Erma

天将降大任于斯人也，必先苦其心志，劳其筋骨，饿其体肤，空乏其身，行拂乱其所为~

iOS多线程技术方案

iOS多线程技术方案

目录

一、多线程简介

1、多线程的由来

一个进程（进程）在执行一个线程（线程中有很多函数或方法（后面简称Function））的时候，其中有一个Function执行的时候需要消耗一些时间，但是这个线程又必须同时执行这个Function之后的Function，问题来了，一个线程中的任何一个Function都必须等待其执行完成后才能执行后面的Function，如果要同时执行两个或者多个Function，那么，就必须多开一个或者多个线程，这就是多线程的产生。我想多线程最开始的诞生就是由这而来吧！

2、耗时操作的模拟试验

2.1 循环测试

代码

```
int main(int argc, const char * argv[]) {
@autoreleasepool {
    NSLog(@"bengin");
    for (int i = 0; i < 10000000; i++) {
    }
    NSLog(@"end");
}
return 0;
}
```

控制台

```
2016-02-16 13:51:54.140 Test[1670:603696] bengin
2016-02-16 13:51:54.160 Test[1670:603696] end
Program ended with exit code: 0
```

结论一：循环一亿次耗时0.02秒，计算机的运行速度是非常快的

2.2 操作栈区

代码

```
int main(int argc, const char * argv[]) {
@autoreleasepool {
    NSLog(@"bengin");
    for (int i = 0; i < 10000000; i++) {
        int n = 1;
    }
    NSLog(@"end");
}
return 0;
}
```

控制台

```
2016-02-16 13:57:37.589 Test[1734:631377] bengin
2016-02-16 13:57:37.612 Test[1734:631377] end
Program ended with exit code: 0
```

结论二：对栈区操作一亿次，耗时0.023秒

2.3 操作常量区

代码：

```
int main(int argc, const char * argv[]) {
@autoreleasepool {
    NSLog(@"bengin");
    for (int i = 0; i < 10000000; i++) {
        NSString *str = @"hellow";
    }
```

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔

随笔分类(14)

随笔档案(11)

积分与排名

积分 - 22315
排名 - 10536

最新评论

1. Re:unity3D-iOS工程整合爬过的坑~
@Erma_Jack我完全按照了以上步骤执行了,然后运行出现两个错误:UI/UnityView.h file not found 和 il2cpp-config.h file not found,能加.....
--仆街仔
2. Re:unity3D-iOS工程整合爬过的坑~
@仆街仔不是移到废纸篓~ 是删除引用,,...
--Erma_Jack
3. Re:unity3D-iOS工程整合爬过的坑~
我想问下第四部那里删除引用那里是指把整个文件夹移到废纸篓里面?
--仆街仔
4. Re:Unity iOS混合开发界面切换思路
不错
--血色底裤
5. Re:iOS多线程技术方案
怎么感觉主播一点都不严谨，1000W当1E，begin写成bengin。TAT
--TommyBiteMe
6. Re:iOS多线程技术方案

```
    NSLog(@"end");
}
return 0;
}
```

控制台

```
2016-02-16 14:03:59.003 Test[1763:659287] bengin
2016-02-16 14:03:59.113 Test[1763:659287] end
Program ended with exit code: 0
```

结论三：对常量区操作一亿次，耗时0.11秒

2.4 操作堆区

代码

```
int main(int argc, const char * argv[]) {
@autoreleasepool {
    NSLog(@"bengin");
    for (int i = 0; i < 10000000; i++) {
        NSString *str = [NSString stringWithFormat:@"%d",i];
    }
    NSLog(@"end");
}
return 0;
}
```

控制台

```
2016-02-16 14:09:03.673 Test[1786:673719] bengin
2016-02-16 14:09:10.705 Test[1786:673719] end
Program ended with exit code: 0
```

结论四：对堆区操作一亿次耗时7秒多一些，较慢！

2.5 I/O操作

代码

```
int main(int argc, const char * argv[]) {
@autoreleasepool {
    NSLog(@"bengin");
    for (int i = 0; i < 10000000; i++) {
        NSLog(@"%d",i);
    }
    NSLog(@"end");
}
return 0;
}
```

控制台输出！正在跑中，一亿次！！！先看截图
CPU

质量很高的一篇文章
--mapanguan

7. Re:iOS多线程技术方案
@陈小怪引用一亿次。。？我还特地数了几个0...读完看了你的评论，我特地滚到上面数了数。。。。。...
--M.D.L

8. Re:iOS多线程技术方案
一亿次。。？我还特地数了几个0...
--陈小怪

9. Re:基于OpenSLL的RSA加密应用(非算法)
标题写错了，是OpenSSL
--msp的昌伟哥哥

10. Re:Unity iOS混合开发界面切换思路
@西海舰队首先，回答你第一个问题，不能，Unity在iOS中重新赋值Unity的View相当于改变了内存地址，其中会被释放一次再被复制一次，Unity会崩溃，也许和iOS平台Unity只能启动一次的原.....
--Erma_Jack

阅读排行榜

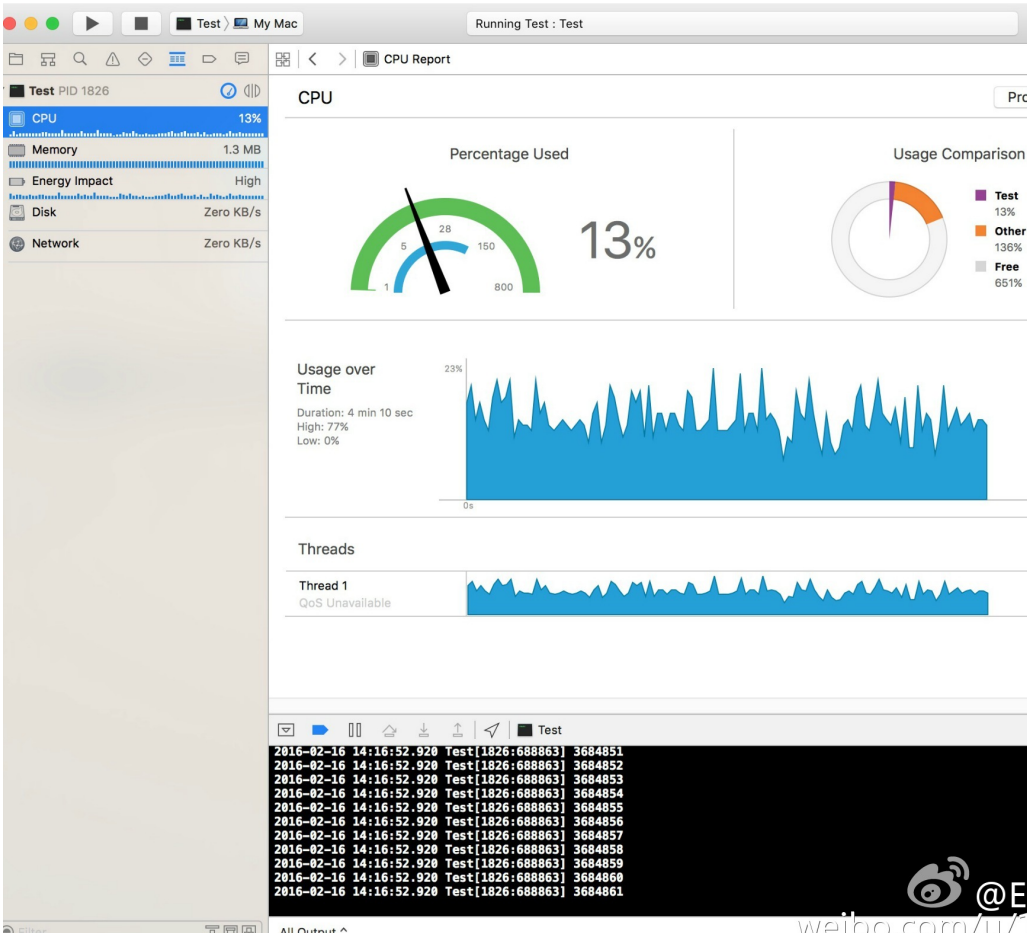
1. 放养的小爬虫--京东定向爬虫(AJAX获取价格数据)(6607)
2. 放养的小爬虫--拉钩网半智能整站小爬虫(3172)
3. 放养的小爬虫--豆瓣电影入门级爬虫(mongodb使用教程~)(1653)
4. unity3D-iOS工程整合爬过的坑~(1256)
5. Vuforia unity开发摄像头问题(532)
6. VR/AR 非技术总结(486)
7. Unity iOS混合开发界面切换思路(392)
8. Unity打开摄像头占满全屏(290)
9. iOS多线程技术方案(280)
10. 基于OpenSLL的RSA加密应用(非算法)(140)

