ByteDance 字节跳动

Objective-C 多线程

session 2

张宇 抖音iOS开发工程师

基本概念

进程、线程

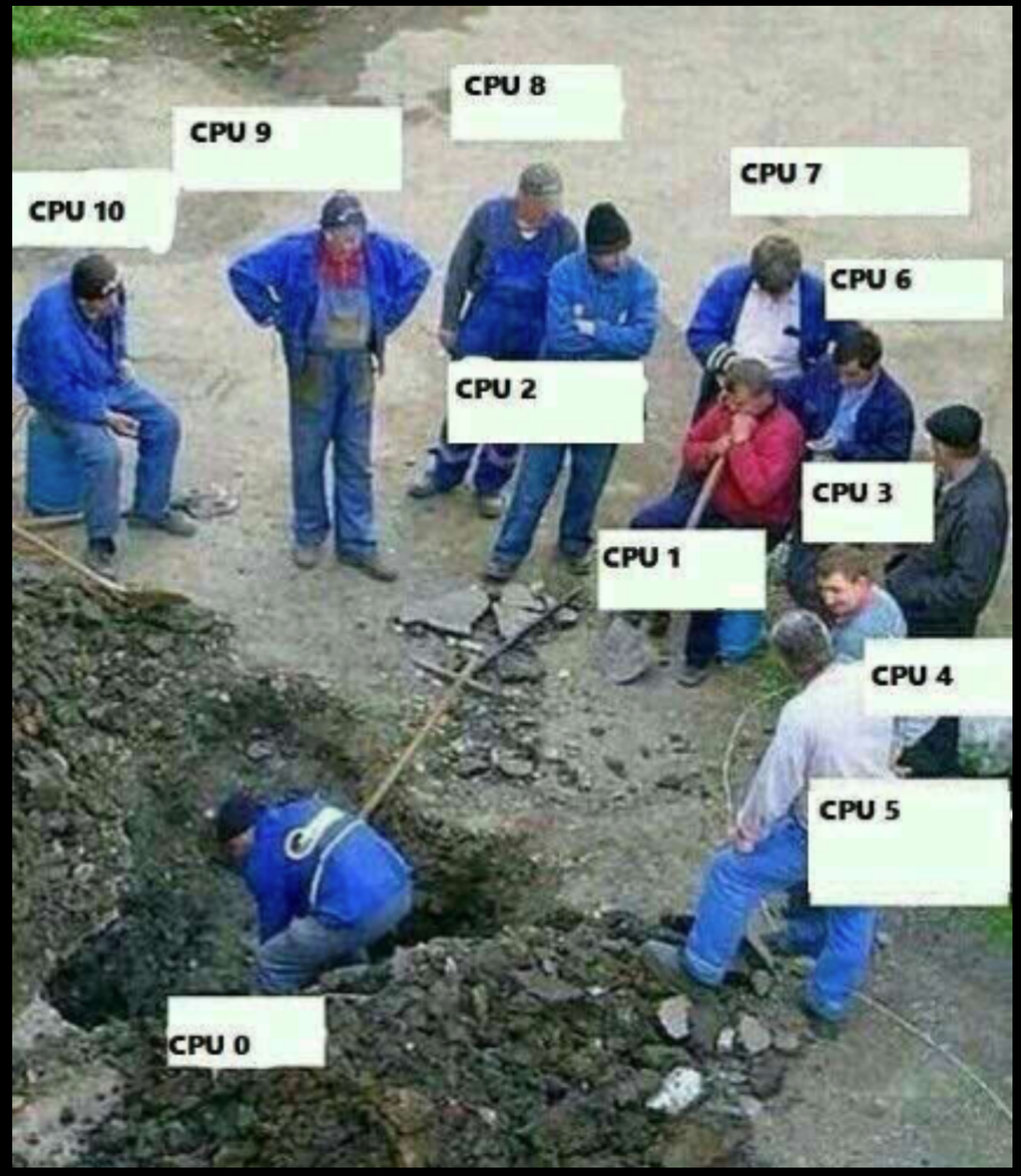
进程：操作系统中运行的一个应用程序，每个进程之间是独立的，每个进程均运行在其专用受保护的内存空间内

