多线程

1. 简介：

(1) 一个应用对应一个进程，负责申请一块内存，每个进程启动后有一个主线程会开启。线程是程序的执行流。同一个进程中多个线程共用进程的内存空间。

(2) 遇到耗时的操作时，新建线程去执行可保证主线程的流畅

(3) 一开始多线程用于单核处理并发任务，由 CPU 整体调度；随着多核处理器的出现，新建线程时，具体在哪个 CPU 上执行任务，由系统调度

(4) 主线程是其他线程的父线程，负责所有界面显示操作和相应用户点击事件

(5) 每创建一个线程，都会消耗一定的内存和 CPU 时间(系统资源)

(6) 当多线程对同一资源进行抢夺时需注意线程安全问题

(7) 线程之间要控制执行顺序相对比较复杂，以及共享资源的争夺问题

(8) 多线程是为了并发同步完成多项任务，而不是为了提高运行效率，是为了通过提高资源使用效率提高系统的整体性能，不会提高单个算法本身的执行时间

(9) 线程使用不是无节制的：主线程堆栈大小 1M，其他线程 512K，且不能修改其值

(10) 的 NSThread:轻量级，使用简单，需自己管理线程生命周期、线程同步、加锁、睡眠等。线程同步对数据的加锁会有一定的系统开销

NSOperation:不需要关心线程管理，数据同步问题，可把精力放在自己要执行的操作上，NSOperation 是面向对象的

GCD:苹果开发的一个多核编程的解决方案。替代 NSOperation、NSThread 的高效和强大的技术，基于 C 语言

2. 多线程的使用

(1) NSObject 封装的基于 NSThread 的简单多线程方法

waitUntilDown:当前线程是否需要被阻塞，直到主线程将我们指定的代码执行完

注意：当前线程为主线程时，waitUntilDown 参数无效，UI 一般是回到主线程进行修改的

内存管理：线程任务需要包裹在@autoreleasepool 中，否者容易出现内存泄漏

```
- (void)multi_Thread_NSObject:(NSInteger)i{
    [self performSelectorInBackground:@selector(nsObjectSubMethod:) withObject:[NSNumber numberWithInteger:i]];
}
- (void)nsObjectSubMethod:(NSNumber *)integer{
   NSInteger  i = [integer integerValue];
    __weak typeof(self) weakSelf = self;
       NSLog(@"%@",[NSThread currentThread]);
       Ticket *ticket = [Ticket sharedTicket];
       if(ticket.ticket > 0){
          [weakSelf.logStr appendFormat:@"用户%ld购票成功，票数剩余%ld\n", (long)i, (long)-- ticket.ticket];
       }else{
          [weakSelf.logStr appendFormat:@"购票失败，余票不足！\n"];
       }
    [self performSelectorOnMainThread:@selector(nsObjectMainMethod) withObject:nil waitUntilDone:NO];
}
- (void)nsObjectMainMethod{
    [self.logTextView setText:self.logStr];
}
```

(2) NSThread 多线程:

```
- (void)multi_Thread_NSThread:(NSInteger)i{
    [NSThread detachNewThreadSelector:@selector(nsthreadSubMethod:) toTarget:self withObject:[NSNumber numberWithInteger:i]];
//       NSThread *thread = [[NSThread alloc]initWithTarget:self selector:@selector(nsoperationSubMethod:) object:[NSNumber numberWithInteger:i]];
//    [thread start];
}
- (void)nsthreadSubMethod:(NSNumber *)integer{
    NSInteger  i = [integer integerValue];
    __weak typeof(self) weakSelf = self;
    NSLog(@"%@",[NSThread currentThread]);
    Ticket *ticket = [Ticket sharedTicket];
    if(ticket.ticket > 0){
       [weakSelf.logStr appendFormat:@"用户%ld购票成功，票数剩余%ld\n", (long)i, (long)-- ticket.ticket];
    }else{
       [weakSelf.logStr appendFormat:@"购票失败，余票不足！\n"];
    }
     [self performSelectorOnMainThread:@selector(nsObjectMainMethod) withObject:nil waitUntilDone:NO];
}
```

缺点：控制线程生命周期难，控制并发线程数难，控制线程先后顺序难

(3) NSOperation&NSOperationQueue

```
- (void)multi_Thread_NSOperation:(NSInteger)i{
    NSInvocationOperation *operation = [[NSInvocationOperation alloc] initWithTarget:self selector:@selector(nsOperationSubMethod:) object:[NSNumber
       numberWithInteger:i]];
//    [operation start];

//    NSBlockOperation *operation = [NSBlockOperation blockOperationWithBlock:^{
//       [self nsoperationSubMethod:i];
//    }];
    [_queue addOperation:operation];
}
- (void)nsOperationSubMethod:(NSNumber *)integer{
    NSInteger  i = [integer integerValue];
    __weak typeof(self) weakSelf = self;
    NSLog(@"%@",[NSThread currentThread]);
    Ticket *ticket = [Ticket sharedTicket];
    if(ticket.ticket > 0){
       [weakSelf.logStr appendFormat:@"用户%ld购票成功，票数剩余%ld\n", (long)i, (long)-- ticket.ticket];
    }else{
       [weakSelf.logStr appendFormat:@"购票失败，余票不足！\n"];
    }
    [[NSOperationQueue mainQueue] addOperationWithBlock:^{
       [weakSelf.logTextView setText:weakSelf.logStr];
    }];
}
```

队列设置并发进程数

[self.queue setMaxConCurrentOperationCount:2]

线程执行顺序控制

[op2 addDependcy:op1];

