数计算错误的出现。

代码示例：

dispatch\_queue\_t myQueue;

myQueue = dispatch\_queue\_create("com.example.MyCustomQueue", NULL);

//挂起队列

dispatch\_suspend(myQueue);

//恢复队列

dispatch\_resume(myQueue);

**NOTE：**执行挂起操作不会对已经开始执行的任务起作用，它仅仅只会阻止将要进行但是还未开始的任务。

**使用Dispatch Semaphores**

信号量的作用是控制多个任务对有限数量资源的访问。一个**dispatch semaphore**就像一个普通信号的例外。当资源可用时，获取dispatch semaphore的时间比获取传统的系统信号量要更少。这是因为GCD不调用这个特殊情况下的内核。唯一的一次需要在内核中调用的情况是，当资源不可用且系统需要在停止你的线程直到获取信号。举例来说更容易理解，如果你创建了一个有着两个资源的信号量，那同时最多只能有两个线程可以访问临界区。其他想使用资源的线程必须在FIFO队列里等待。

常用的dispatch semaphore的语法：

1. 当创建信号量（使用**dispatch\_semaphore\_create**方法），我们可以指定一个正整数，表示可用资源的数量。
2. 在每一个任务里，调用**dispatch\_semaphore\_wait**来等待信号量。
3. 当等待调用返回时，获取资源并做自己的工作。
4. 当我们用到资源后，释放掉它，然后通过调用**dispatch\_semaphore\_signal**方法来发出信号。

每一个应用都提供了有限的文件描述符来使用，如果我们需要处理一大堆的文件时，我们不想在运行文件描述符的时候同时打开很多文件。取而代之的是，我们可以用信号量来限制同一时间里文件描述符的数量。下面就是为了实现此需求的简单代码：

// 创建一个信号量

dispatch\_semaphore\_t fd\_sema = dispatch\_semaphore\_create(getdtablesize() / 2);

// 等待一个空闲的文件描述符

dispatch\_semaphore\_wait(fd\_sema, DISPATCH\_TIME\_FOREVER);

fd = open("/etc/services", O\_RDONLY);

// 当完成时，释放掉文件描述符

close(fd);

dispatch\_semaphore\_signal(fd\_sema);

**Dispatch Groups的使用**

**Dispatch groups**是阻塞线程直到一个或多个任务完成的一种方式。在那些需要等待任务完成才能执行某个处理的时候，你可以使用这个方法。**Dispatch Group**会在整个组的任务都完成时通知你，这些任务可以是同步的，也可以是异步的，即便在不同的队列也行。而且在整个组的任务都完成时，**Dispatch Group**可以用同步的或者异步的方式通知你。当group中所有的任务都完成时，GCD 提供了两种通知方式。

1. **dispatch\_group\_wait**。它会阻塞当前线程，直到组里面所有的任务都完成或者等到某个超时发生。

代码示例：

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

dispatch\_group\_t group = dispatch\_group\_create();

// 添加队列到组中

dispatch\_group\_async(group, queue, ^{

// 一些异步操作

});

//如果在所有任务完成前超时了，该函数会返回一个非零值。

//你可以对此返回值做条件判断以确定是否超出等待周期；

dispatch\_group\_wait(group, DISPATCH\_TIME\_FOREVER);

// 不需要group后将做释放操作

dispatch\_release(group);

1. **dispatch\_group\_notify**。它以异步的方式工作，当 Dispatch Group中没有任何任务时，它就会执行其代码，那么 completionBlock便会运行。

代码示例：

dispatch\_queue\_t queue = dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

dispatch\_group\_t group = dispatch\_group\_create();

// 添加队列到组中

dispatch\_group\_async(group, queue, ^{

// 一些异步操作

});

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

if (completionBlock) { completionBlock(error); }

});