评论排行榜

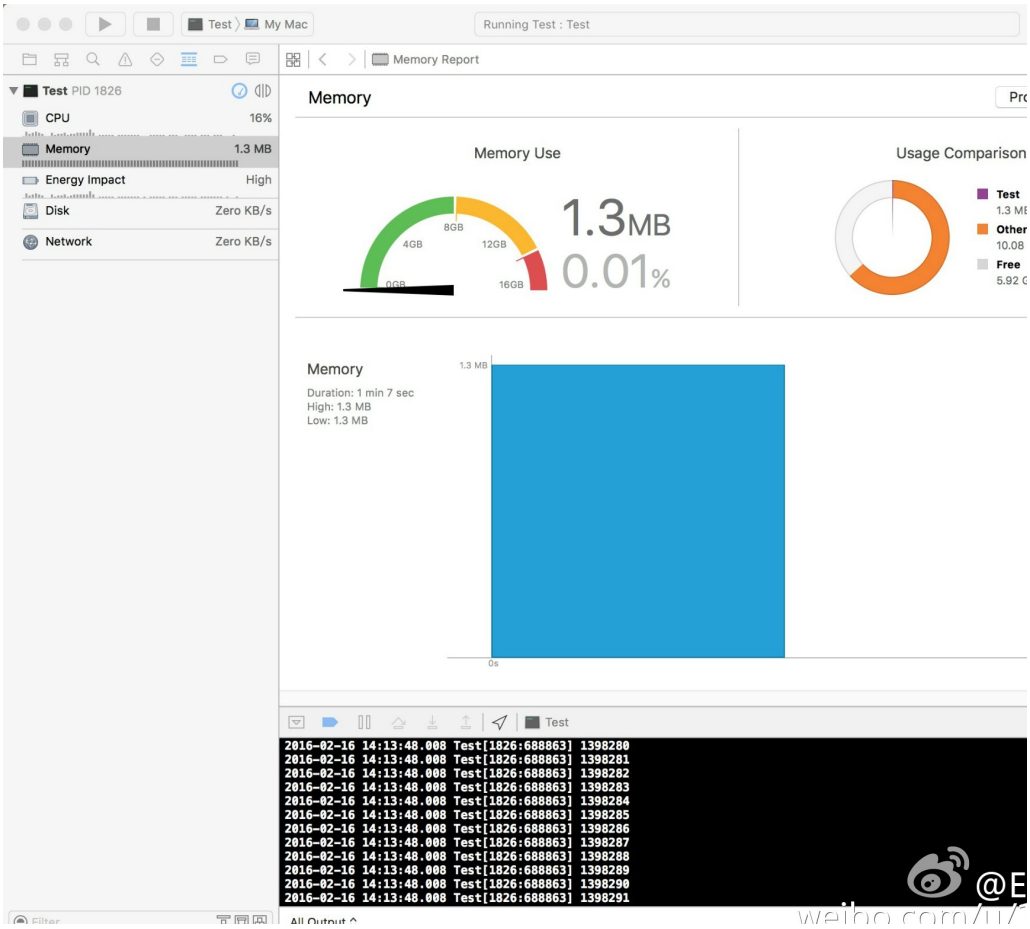
1. 放养的小爬虫--京东定向爬虫(AJAX获取价格数据)(78)
2. 放养的小爬虫--拉钩网半智能整站小爬虫(32)
3. unity3D-iOS工程整合爬过的坑~(14)
4. VR/AR 非技术总结(6)
5. iOS多线程技术方案(4)
6. Unity iOS混合开发界面切换思路(4)
7. 放养的小爬虫--豆瓣电影入门级爬虫(mongodb使用教程~)(3)
8. 基于OpenSLL的RSA加密应用(非算法)(1)

推荐排行榜

1. 放养的小爬虫--京东定向爬虫(AJAX获取价格数据)(38)
2. 放养的小爬虫--拉钩网半智能整站小爬虫(12)
3. 放养的小爬虫--豆瓣电影入门级爬虫(mongodb使用教程~)(8)
4. Unity iOS混合开发界面切换思路(5)
5. VR/AR 非技术总结(4)
6. iOS多线程技术方案(4)
7. 基于OpenSLL的RSA加密应用(非算法)



再看内存



好吧，还在跑，现在已经达到10分钟了，怕心疼本本炸掉！stop。。。

结论五：I/O操作非常慢，一亿次10分钟也没能跑完！

最终结论：通过以上结论一、二、三、四、五得出一个结论，各个区的执行效率：栈区>常量区>堆区>I/O操作。同时也说明了一个问题，执行不同的方法会产什么耗时操作。这是，为了解决耗时操作问题，多线程闪亮诞生！