线程：是进程中的一个实例，是系统实施调度的独立单位

多线程

手机多核是趋势，iPhone XR

一核有难、九核围观



并发与并行

并发指的是一种现象，一种经常出现，无可避免的现象。它描述的是“多个任务同时发生，需要被处理”这一现象。它的侧重点在于“发生”。

并行指的是一种技术，一个同时处理多个任务的技术。它描述了一种能够同时处理多个任务的能力，侧重点在于“运行”。

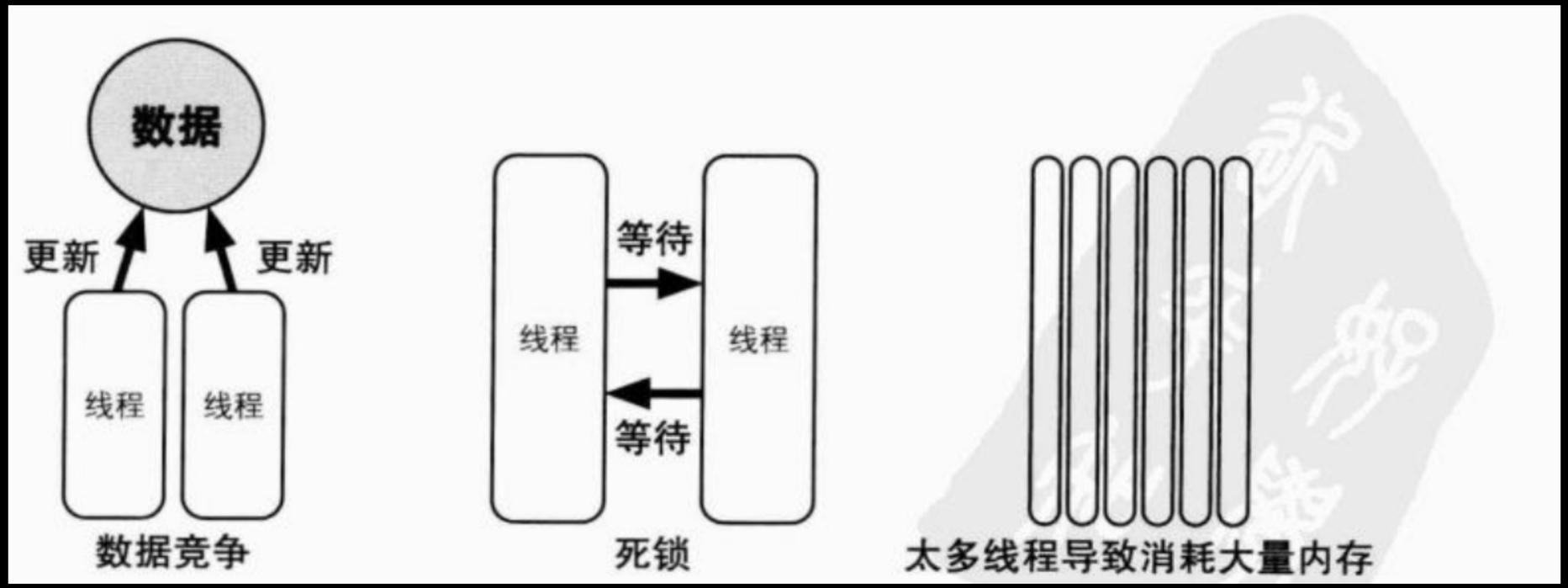
并行的反义词就是串行，表示任务必须按顺序来，一个一个执行，前一个执行完了才能执行后一个。

我们经常挂在嘴边的“多线程”，正是采用了并行技术，从而提高了执行效率。因为有多个线程，所以计算机的多个CPU可以同时工作，同时处理不同线程内的指令。

多线程

一个进程中可以开启多个线程，多个线程可以并行（同时）执行不同的任务，多线程

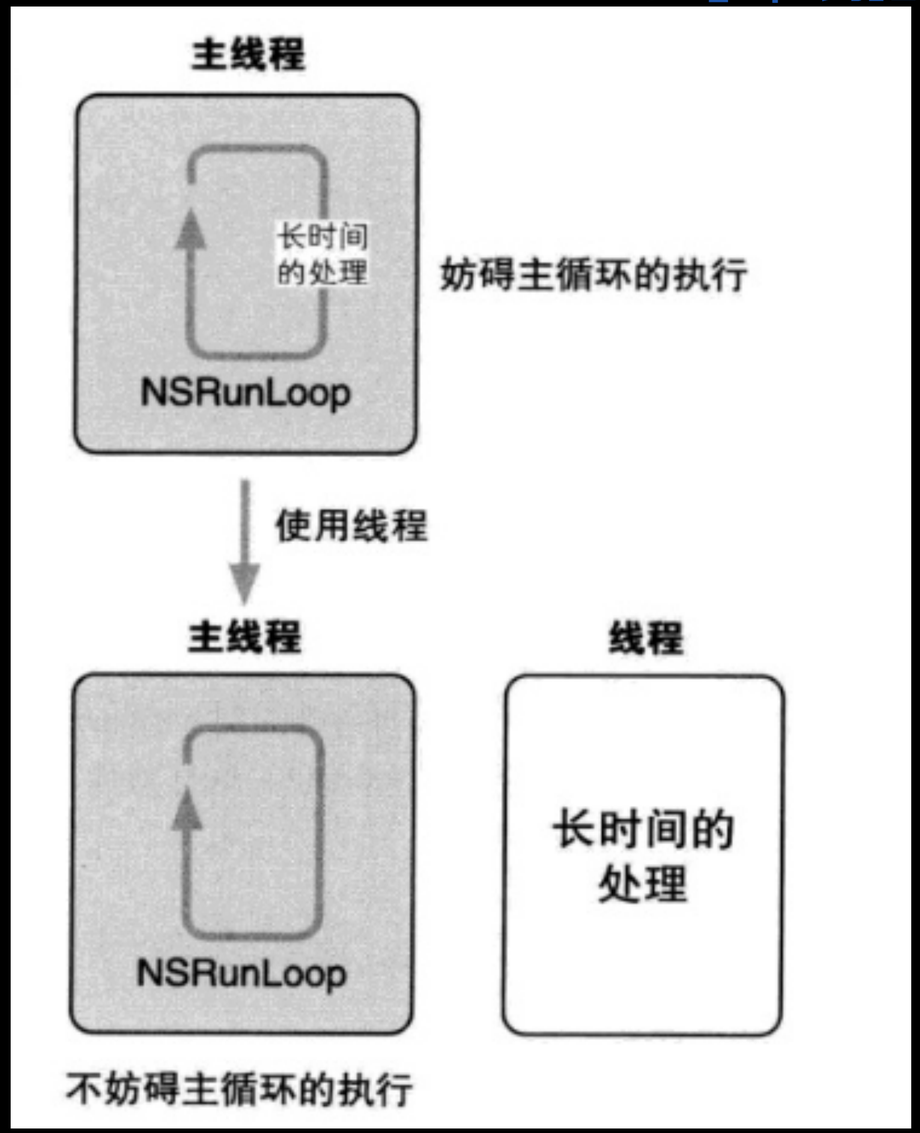
并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）



