GCD,NSOperation,NSThread

参考
在 iOS 编程中, 主要通过这三种方式来实现多线程方案. 一般在开发中使用最广泛的还是 GCD, 其是 Apple 基于 C 实现的框架, 执行效率也是相对高的. 比较推荐使用 GCD.

基本概念

- 进程: 一个具有独立运行功能的程序关于某个数据集合的运动, 可以理解成一个运行中的程序。
- 线程: 程序执行流的最小单位,线程是进程中的一个实体。
- 同步: 在当前线程中,不开辟新的线程的情况下, 按顺序执行任务。
- 异步: 在当前线程中开辟多个线程,并且不按顺序执行任务。
- 队列: 装载线程任务的队列结构。
- 串行: 线程执行只能依次逐一先后有序的执行。
- 并行: 线程执行可以同时一起执行。
- 注意: • 一个进程可有多个线程。 • 一个进程可有多个队列。 • 队列可分并发队列和串行队列。

二.iOS多线程对比

NSThread

- 优点:
- 缺点:

NSOperation

- 优点:
- 缺点:

GCD

- 优点: 最高效，避开并发陷阱。
- 缺点: 基于C实现。

使用API

NSThread

1. 动态实例化

```
NSThread *thread = [[NSThread alloc] initWithTarget:self selector:@sele
ctor(loadImageSource:) object:imgUrl];
thread.threadPriority = 1; // 设置线程的优先级(0.0 - 1.0, 1.0最高级)
[thread start];
```

1. 静态实例化
   ```
   [NSThread detachNewThreadSelector:@selector(loadImageSource:)
   toTarget:self withObject:imgUrl];
   ```
2. 隐式实例化
   ```
   [self performSelectorInBackground:@selector(loadImageSource:)
   withObject:imgUrl];
   ```
   ###NSOperation

主要通过 NSOperation 和 NSOperationQueue 实现多线程。

1. 初始化 NSOperation 子类. alloc initWithTarget 或者 blockOperationWithBlock· 其代表的是一个线程任务.
2. 可以直接使用创建的 NSOperation start 运行。 直接运行的话相当于在当前线程直接同步操作. 不过一般都会在创建一个 NSOperationQueue 队列, 用来装载线程任务, 队列分串行并行.
3. 使用队列 通过 addOperation: 添加线程任务, 来完成 队列装载线程任务, 执行完之后相当于 使用此队列执行相应线程操作。

一般分三种使用 NSOperation 的方法;
1. 使用其子类 NSInvocationOperation 通过动态初始化方法或者也可以直接将 NSInvocation 的方式创建.
2. 使用其子类 NSBlockOperation 通过静态初始化方法, blockOperationWithBlock 添加 block 代码块的方式来创建.
3. 自定义 NSOperation 子类来完成, 自定义子类来完成的方法会相当灵活一点, 我们可以在子类中添加代理, 用代理模式来实现. 在子类模块中添加一个 @protocal 写好相应方法声明, 每次具体实现都可以放在具体想要使用多线程的实例类里去应用即可。 不过有一个特别的地方是, 当执行时,会自动调用 main 函数, 我们可以通过 重写的方法。

总结: 三种创建线程任务的方式,都是通过对其 NSOperation 子类的创建, 然后我们可以通过 main 或者 start 方式来执行, 或者通过加入相应队列的方式来执行。 添加到队列

之后会自动执行 start 或者 main。 我们创建子类的时候 可以通过重写这个方法来进行一下我们想要执行的任务。

GCD

GCD 是 Apple 开发,基于 C 语言编写的一个多线程解决方案, 其也采用了 NSOperation 队列和线程任务的思想来执行多线程任务。

分发队列种类(dispatch queue)

* UI主线程队列 main queue 通过 dispatch_get_main_queue() 主线程串行队列

* 并行队列global dispatch queue

dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)

参数1 是一个枚举, 表示优先级.

参数2 未使用到, 一般默认填0即可.

* 串行队列serial queues dispatch_queue_create("minggo.app.com", NULL);

6种多线程实现

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    [self loadImageSource:imgUrl1];
});
// 通过异步创建子线程, 然后加入全局队列实现后台实现图片下载的方式

dispatch_async(dispatch_get_main_queue(), ^{
    [self loadImageSource:imgUrl1];
}); // 通过异步创建子线程, 然后添加主线程队列实现

static dispatch_once_t onceToken;
dispatch_once(&onceToken, ^{
    [self loadImageSource:imgUrl1];
}); 在写单利的时候用到 只会执行一次的线程.

dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR
ITY_DEFAULT, 0); size_t count = 10; dispatch_apply(count, queue, ^(size
_t i) { NSLog(@"循环执行第%li次",i); [self loadImageSource:imgUrl1]; });
  // 通过异步执行多个在全局线程的任务.

double delayInSeconds = 2.0; dispatch_time_t popTime = dispatch_time(DI
SPATCH_TIME_NOW, delayInSeconds * NSEC_PER_SEC); dispatch_after(popTime
, dispatch_get_main_queue(), ^(void){ [self loadImageSource:imgUrl1]; }
);
// 延迟执行.

dispatch_queue_t urls_queue = dispatch_queue_create("minggo.app.com", N
ULL); dispatch_async(urls_queue, ^{ [self loadImageSource:imgUrl1]; });
// 通过异步执行自定义队列。
```