3、进程和线程

先说说进程和线程吧！

3.1 进程

3.1.1 进程的概念： 系统中正在运行的应用程序。

3.1.2 进程的特点： 每个进程都运行在其专用且受保护的内存空间，不同的进程之间相互独立，互不干扰。

3.2 线程

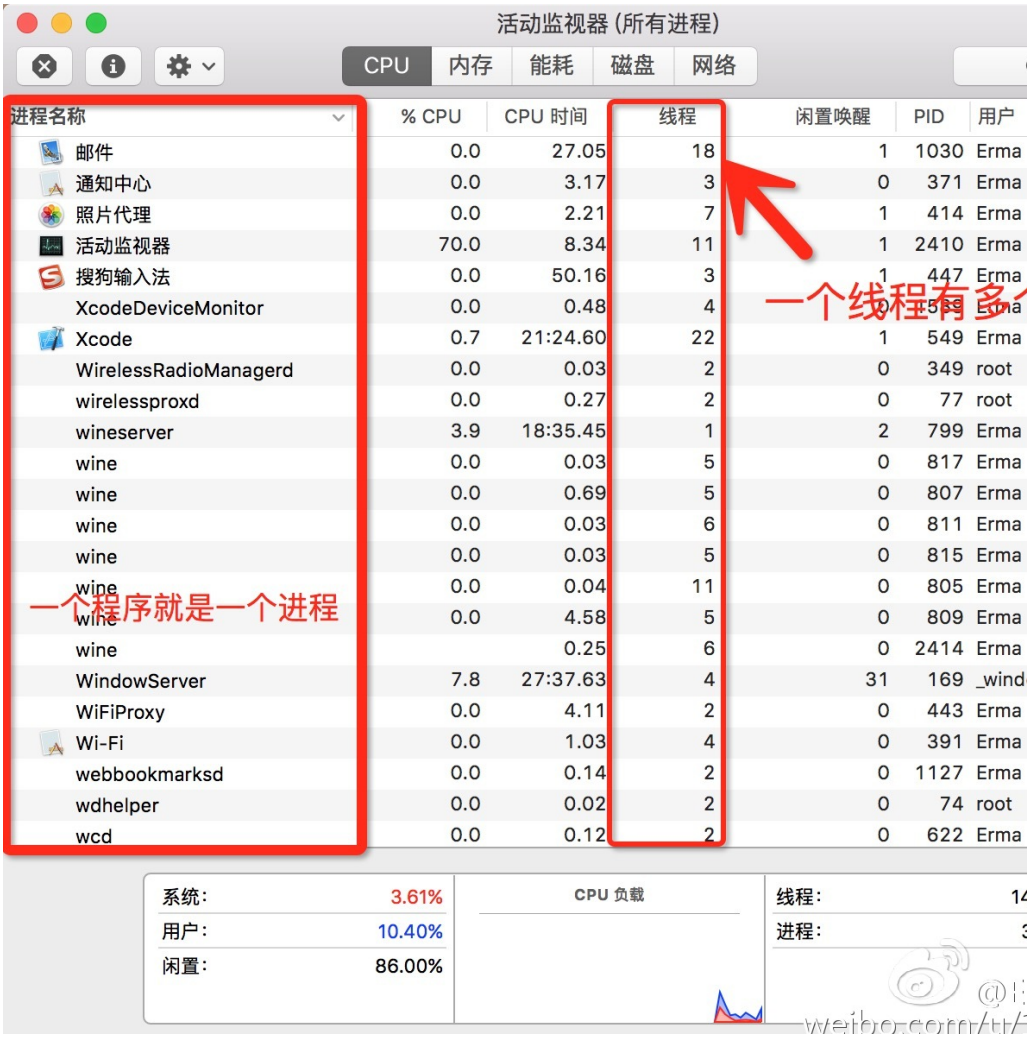
3.2.1 线程的概念：

线程是进程的执行任务的基本单元，一个进程的所有任务都是在线程中执行的。（每一个进程至少要有一条线程）。

3.2.2 线程的特点：

线程在执行任务的时候是按顺序执行的。如果要让一条线程执行多个任务，那么只能一个一个地并且按顺序执行这些任务。也就是说，在同一时间，一条线程只能执行一个任务。

我们可以通过Mac中的活动监视器查看进程和线程，下图！

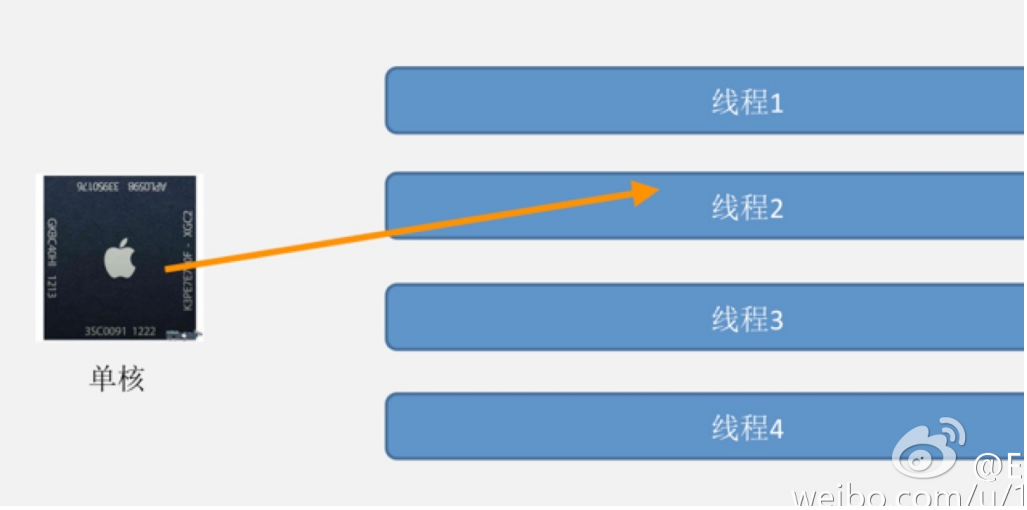


4、多线程的概念及原理

4.1 多线程概念：

1个进程可以开启多条线程，多条线程可以同时执行不同的任务。

4.2 多线程原理：



前提是在单核CPU的情况下，同一时间，CPU只能处理一条线程，也就是说只有一条线程在执行任务。多线程同时执行，那是不可能的！但是是CPU快速地在多条线程之间进行调度和切换执行任务。如果CPU调度线程的速度足够快，就会造成多条线程同时执行任务的"假象"，这种假象，就被美誉为：多线程！

5、多线程的优缺点和一个Tip

5.1 多线程的优点

- 可以适当提高程序的执行效率
- 也可以适当提高资源的利用率（CPU、内存利用率）

5.2 多线程的缺点

- 开启一条线程需要占用一定的内存空间（默认情况下，每一条线程都占用512KB），如果开启大量的线程，会占用大量的内存空间，从而降低程序的性能。
- 线程越多，CPU在调度和切换线程上的开销就会越大。
- 线程数越多，程序的设计会越复杂。

5.3 Tip

- 开启新的线程就会消耗资源，但是却可以提高用户体验。在保证良好的用户体验的前提下，可以适当地开线程，一般开3-6条。

Table 2-1 Thread creation costs

Item	Approximate cost	Notes
Kernel data structures	Approximately 1 KB	This memory is used to store the thread data structures and attributes, is allocated as wired memory and therefore cannot be paged to disk.
Stack space	512 KB (secondary threads) 8 MB (OS X main thread) 1 MB (iOS main thread)	The minimum allowed stack size for secondary threads is 16 KB and th must be a multiple of 4 KB. The space for this memory is set aside in y space at thread creation time, but the actual pages associated with tha not created until they are needed.
Creation time	Approximately 90 microseconds	This value reflects the time between the initial call to create the thread which the thread's entry point routine began executing. The figures we by analyzing the mean and median values generated during thread crea Intel-based iMac with a 2 GHz Core Duo processor and 1 GB of RAM ru v10.5.