主线程

iOS App运行后，默认会开启1条线程，称为“主线程”或“UI线程”

主线程处理UI事件（比如点、滑动、拖拽等等）和显示、刷新UI界面



进程、线程

ByteDance 字节跳动

技术方案	简介	语言	线程生命周期	使用频率
pthread	▪ 一套通用的多线程API ▪ 适用于Unix\Linux\Windows等系统 ▪ 跨平台\可移植 ▪ 使用难度大	C	程序员管理	几乎不用
NSThread	▪ 使用更加面向对象 ▪ 简单易用，可直接操作线程对象	OC	程序员管理	偶尔使用
GCD	▪ 旨在替代NSThread等线程技术 ▪ 充分利用设备的多核	C	自动管理	经常使用
NSOperation	▪ 基于GCD（底层是GCD） ▪ 比GCD多了一些更简单实用的功能 ▪ 使用更加面向对象	OC	自动管理	经常使用

多线程优缺点

ByteDance 字节跳动

优点：

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

缺点：

创建线程是有开销的，iOS下主要成本包括：内核数据结构（大约1KB）、栈空间、创建时间90毫秒

如果开启大量的线程，会降低程序的性能

线程越多，CPU在调度线程上的开销越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

NSThread

NSThread - 介绍

NSThread是经过Apple封装的面向对象的，它允许开发者直接以面向对象的思想对线程进行操作，每一个NSThread对象就代表一条线程，但是开发者必须手动管理线程的生命周期，这点是Apple 不提倡的

NSThread - 常见方法 - 其他方法


```
NSThread *thread = [[NSThread alloc] initWithTarget:self selector:@selector(run)
object:nil];
[thread start];
```

NSThread - 常见方法 - 创建、启动线程


```
[NSThread mainThread]; // 获取主线程
[NSThread currentThread]; // 获取当前线程
[NSThread exit]; // 退出线程
[thread cancel]; // 取消线程
[NSThread isMainThread]; // 判断是否为多线程
```

NSThread - 其他创建方式

ByteDance 字节跳动

```
[NSThread detachNewThreadSelector:@selector(run) toTarget:self withObject:nil];
[self performSelectorInBackground:@selector(run) withObject:nil];
```

NSOperation

NSOperation - 介绍

NSOperation、NSOperationQueue 是苹果提供给我们的一套多线程解决方案。实际上 NSOperation、NSOperationQueue 是基于 GCD 更高一层的封装，完全面向对象。但是比 GCD 更简单易用、代码可读性也更高。

为什么要使用 NSOperation、NSOperationQueue?

可添加完成的代码块，在操作完成后执行。

添加操作之间的依赖关系，方便的控制执行顺序。

设定操作执行的优先级。

可以很方便的取消一个操作的执行。

使用 KVO 观察对操作执行状态的更改：isExecuteing、isFinished、isCancelled。

GCD

GCD - 介绍

ByteDance 字节跳动

Grand Central Dispatch(GCD)

GCD 可用于多核的并行运算

GCD 会自动利用更多的 CPU 内核（比如双核、四核）

GCD 会自动管理线程的生命周期（创建线程、调度任务、销毁线程）

程序员只需要告诉 GCD 想要执行什么任务，不需要编写任何线程管理代码

GCD - 同步

同步添加任务到指定的队列中，在添加的任务执行结束之前，会一直等待，直到队列里面的任务完成之后再继续执行。

只能在当前线程中执行任务，不具备开启新线程的能力

```
dispatch_queue_t queue = dispatch_get_main_queue();
dispatch_async(queue, ^{
    // 想执行的任务
});
```

GCD - 异步

异步添加任务到指定的队列中，它不会做任何等待，可以继续执行任务。

可以在新的线程中执行任务，具备开启新线程的能力。

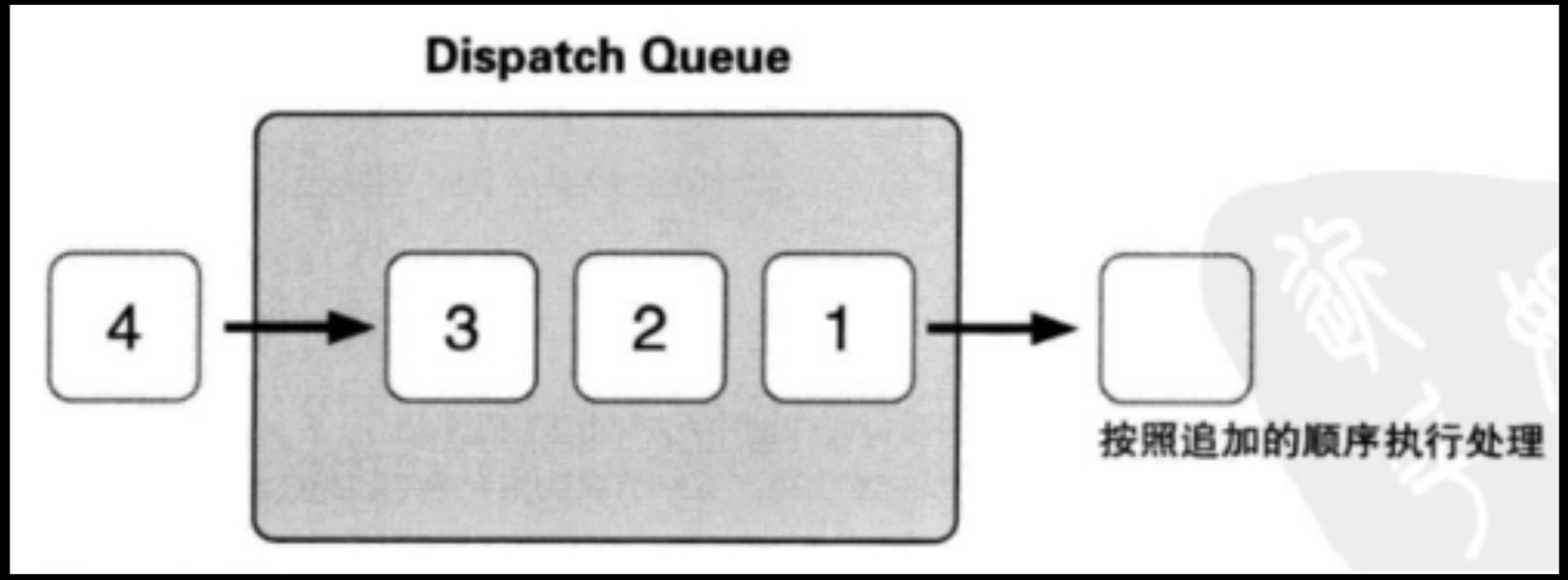
```
dispatch_queue_t queue = dispatch_get_main_queue();
dispatch_async(queue, ^{
    // 想执行的任务
});
```

GCD - 队列

Dispatch Queue按照追加的顺序(FIFO)执行处理

Dispatch Queue分为4种队列：

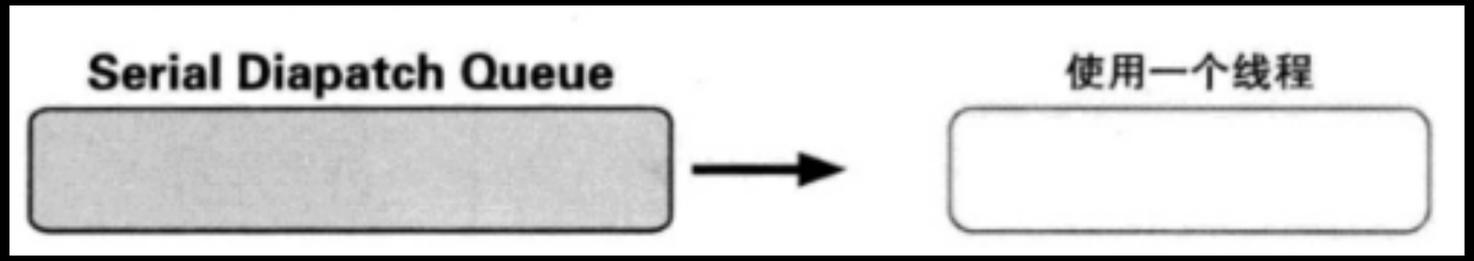
Serial Queue(串行队列)、Concurrent Queue(并发队列)、Main Dispatch Queue(主调度队列)、Global Dispatch Queue(全局并发队列)