- 开启一条新的线程，默认情况下，一条线程都是占用512KB，但是官方的文档里面给出的说明却不是，为了得出真相，下面做个小小的测试！

代码

```
int main(int argc, const char * argv[]) {
@autoreleasepool {
/** 操作主线程 */
NSLog(@"主线程默认 %tu", [NSThread currentThread].stackSize / 1024);
// 设置主线程的stackSize
[NSThread currentThread].stackSize = 1024 1024;
NSLog(@"主线程修改 %tu", [NSThread currentThread].stackSize / 1024);
```

```
/** 操作子线程 */
NSThread *thread = [[NSThread alloc] init];
NSLog(@"thread默认 %tu", thread.stackSize / 1024);
// 设置子线程的stackSize
thread.stackSize = 8 * 1024;
NSLog(@"thread修改 %tu", thread.stackSize / 1024);
[thread start];
```

```
}
return 0;
}
```

控制台

```
2016-02-17 08:36:02.652 Test[609:110129] 主线程默认 512
2016-02-17 08:36:02.654 Test[609:110129] 主线程修改 1024
2016-02-17 08:36:02.654 Test[609:110129] thread默认 512
2016-02-17 08:36:02.654 Test[609:110129] thread修改 8
```

结论七：证明了，不管什么线程，默认都是512，最小为8.可能是官方文档没有及时更新吧！

6、主线程

6.1 主线程的概念：

一个应用程序在启动运行后，系统会自动开启1条线程，这条称为”主线程”。

6.2 主线程的作用：主线程的作用主要用于处理UI界面刷新和UI时间！

6.3 结论： 主线程上不能执行耗时操作，这样会造成界面卡顿，给用户一种不好的体验。

7、技术方案

技术方案	简介	语言	线程生命周
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
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
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
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

***

二、Pthread

1、函数

```
pthread_create(pthread_t *restrict, const pthread_attr_t *restrict, void *(*)(void *),
void *restrict)
```

2、参数和返回值

- pthread_t *restrict 线程编号的地址
- const pthread_attr_t *restrict 线程的属性
- void *(*)(void *) 线程要执行的函数void * (*) (void *)
  - int * 指向int类型的指针 void * 指向任何类型的指针 有点类似OC中的id
- void *restrict 要执行的函数的参数
- 返回值 int类型 0是成功 非0 是失败

3、使用

代码

```
#import <Foundation/Foundation.h>
#import <pthread/pthread.h>

void *demo(void *param) {
NSString *name = (__bridge NSString *)(param);

NSLog(@"hello %@ %@",name,[NSThread currentThread]);
return NULL;
}

int main(int argc, const char * argv[]) {
@autoreleasepool {
    //创建子线程
    pthread_t pthread; //线程编号

    NSString *test = @"test";
    int result =  pthread_create(&pthread, NULL, demo, (__bridge void *)(test));
    NSLog(@"Began  %@",[NSThread currentThread]);

    if (result == 0) {
        NSLog(@"成功");
```

```
    }else {
        NSLog(@"失败");
    }
}
return 0;
}
```

控制台

```
2016-02-16 22:00:57.401 Test[888:42585] Began  <NSThread: x100502d70>{number = 1, name =
main}
2016-02-16 22:00:57.403 Test[888:42615] hello test <NSThread: x100102a30>{number = 2,
name = (null)}
2016-02-16 22:00:57.403 Test[888:42585] 成功
```

- __bridge 桥接，把OC中的对象，传递给c语言的函数，使用__bridge
***

三、NSThread

1、创建一个新的线程

- **方式一**

```
NSThread *thread = [[NSThread alloc] initWithTarget:self selector:@selector(demo) obj
ect:nil];
```

- **方式二**

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

- **方式三**

```
[self performSelectorInBackground:@selector(demo) withObject:nil];
```

2、线程的状态

线程状态分为五种

- 创建 New
- 就绪 Runnable
- 运行 Running

```
- (void)start;
```

- 阻塞（暂停） Blocked

```
+ (void)sleepUntilDate:(NSDate *)date;
+ (void)sleepForTimeInterval:(NSTimeInterval)ti;
```

- 死亡 Dead

```
+ (void)exit;
```

3、线程的属性

线程有两个重要的属性：名称和优先级

3.1 名称 name

设置线程名用于记录线程，在出现异常时可以DeBug

3.2 优先级，也叫做“服务质量”。threadPriority，取值0到1.

优先级或者服务质量高的，可以优先调用，只是说会优先调用，但是不是百分之百的优先调用，这里存在一个概率

问题，内核里的算法调度线程的时候，只是把优先级作为一个考虑因素，还有很多个因数会决定哪个线程优先调用。这点得注意注意！！！

下面是测试代码

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event {
//新建状态
NSThread *test1 = [[NSThread alloc] initWithTarget:self selector:@selector(demo)
object:nil];
test1.name = @"test1";
//线程的优先级
test1.threadPriority = 1.0;
//就绪状态
[test1 start];


//新建状态
NSThread *test2= [[NSThread alloc] initWithTarget:self selector:@selector(demo)
object:nil];
test2.name = @"test2";
test2.threadPriority = 0;
//就绪状态
[test2 start];
}

//线程执行完成之后会自动销毁
- (void)demo {
for (int i = 0; i < 20; i++) {
    NSLog(@"%d--%@",i,[NSThread currentThread]);
}
}
```

控制台

```
2016-02-16 22:43:28.182 05-线程状态[1241:78688] 0--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.182 05-线程状态[1241:78689] 0--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.182 05-线程状态[1241:78688] 1--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.182 05-线程状态[1241:78688] 2--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.182 05-线程状态[1241:78689] 1--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.183 05-线程状态[1241:78688] 3--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.183 05-线程状态[1241:78689] 2--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.183 05-线程状态[1241:78688] 4--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.183 05-线程状态[1241:78688] 5--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.183 05-线程状态[1241:78689] 3--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.183 05-线程状态[1241:78688] 6--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.183 05-线程状态[1241:78688] 7--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.183 05-线程状态[1241:78689] 4--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.183 05-线程状态[1241:78688] 8--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 9--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 10--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78689] 5--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 11--<NSThread: x7fead2017f30>{number = 2,
```