多线程 - 串行队列

串行队列（也称为私有调度队列）按顺序将其中一个任务添加到队列中，并且一次只执行一个任务

如果创建四个串行队列，每个队列一次只执行一个任务，但最多四个任务可以并发执行，每个队列中有一个任务



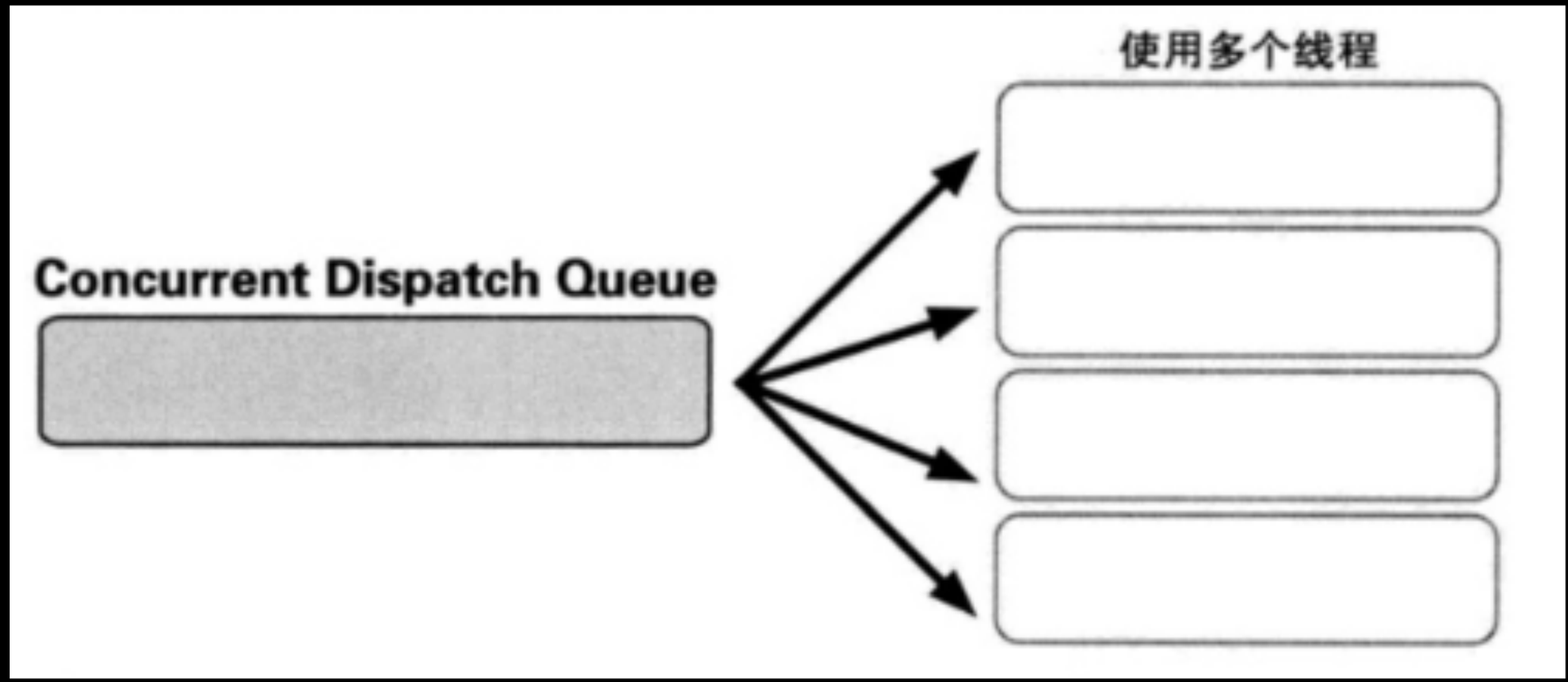
多线程 - 串行队列(Serial Queue)

ByteDance 字节跳动

```
dispatch_queue_t queue = dispatch_get_main_queue();

dispatch_async(queue, ^{
    NSLog(@"thread1");
});

dispatch_async(queue, ^{
    NSLog(@"thread2");
});

dispatch_async(queue, ^{
    NSLog(@"thread3");
});
```

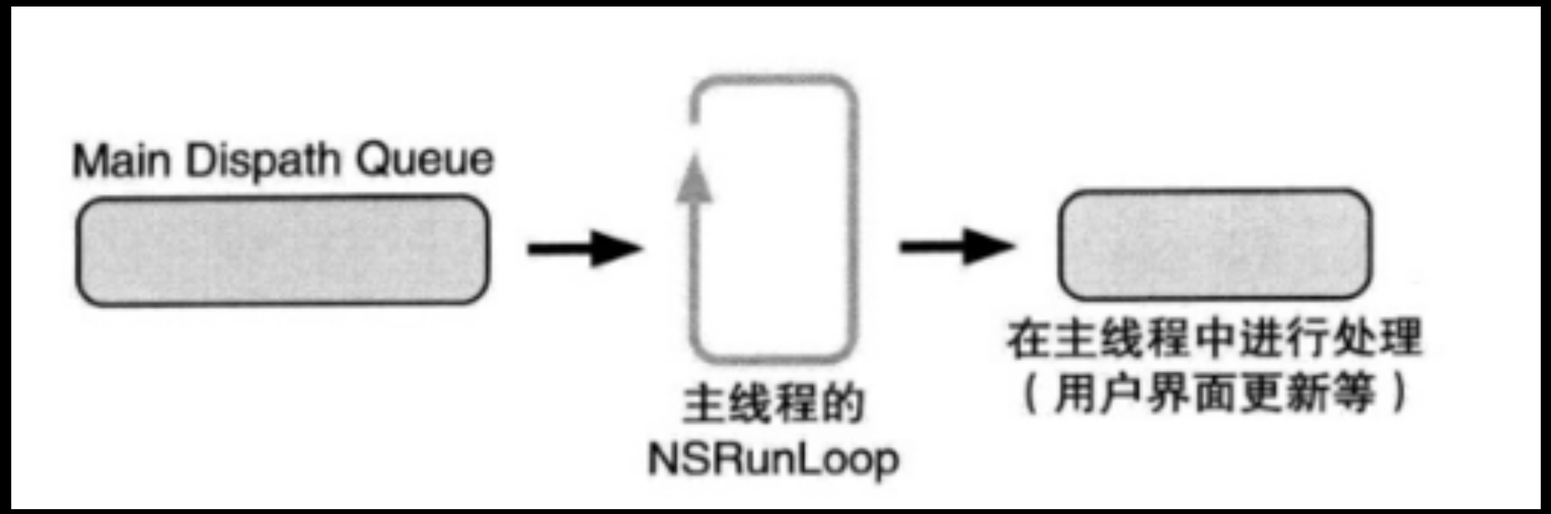
GCD - 并行队列(Concurrent Queue)

并发队列（也称为全局调度队列）同时执行一个或多个任务，但任务仍然按照它们添加到队列的顺序执行.当前执行的任务运行在不同线程上,而这些线程由调度队列所管理.