```
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78689] 6--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 12--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 13--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78689] 7--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 14--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78688] 15--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.185 05-线程状态[1241:78688] 16--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.184 05-线程状态[1241:78689] 8--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.185 05-线程状态[1241:78688] 17--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.185 05-线程状态[1241:78688] 18--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.185 05-线程状态[1241:78689] 9--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.185 05-线程状态[1241:78688] 19--<NSThread: x7fead2017f30>{number = 2,
name = test1}
2016-02-16 22:43:28.185 05-线程状态[1241:78689] 10--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.186 05-线程状态[1241:78689] 11--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.186 05-线程状态[1241:78689] 12--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.186 05-线程状态[1241:78689] 13--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.186 05-线程状态[1241:78689] 14--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.187 05-线程状态[1241:78689] 15--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.187 05-线程状态[1241:78689] 16--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.187 05-线程状态[1241:78689] 17--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.187 05-线程状态[1241:78689] 18--<NSThread: x7fead050a250>{number = 3,
name = test2}
2016-02-16 22:43:28.187 05-线程状态[1241:78689] 19--<NSThread: x7fead050a250>{number = 3,
name = test2}
```

结论六：优先级高，不一定先执行，只能说明先执行的概率要大一些！！！

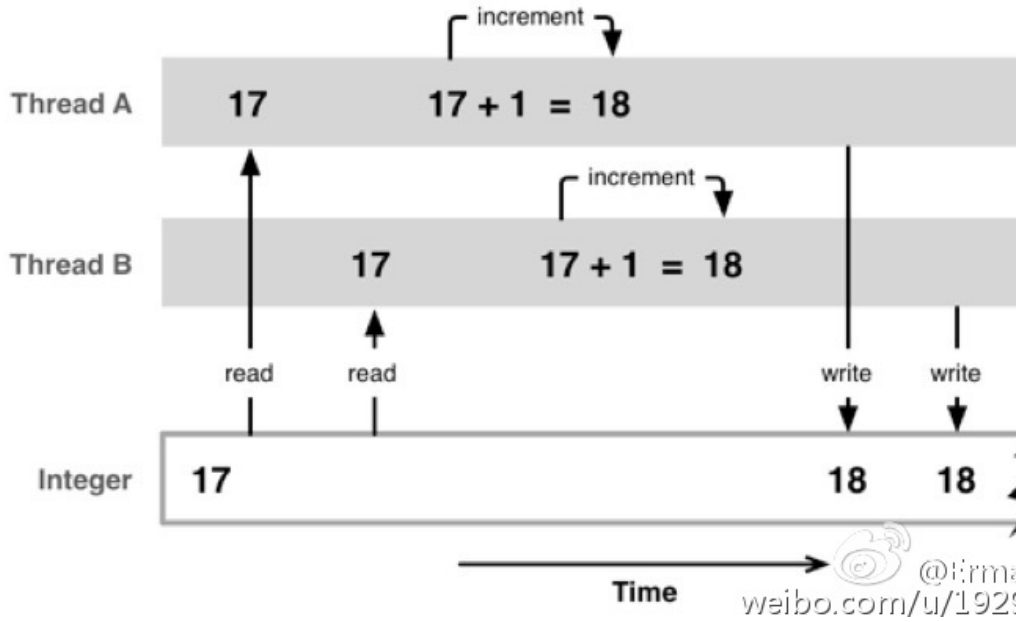
四、互斥锁

1、访问共享资源引入问题！

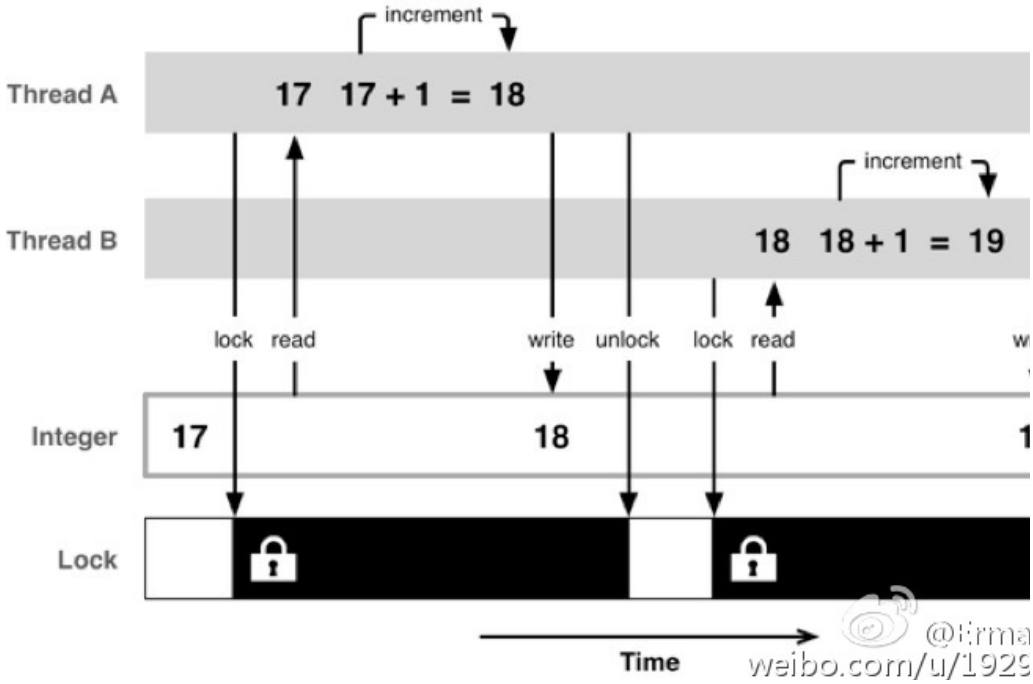
1.1 问题?

不同的线程要访问共享的资源，而且对共享的资源做操作，由于上面 结论六 得出服务质量和优先级不能决定线程执行的先后顺序，那么问题来了，一个线程对共享资源做了修改，而另外一个线程拿到的是未被修改之前资源，这是这个线程也对该资源做了修改，现在请问，两个线程都对该资源做了不同的修改，那么这个修改应该算谁的？！？！这就是问题所在！！！

1.2 问题分析



1.3 问题解决



这个文档里盗的图!

> 解决方案很简单，就是用一把锁锁住共享资源，等待线程1对其操作完毕后再打开，让线程2来执行，这就是传说中的互斥锁

！！！

2、互斥锁介绍

2.1 互斥锁代码

```
@synchronized(锁对象) { 需要锁定的代码 }
```

2.2 互斥锁的作用

可以防止因多线程执行顺序不定导致的抢夺资源造成的数据安全的问题

2.3 真相：互斥锁其实就是同步的意思，也就是按顺序执行！

3、互斥锁原理

每个NSObject对象内部都有一把锁，当线程要进入synchronized到对象的时候就要判断，锁是否被打开，如果打开，进入执行，如果锁住，继续等待，这就是互斥锁的原理！

4、互斥锁和自旋锁

自旋锁就是atomic！

4.1 原子属性和非原子属性（nonatomic 和 atomic）

- nonatomic:非原子属性，不会为 setter 方法加锁。
- atomic: 原子属性，为 setter 方法加锁(默认就是atomic)。
  - 通过给 setter 加锁，可以保证同一时间只有一个线程能够执行写入操作(setter)，但是同一时间允许多个线程执行读取操作(getter)。atomic本身就有一把自旋锁。
    这个特点叫做”单写多读”: 单个线程写入，多个线程读取。
  - atomic 只能保证写入数据的时候是安全的，但不能保证同时读写的时候是安全的。所以，不常使用！

4.2 nonatomic 和 atomic 的对比

atomic：线程安全(执行setter方法的时候)，需要消耗大量的资源。

nonatomic：非线程安全，适合内存小的移动设备。

4.3 互斥锁和自旋锁的区别

互斥锁

如果发现其它线程正在执行锁定代码，线程会进入休眠(阻塞状态)，等其它线程时间片到了打开锁后，线程就会被唤醒(执行)。

自旋锁

如果发现有其它线程正在执行锁定代码，线程会以死循环的方式，一直等待锁定的代码执行完成。
***

五、GCD

1、GCD介绍

全称Grand Central Dispatch,可翻译为”牛逼的中枢调度器”

纯C语言开发，是苹果公司为多核的并行运算提出的解决方案，会自动利用更多的CPU内核（比如双核、四核），可以自动管理线程的生命周期（创建线程、调度任务、销毁线程）。

2、GCD的两个核心

2.1 任务

- **执行的操作,在GCD中，任务是通过 block来封装的。并且任务的block没有参数也没有返回值。**

2.2 队列

- **存放任务**

包括

- 串行队列
- 并发队列
- 主队列
- 全局队列

队列的类型

Table 3-1 Types of dispatch queues

Type	Description
Serial	Serial queues (also known as *private dispatch queues*) execute one time in the order in which they are added to the queue. The current executing task runs on a distinct thread (which can vary from task that is managed by the dispatch queue. Serial queues are often use synchronize access to a specific resource. You can create as many serial queues as you need, and each queue concurrently with respect to all other queues. In other words, if you four serial queues, each queue executes only one task at a time bu tasks could still execute concurrently, one from each queue. For inf on how to create serial queues, see Creating Serial Dispatch Queue
Concurrent	Concurrent queues (also known as a type of *global dispatch queue*) one or more tasks concurrently, but tasks are still started in the orc they were added to the queue. The currently executing tasks run o threads that are managed by the dispatch queue. The exact numbe executing at any given point is variable and depends on system co In iOS 5 and later, you can create concurrent dispatch queues yours specifying `DISPATCH_QUEUE_CONCURRENT` as the queue type. In adc are four predefined global concurrent queues for your application t more information on how to get the global concurrent queues, see Global Concurrent Dispatch Queues.
Main dispatch queue	The main dispatch queue is a globally available serial queue that ex tasks on the application's main thread. This queue works with the application's run loop (if one is present) to interleave the execution tasks with the execution of other event sources attached to the run Because it runs on your application's main thread, the main queue used as a key synchronization point for an application. Although you do not need to create the main dispatch queue, you make sure your application drains it appropriately. For more inform how this queue is managed, see Performing Tasks on the Main Thr

3、函数

3.1 GCD函数

3.1.1 同步 dispatch_sync

同步：任务会在当前线程执行，因为同步函数不具备开新线程的能力。

```
void dispatch_sync(dispatch_queue_t queue, dispatch_block_t block);
```

3.1.2 异步 dispatch_async

异步：任务会在子线程执行，因为异步函数具备开新线程的能力。

```
void dispatch_async(dispatch_queue_t queue, dispatch_block_t block);
```

3.2 GCD使用步骤:

- 创建队列，或则获取队列
- 创建任务
- 将任务添加到队列中
  - GCD会自动将队列中的任务取出，放到对应的线程中执行
  - 任务取出遵循队列的FIFO原则：先进先出，后进后出

示例代码

```
// 1. 获取全局队列
dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
// 2. 创建任务
dispatch_block_t task = ^ {
    NSLog(@"hello %@", [NSThread currentThread]);
};
// 3. 将任务添加到队列，并且指定执行任务的函数
dispatch_async(queue, task);
```

通常写成一句代码

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
        NSLog(@"hello %@", [NSThread currentThread]);
    });
```

4、串行队列和并发队列

4.1 串行队列 (Serial Dispatch Queue)

4.1.1 特点

- 先进先出，按照顺序执行,并且一次只能调用一个任务
- 无论队列中所指定的执行任务的函数是同步还是异步，都必须等待前一个任务执行完毕才可以调用后面的人

4.1.2 创建一个串行队列

方法一

```
dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_SERIAL);
```

方法二

```
dispatch_queue_t queue = dispatch_queue_create("test", NULL);
```

4.1.3 串行队列，同步执行

代码：

```
// 1、创建串行队列
dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_SERIAL);
// 2、将任务添加到队列，并且指定同步执行
for (int i = 0; i < 10; i++) {
    dispatch_sync(queue, ^{
        NSLog(@"%@--%d",[NSThread currentThread],i);
    });
}
```

打印结果：

2016-02-25 16:31:07.849 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--0
2016-02-25 16:31:07.849 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--1
2016-02-25 16:31:07.849 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--2
2016-02-25 16:31:07.849 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--3
2016-02-25 16:31:07.849 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--4
2016-02-25 16:31:07.849 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--5
2016-02-25 16:31:07.850 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--6
2016-02-25 16:31:07.850 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name

= main}--7
2016-02-25 16:31:07.850 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--8
2016-02-25 16:31:07.850 test[1924:376468] <NSThread: 0x7ff040404370>{number = 1, name = main}--9

结论：串行队列，同步执行，不开新线程，按顺序执行

4.1.4 串行队列，异步执行

代码：

```
// 1、创建串行队列
dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_SERIAL);
// 2、将任务添加到队列，并且指定同步执行
for (int i = 0; i < 10; i++) {
    dispatch_async(queue, ^{
        NSLog(@"%@--%d",[NSThread currentThread],i);
    });
}
```

打印结果：

2016-02-25 17:08:32.167 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--0
2016-02-25 17:08:32.168 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--1
2016-02-25 17:08:32.168 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--2
2016-02-25 17:08:32.168 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--3
2016-02-25 17:08:32.168 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--4
2016-02-25 17:08:32.168 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--5
2016-02-25 17:08:32.169 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--6
2016-02-25 17:08:32.169 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--7
2016-02-25 17:08:32.169 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--8
2016-02-25 17:08:32.169 test[1959:391722] <NSThread: 0x7fbb98d24fa0>{number = 2, name = (null)}--9

结论：串行队列，异步执行，开启一条新的线程，按顺序执行

4.2 并发队列 (Concurrent Dispatch Queue)

4.2.1 特点

- 并发同时调度队列中的任务去执行
- 如果当前调度的任务是同步执行的，会等待当前任务执行完毕后，再调度后续的任务
- 如果当前调度的任务是异步执行的，同时底层线程池有可用的线程资源，就不会等待当前任务，直接调度任务到新线程去执行。

4.2.2 创建并发队列

```
dispatch_queue_t q = dispatch_queue_create("test", DISPATCH_QUEUE_CONCURRENT);
```

4.2.3 并发队列，同步执行

代码：

```
// 1. 创建并发队列
dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_CONCURRENT);
// 2. 将任务添加到队列，并且指定同步执行
for (int i = 0; i < 10; i++) {
    dispatch_sync(queue, ^{
```

```
        NSLog(@"%@ %d", [NSThread currentThread], i);
    });
}
```

输出：

2016-02-25 17:18:38.039 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 0
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 1
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 2
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 3
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 4
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 5
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 6
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 7
2016-02-25 17:18:38.040 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 8
2016-02-25 17:18:38.041 test[1979:399667] <NSThread: 0x7ffef86024b0>{number = 1, name = main} 9

结论:并发队列，同步执行，不开线程，顺序执行

4.2.4 并发队列，异步执行

代码：

```
// 1. 创建并发队列
dispatch_queue_t queue = dispatch_queue_create("test", DISPATCH_QUEUE_CONCURRENT);
// 2. 将任务添加到队列，并且指定同步执行
for (int i = 0; i < 10; i++) {
    dispatch_async(queue, ^{
        NSLog(@"%@ %d", [NSThread currentThread], i);
    });
}
```

输出：

2016-02-25 17:22:59.357 test[1992:403694] <NSThread: 0x7fe531c1a9b0>{number = 7, name = (null)} 6
2016-02-25 17:22:59.356 test[1992:403684] <NSThread: 0x7fe531d18fa0>{number = 3, name = (null)} 1
2016-02-25 17:22:59.356 test[1992:403689] <NSThread: 0x7fe534300610>{number = 5, name = (null)} 3
2016-02-25 17:22:59.356 test[1992:403683] <NSThread: 0x7fe531e94d80>{number = 2, name = (null)} 0
2016-02-25 17:22:59.356 test[1992:403692] <NSThread: 0x7fe531e9df80>{number = 6, name = (null)} 4
2016-02-25 17:22:59.356 test[1992:403693] <NSThread: 0x7fe531d18f40>{number = 8, name = (null)} 5
2016-02-25 17:22:59.356 test[1992:403695] <NSThread: 0x7fe5343015e0>{number = 9, name = (null)} 7
2016-02-25 17:22:59.357 test[1992:403688] <NSThread: 0x7fe531c16e30>{number = 4, name = (null)} 2
2016-02-25 17:22:59.357 test[1992:403694] <NSThread: 0x7fe531c1a9b0>{number = 7, name = (null)} 9
2016-02-25 17:22:59.357 test[1992:403696] <NSThread: 0x7fe531c237a0>{number = 10, name = (null)} 8