GCD - 并行队列

```
dispatch_queue_t queue = dispatch_queue_create("myConcurrentDispatchQueue",
DISPATCH_QUEUE_CONCURRENT);
```



GCD - 并行队列

ByteDance 字节跳动

四个全局默认并发队列：用dispatch_get_global_queue函数的获取其中一个队列

名称	Dispatch Queue的种类	说明
Main Dispatch Queue	Serial Diapatch Queue	主线程执行
Global Dispatch Queue(High Priority)	Concurrent Dispatch Queue	执行优先级:高(最高优先级)
Global Dispatch Queue(Default Priority)	Concurrent Dispatch Queue	执行优先级:默认
Global Dispatch Queue(Low Priority)	Concurrent Dispatch Queue	执行优先级:低
Global Dispatch Queue(Background Priority)	Concurrent Dispatch Queue	执行优先级:后台

GCD - 并行队列

ByteDance 字节跳动

```
dispatch_queue_t queue = dispatch_queue_create("myConcurrentDispatchQueue",
DISPATCH_QUEUE_CONCURRENT);

dispatch_async(queue, ^{
    // 想执行的任务
})
dispatch_async(queue, ^{
    // 想执行的任务
})
dispatch_async(queue, ^{
    // 想执行的任务
})
```

GCD - 并行队列

ByteDance 字节跳动

```
// 高优先级全局并发队列
dispatch_queue_t globalDiapatchQueueHigh =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);

// 默认先级全局并发队列
dispatch_queue_t globalDiapatchQueueDefault =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

// 低优先级全局并发队列
dispatch_queue_t globalDiapatchQueueLow =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_LOW, 0);

// 后台先级全局并发队列
dispatch_queue_t globalDiapatchQueueBackground =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0);
```

GCD - 主线程队列

ByteDance 字节跳动

```
// 主队列
dispatch_queue_t mainDiapatchQueue = dispatch_get_main_queue();
```

GCD - dispatch_queue_create

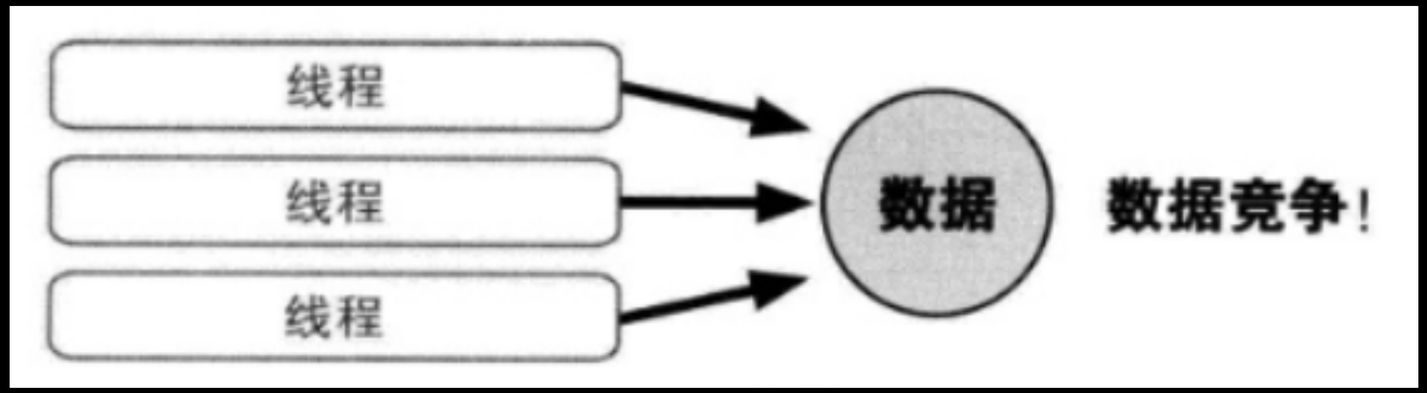

```
// 生成串行队列
dispatch_queue_t mySerialDiapatchQueue =
dispatch_queue_create("MySerialDiapatchQueue", DISPATCH_QUEUE_SERIAL);

// 生成并发队列
dispatch_queue_t myConcurrentDiapatchQueue =
dispatch_queue_create("MyConcurrentDiapatchQueue", DISPATCH_QUEUE_CONCURRENT);
```