结论：开启足够多的线程，不按照顺序执行

CPU在调度的时候以最高效的方式调度和执行任务，所以会开启多条线程，因为并发，执行顺序不一定

5、主队列

5.1 主队列

主队列是系统提供的，无需自己创建，可以通过dispatch_get_main_queue()函数来获取。

5.2 特点

- 添加到主队列的任务只能由主线程来执行。
- 先进先出的，只有当主线程的代码执行完毕后，主队列才会调度任务到主线程执行

5.3 主队列 异步执行

代码

```
// 1. 获取主队列
dispatch_queue_t q = dispatch_get_main_queue();
// 2. 将任务添加到主队列，并且指定异步执行
for (int i = 0; i < 10; i++) {
dispatch_async(q, ^{
    NSLog(@"%@ %d", [NSThread currentThread], i);
});
}
// 先执行完这句代码，才会执行主队列中的任务
NSLog(@"hello %@", [NSThread currentThread]);
```

打印

2016-02-25 21:10:43.293 test[773:786816] hello <NSThread: 0x7ff158c05940>{number = 1, name = main}
2016-02-25 21:10:43.295 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 0
2016-02-25 21:10:43.295 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 1
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 2
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 3
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 4
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 5
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 6
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 7
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 8
2016-02-25 21:10:43.296 test[773:786816] <NSThread: 0x7ff158c05940>{number = 1, name = main} 9

打印结果得出的一些结论

- 在主线程顺序执行，不开启新的线程
- 主队列的特点：只有当主线程空闲时，主队列才会调度任务到主线程执行
- 主队列就算是异步执行也不会开启新的线程
- 主队列相当于一个全局的串行队列
- 主队列和串行队列的区别
  - 串行队列:必须等待一个任务执行完毕，才会调度下一个任务。
  - 主队列:如果主线程上有代码执行，主队列就不调度任务。

5.4 主队列 同步执行（死锁）

代码

```
    NSLog(@"begin");
// 1. 获取主队列
dispatch_queue_t q = dispatch_get_main_queue();
// 2. 将任务添加到主队列，并且指定同步执行
// 死锁
for (int i = 0; i < 10; i++) {
    dispatch_sync(q, ^{
        NSLog(@"%@ %d", [NSThread currentThread], i);
    });
}
NSLog(@"end");
```

打印

```
2016-02-25 21:19:25.986 test[791:790967] begin
```

打印结果可以看出，不是想要的结果，这时候发生了死锁
在主线程执行，主队列同步执行任务，会发生死锁，主线程和主队列同步任务相互等待，造成死锁

解决办法

代码

```
    NSLog(@"begin");
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    NSLog(@"--- %@", [NSThread currentThread]);
    // 1. 获取主队列
    dispatch_queue_t q = dispatch_get_main_queue();
    // 2. 将任务添加到主队列，并且指定同步执行
    // 死锁
    for (int i = 0; i < 10; i++) {
        dispatch_sync(q, ^{
            NSLog(@"%@ %d", [NSThread currentThread], i);
        });
    }
});
NSLog(@"end");
```

打印

2016-02-25 21:23:23.205 test[803:794826] begin
2016-02-25 21:23:23.206 test[803:794826] end
2016-02-25 21:23:23.206 test[803:794866] --- <NSThread: 0x7f8830514cb0>{number = 2, name = (null)}
2016-02-25 21:23:23.209 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 0
2016-02-25 21:23:23.209 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 1
2016-02-25 21:23:23.209 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 2
2016-02-25 21:23:23.209 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 3
2016-02-25 21:23:23.209 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 4
2016-02-25 21:23:23.210 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 5
2016-02-25 21:23:23.210 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 6
2016-02-25 21:23:23.210 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 7
2016-02-25 21:23:23.210 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 8
2016-02-25 21:23:23.210 test[803:794826] <NSThread: 0x7f8830507dd0>{number = 1, name = main} 9

打印结果可以看出，当我们将主队列同步执行任务放到子线程去执行，就不会出现死锁。由于将主队列同步放到了子线程中执行，主队列同步任务无法阻塞主线程执行代码，因此主线程可以将主线程上的代码执行完毕。当主线程执行完毕之后，就会执行主队列里面的任务。

6、全局队列

全局队列是系统提供的，无需自己创建，可以直接通过dispatch_get_global_queue(long identifier, unsigned long flags);函数来获取。

6.1 全局队列 异步执行

代码

```
    // 1. 获取全局队列
dispatch_queue_t q = dispatch_get_global_queue(0, 0);
// 2. 将任务添加到全局队列，异步执行
for (int i = 0; i < 10; i++) {
    dispatch_async(q, ^{
        NSLog(@"%d %@", i, [NSThread currentThread]);
    });
}
```

打印输出

2016-02-25 21:29:06.978 test[816:799523] 1 <NSThread: 0x7fd428e15760>{number = 3, name = (null)}
2016-02-25 21:29:06.978 test[816:799530] 4 <NSThread: 0x7fd428d2fbb0>{number = 6, name = (null)}
2016-02-25 21:29:06.978 test[816:799522] 0 <NSThread: 0x7fd428f094e0>{number = 2, name = (null)}
2016-02-25 21:29:06.978 test[816:799529] 3 <NSThread: 0x7fd428c0e1b0>{number = 5, name = (null)}
2016-02-25 21:29:06.978 test[816:799532] 6 <NSThread: 0x7fd428f06740>{number = 7, name = (null)}
2016-02-25 21:29:06.978 test[816:799533] 7 <NSThread: 0x7fd428d37be0>{number = 8, name = (null)}
2016-02-25 21:29:06.978 test[816:799531] 5 <NSThread: 0x7fd428e0c490>{number = 9, name = (null)}
2016-02-25 21:29:06.978 test[816:799526] 2 <NSThread: 0x7fd428d3e4b0>{number = 4, name = (null)}
2016-02-25 21:29:06.979 test[816:799534] 8 <NSThread: 0x7fd428d36ab0>{number = 10, name = (null)}
2016-02-25 21:29:06.979 test[816:799523] 9 <NSThread: 0x7fd428e15760>{number = 3, name = (null)}

特点：

1、全局队列的工作特性跟并发队列一致。 实际上，全局队列就是系统为了方便程序员，专门提供的一种特殊的并发队列。

2、全局队列和并发队列的区别：

* 全局队列没有名称，无论ARC还是MRC都不需要考虑内存释放，日常开发，建议使用全局队列
* 并发队列有名称，如果在MRC开发中，需要使用dispatch_*release来释放相应的对象，*dispatch_barrier 必须使用自定义的并发队列，开发第三方框架，建议使用并发队列

3、函数

```
dispatch_get_global_queue(long identifier, unsigned long flags);
```

这个函数中有两个参数：
第一个参数：identifier
iOS7.0，表示的是优先级：
DISPATCH_QUEUE_PRIORITY_HIGH = 2; 高优先级
DISPATCH_QUEUE_PRIORITY_DEFAULT = 0; 默认优先级
DISPATCH_QUEUE_PRIORITY_LOW = -2; 低优先级
DISPATCH_QUEUE_PRIORITY_BACKGROUND = INT16_MIN; 后台优先级
iOS8.0开始，推荐使用服务质量(QOS)：
QOS_CLASS_USER_INTERACTIVE = 0x21; 用户交互
QOS_CLASS_USER_INITIATED = 0x19; 用户期望
QOS_CLASS_DEFAULT = 0x15; 默认

QOS_CLASS_UTILITY = 0x11; 实用工具
QOS_CLASS_BACKGROUND = 0x09; 后台
QOS_CLASS_UNSPECIFIED = 0x00; 未指定
通过对比可知: 第一个参数传入0，可以同时适配iOS7及iOS7以后的版本。
服务质量和优先级是一一对应的:
DISPATCH_QUEUE_PRIORITY_HIGH: QOS_CLASS_USER_INITIATED
DISPATCH_QUEUE_PRIORITY_DEFAULT: QOS_CLASS_DEFAULT
DISPATCH_QUEUE_PRIORITY_LOW: QOS_CLASS_UTILITY
DISPATCH_QUEUE_PRIORITY_BACKGROUND: QOS_CLASS_BACKGROUND
第二个参数: flags
为未来保留使用的，始终传入0。
Reserved for future use.