GCD - dispatch_queue_create

因为一个串行队列，只生成并使用一个线程，所以创建几个串行队列就生成几个线程，

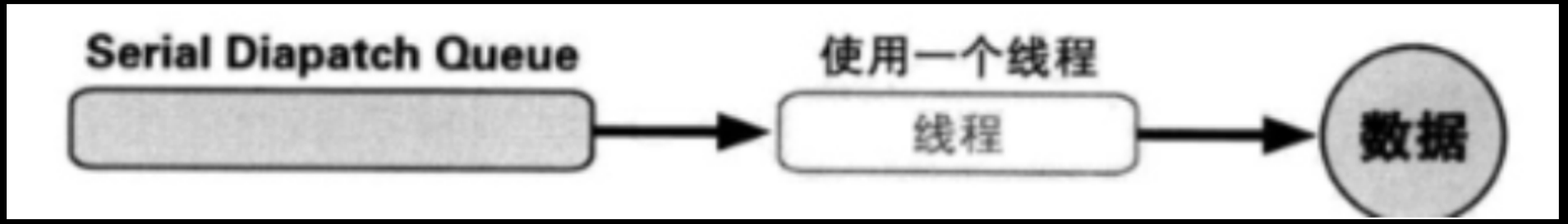
线程过多，会消耗大量内存,影响系统响应性能

问题：多个线程竞争同一资源时，会出现数据安全问题

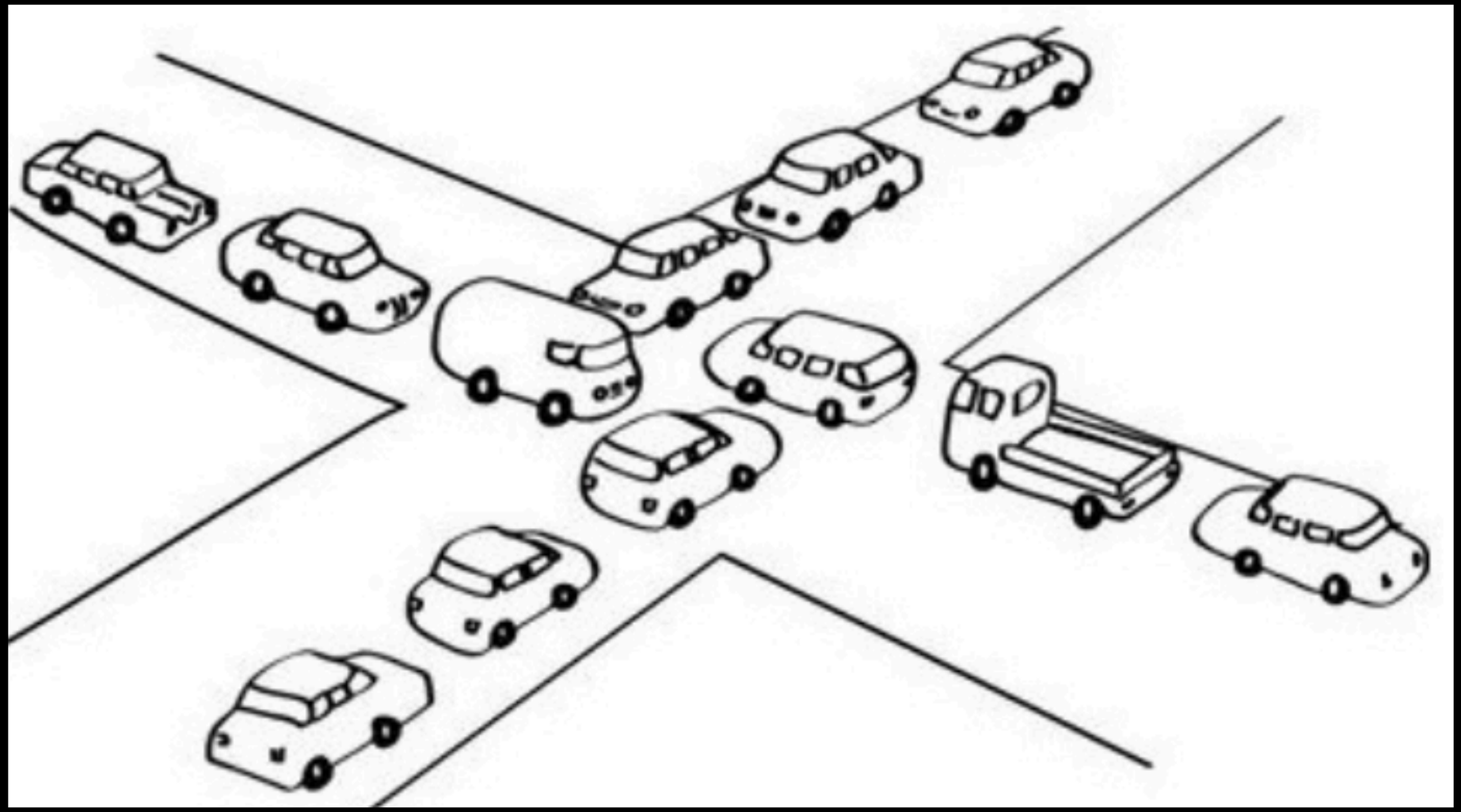


GCD - dispatch_queue_create

使用Serial Diapatch Queue可以保证数据的安全

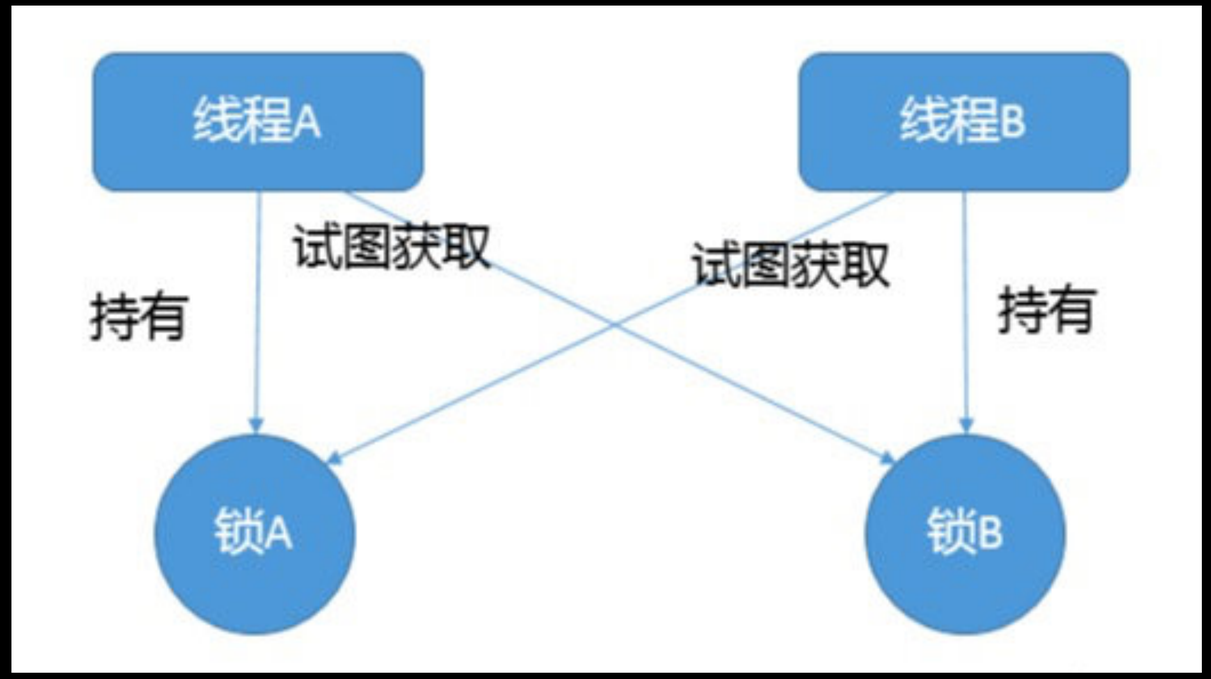


GCD - 死锁



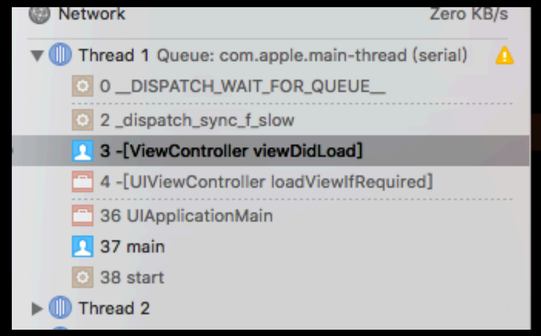
GCD - 死锁

因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去



GCD - 死锁

```
dispatch_sync(dispatch_get_main_queue(), ^{

});
```




```
22
23  - (void)viewDidLoad {
24      [super viewDidLoad];
25      dispatch_sync(dispatch_get_main_queue(), ^{
26
27      });
28
29
30
```

= Thread 1: EXC_BAD_INSTRUCTION (code=EXC_I386_INVOP, subcode=0x0)

死锁无法解决，只能避免

GCD - dispatch_after

ByteDance 字节跳动

dispatch_after主要进行延迟操作。但是延迟只说明再一定时间后，将任务添加到队列

中执行，真正开始执行任务，可能因为主线程本身的处理有延迟，导致时间不准确

```
/**
 * dispatch_time:获取dispatch_time_t类型的时间
 * DISPATCH_TIME_NOW:当前时间
 * NSEC_PER_SEC:单位秒
 */
dispatch_time_t time = dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3 *
NSEC_PER_SEC));
//  dispatch_after:在3秒后,将任务添加到队列中
dispatch_after(time, dispatch_get_main_queue(), ^{
    NSLog(@"wait at least three second");
});
```

GCD - dispatch group

ByteDance 字节跳动

调度组中的所有异步任务执行结束之后，会得到统一的通知

监听一组异步任务是否执行结束，如果执行结束就能够得到统一的通知

GCD - dispatch group

ByteDance 字节跳动

```
// 创建默认优先级的全局并发队列
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0);
// 创建调度组
dispatch_group_t group = dispatch_group_create();
dispatch_group_async(group, queue, ^{
    NSLog(@"下载图片A");
});
dispatch_group_async(group, queue, ^{
    NSLog(@"下载图片B");
});
dispatch_group_async(group, queue, ^{
    NSLog(@"下载图片C");
});