7、GCD总结

1、开不开线程，由执行任务的函数决定

- 同步执行不开线程
- 异步执行开线程

2、异步执行任务，开几条线程由队列决定

- 串行队列，只会开一条线程，因为一条就足够了
- 并发队列，可以开多条线程，具体开几条由线程池决定

> 对主队列而言，不管是同步执行还是异步执行，都不会开线程。

最后盗图总结一张

	全局并行队列	手动创建串行队列	主队列
同步（sync）	没有开启新线程 串行执行任务	没有开启新线程 串行执行任务	会死锁
异步（async）	有开启新线程 并行执行任务	有开启新线程 串行执行任务	没有开启新 执行

六、NSOperation

1、NSOperation简介

1.1 NSOperation与GCD的区别：

- OC语言中基于 GCD 的面向对象的封装;
- 使用起来比 GCD 更加简单;
- 提供了一些用 GCD 不好实现的功能;
- 苹果推荐使用，使用 NSOperation 程序员不用关心线程的生命周期

1.2NSOperation的特点

- NSOperation 是一个抽象类，抽象类不能直接使用,必须使用它的子类
- 抽象类的用处是定义子类共有的属性和方法

2、核心概念

将操作添加到队列，异步执行。相对于GCD创建任务，将任务添加到队列。

将NSOperation添加到NSOperationQueue就可以实现多线程编程

3、操作步骤

- 先将需要执行的操作封装到一个NSOperation对象中
- 然后将NSOperation对象添加到NSOperationQueue中
- 系统会自动将NSOperationQueue中的NSOperation取出来
- 将取出的NSOperation封装的操作放到一条新线程中执行

4、NSInvocationOperation

No1.

代码

```
- (void)viewDidLoad {
[super viewDidLoad];
//创建操作，然后调用操作的start方法
NSInvocationOperation *op = [[NSInvocationOperation alloc] initWithTarget:self
selector:@selector(demo) object:nil];
NSLog(@"%d",op.isFinished);
[op start];
NSLog(@"%d",op.isFinished);
}

- (void)demo {
NSLog(@"hello %@",[NSThread currentThread]);
}
```

打印输出

2016-02-25 22:12:30.054 test[892:834660] 0
2016-02-25 22:12:30.054 test[892:834660] hello <NSThread: 0x7fad12704f80>{number = 1, name = main}
2016-02-25 22:12:30.054 test[892:834660] 1

结论：

[op start]在主线程中调用的，所以执行的线程也会是在主线程执行！ 重复调用start也只会执行一次，因为NSOperation会有一个属性去记住，是否已经完成了该操作！

No2.

代码

```
- (void)viewDidLoad {
[super viewDidLoad];
//    创建操作，将操作添加到NSOperationQueue中，然后就会异步的自动执行
NSInvocationOperation *op = [[NSInvocationOperation alloc] initWithTarget:self
selector:@selector(demo) object:nil];
//队列
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
//把操作添加到队列
[queue addOperation:op];
}

- (void)demo {
NSLog(@"hello %@",[NSThread currentThread]);
}
```

打印

2016-02-25 22:21:44.999 test[912:842412] hello <NSThread: 0x7fab92610080>{number = 2, name = (null)}

将操作添加到NSOperationQueue中，然后就会异步的自动执行

5、NSBlockOperation

NSBlockOperation 中使用block的方式让所有代码逻辑在一起，使用起来更加简便。

NO1.

代码

```
//创建操作
NSBlockOperation *op = [NSBlockOperation blockOperationWithBlock:^{
    NSLog(@"hello %@",[NSThread currentThread]);
}];
//更新op的状态，执行main方法，不会开新线程
[op start];
```

输出

2016-02-25 22:25:30.442 test[923:846208] hello <NSThread: 0x7fd410d055a0>{number = 1, name = main}

NO2.

代码

```
//    创建队列，创建操作，将操作添加到队列中执行
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
NSBlockOperation *op = [NSBlockOperation blockOperationWithBlock:^{
    NSLog(@"hello %@",[NSThread currentThread]);
}];
[queue addOperation:op];
```

输出
2016-02-25 22:26:48.064 test[934:848038] hello <NSThread: 0x7fc6bbb24c80>{number = 2, name = (null)}

NO3.

代码

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

[queue addOperationWithBlock:^{
    NSLog(@"hello %@",[NSThread currentThread]);

}];
```

输出

```
2016-02-25 22:27:56.445 test[945:850128] hello <NSThread: x7f98dbc2cae0>{number = 2, name = (null)}
```

创建队列，添加block形式的操作

七、案例

线程之间的通信问题

技术方案：NSOperation

```
[self.queue addOperationWithBlock:^{
    NSLog(@"异步下载图片");
    [[NSOperationQueue mainQueue] addOperationWithBlock:^{
        NSLog(@"回到主线程，更新UI");
    }];
}];
```

技术方案：GCD

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
      NSLog(@"下载图片---%@",[NSThread currentThread]);
    dispatch_async(dispatch_get_main_queue(), ^{
NSLog(@"回到主线程刷新图片的显示 -%@",[NSThread currentThread]);
```

```
        });
    });
```

这文章写了好久，，过年一直到现在，终于写完。。。

转载请注明来自吃饭睡觉撸码的博客 http://www.cnblogs.com/Erma-king/，并包含相关链接。

人与人斗，其乐无穷。人与天斗——人定胜天！！！

分类: iOS开发

Erma_Jack
关注 - 3
粉丝 - 125
+加关注

4

« 上一篇：基于OpenSLL的RSA加密应用(非算法)

posted @ 2016-10-06 18:38 Erma_Jack 阅读(280) 评论(4) 编辑 收藏

评论

#1楼 2016-10-06 21:03 | 陈小怪

一亿次。。？ 我还特地数了几个0...

支持(0) 反对(0)

#2楼 2016-10-07 14:21 | M.D.L

@ 陈小怪

引用

一亿次。。？ 我还特地数了几个0...

读完看了你的评论，我特地滚到上面数了数。。。。。

支持(0) 反对(0)

#3楼 2016-10-07 19:15 | mapanguan

质量很高的一篇文章

支持(1) 反对(1)

#4楼 2016-10-09 10:37 | TommyBiteMe

怎么感觉主播一点都不严谨，1000W当1E，begin写成bengin。TAT

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻:
· 孵化 投资 升级：网易有道要在外语在线教育上大做一笔
· 蚂蚁金服推出全球首项VR支付技术 VR行业有望迎来第二春
· 百度宣布成立200亿百度资本，在投资方面它是否能追上腾讯和阿里?
· 库克访问深圳 宣布苹果将在深圳设立研发中心
· 张锐身后事：他持有的春雨医生股权怎么办?
» 更多新闻...

最新知识库文章:
· 陈皓：什么是工程师文化?
· 没那么难，谈CSS的设计模式
· 程序猿媳妇儿注意事项
· 可是姑娘，你为什么要编程呢?
· 知其所以然（以算法学习为例）
» 更多知识库文章...