dispatch_group_notify(group, dispatch_get_main_queue(), ^{
    NSLog(@"处理下载完成的图片");
});
```

GCD - dispatch_group_enter、dispatch_group_leave

ByteDance 字节跳动

```
dispatch_group_t group = dispatch_group_create();
dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
dispatch_group_enter(group);
dispatch_async(queue, ^{
    NSBlockOperation *op = [NSBlockOperation blockOperationWithBlock:^{
        NSLog(@"group 1");
        dispatch_group_leave(group);
    }];
    [op start];
});

dispatch_group_enter(group);
dispatch_async(queue, ^{
    NSBlockOperation *op = [NSBlockOperation blockOperationWithBlock:^{
        NSLog(@"group 2");
        dispatch_group_leave(group);
    }];
    [op start];
});

dispatch_group_notify(group, dispatch_get_main_queue(), ^{
    NSLog(@"group end");
});
```

Text

GCD - dispatch_barrier_async

dispatch_barrier_async在并发执行任务的队列中追加处理任务，该任务在等待前面并发任务执行完成之后才执行,当dispatch_barrier_async中的任务执行完成，才会继续执行后续的并发执行的任务

GCD - dispatch_barrier_async

ByteDance 字节跳动

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

dispatch_async(queue, ^{
    NSLog(@"thread0"); // thread0和thread1输出顺序不固定
});

dispatch_async(queue, ^{
    NSLog(@"thread1");
});

dispatch_barrier_async(queue, ^{
    NSLog(@"thread2"); // thread2必定在thread0和thread1之后执行
});

dispatch_async(queue, ^{
    NSLog(@"thread3"); // thread3和thread4输出顺序不固定
});
dispatch_async(queue, ^{
    NSLog(@"thread4");
});
```

GCD - 单例 dispatch_once

ByteDance 字节跳动

```
- (instancetype)shareInstance {
    static ViewController *instance;
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[ViewController alloc] init];
    });
    return instance;
}
```


ByteDance
字节跳动