注：不要建立循环依赖，否者系统不崩溃也不干活，依赖可以跨队列

(4) GCD 技术

1) 核心理念：针对多核处理器定制的，FIFO 队列，dispatch queue，可保证先来的任务先执行

2) 几种队列

全局队列：所有添加到全局队列到任务都是并发执行的

串行队列：所有添加到串行队列的任务都是顺序执行的

主队列：所有添加到主队列中的任务都是在主线程中执行的

3) 派发 dispatch

```
//异步 async 执行，并发执行
dspatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,0);//0 保留属性
dispatch_async(queue, ^{
 //下载图片后
 dispatch_async(dispatch_get_main_queue(),^{
    [_ setImage:image];
 });
});
```

4) 串行队列：需创建，不能 get

```
dispatch_queue_t = dispatch_queue_create(“”,
DISPATCH_QUEUE_SERIAL);
dispatch_async(queue, ^{
```

```
//任务 1
})
dispatch_async(queue, ^{
//任务 2
})
//结果 1->2 顺序执行
```

```
- (void)multi_Thread_GCD:(NSInteger)i{
    __weak typeof(self) weakSelf = self;
  dispatch_queue_t queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
  dispatch_async(queue, ^{
    NSLog(@"%@",[NSThread currentThread]);
    Ticket *ticket = [Ticket sharedTicket];
    if(ticket.ticket > 0){
      [weakSelf.logStr appendFormat:@"用户%ld购票成功，票数剩余%ld\n", (long)i, (long)-- ticket.ticket];
    }else{
      [weakSelf.logStr appendFormat:@"购票失败，余票不足！\n"];
    }
    dispatch_async(dispatch_get_main_queue(), ^{
      [weakSelf.logTextView setText:weakSelf.logStr];
    });
  });
}
```

5) 注意点

1. 串行队列异步任务，会使用同一个子线程依此执行
2. 并行队列的异步任务会使用多个子线程无序执行
3. 全局队列类似并行队列，但无法命名，调试，无法确认任务所在队列
4. 串行队列中的任务只会顺序执行
5. 并行队列：队列中的任务通常会并发执行
6. dispatch_async：异步操作，会并发执行，无法确定任务顺序
7. dspatch_sync：同步操作，会依次执行，能决定任务的执行顺序
8. 串行队列同步操作，操作不会新建线程，操作是顺序执行的
9. 串行队列异步操作，新建一个子线程顺序执行，是最安全的选择
10. 并行队列同步操作，操作不会新建线程，操作顺序执行
11. 并行队列异步操作，新建线程，操作无序执行
12. 主队列异步：操作都在主线程上顺序执行，不存在异步
13. 主队列同步：除非主线程被干掉，否者永远不会结束，主队列中添加的同步操作永远也不会执行，会死锁

14. 主队列异步虽然不开新线程，但会把异步任务降低优先级，空闲时执行

15. 同步任务执行完返回，异步任务直接返回。同步会阻塞当前线程，并把任务丢到相应的队列，并等待完成。

6) GCD 死锁案例

1. 当同步遇到了串行

```
NSLog(@“1”);
dipatch_sync(dispatch_get_main_queue(),^{
  NSLog(@“2”);
})
NSLog(@“3”);
```

结果：输出 1

2. 当同步遇到了并行

```
NSLog(@“1”);
dipatch_sync(dispatch_get_global_queue(...),^{
  NSLog(@“2”);
})
NSLog(@“3”);
```

结果：123 顺序输出

3. 同步与异步遇到了串行

```
dispatch_queue_t queue = dispatch_queue_create(“”, ...SERIAL);
NSLog(@“1”);
dispatch_async(queue,^{
  NSLog(@“2”);
  dispatch_sync(queue,^{
     NSLog(@“3”);
  })
  NSLog(@“4”);
```

```
})
NSLog(“5”);
```

结果：输出 125 或 152

4. 异步遇到了同步，回到主线程

```
NSLog(“1”);
dispatch_async(dispatch_get_global_queue(0,0),^{
  NSLog(@“2”);
  dispatch_sync(dispatch_get_main_queue(),^{
     NSLog(@“3”);
  })
  NSLog(@“4”);
})
  NSLog(“5”);
```

结果：12534 或 15234

5. 主线程出现了死循环

```
dispatch_async(dispatch_get_global_queue(0,0),^{
  NSLog(“1”);
  dispatch_sync(dispatch_get_main_queue(),^{
     NSLog(“2”);
  })
     NSLog(“3”);
  })
     NSLog(“4”);
     While(1){};
     NSLog(“5”);
```

结果：14 或 41

7) GCD 定时器

GCD 定时器不受 runLoop 约束，比　NSTimer 更加准时

实现

```
@property(nonatomic,strong) dispatch_source_t timer;

int count = 0;

dispatch_queue_t queue = dispatch_get_main_queue();

self.timer = dispatch_source_create(DISPATCH_SOURCE_TYPE_TIMER, 0, 0, queue);

//设置定时器参数

 dispatch_time start = dispatch_time(DISPATCH_TIME_NOW,(int64_t)1.0* NSEC_PER_SEC)比当前时间晚 1 秒

uint64_t interval = (uint64_t)(1.0*NSEC_PER_SEC);

dispatch_source_set_timer(self.timer, start, interval, 0);

//设置回调

dispatch_source_set_event_handler(self.timer, ^{

 NSLog(@”fdasf%@”,[NSThread currentThread]);

 count ++;

 if(count == 4){

    dispatch_cancel(self.timer);

    self.timer = nil;

 }

});

dispatch_resume(self.timer);

//只执行一次

double delayInSeconds = 2.0;

dispatch_time start = dispatch_time(DISPATCH_TIME_NOW,(int64_t) delayInSeconds *  NSEC_PER_SEC)比当前时间晚 delayInSeconds 秒

dispatch_after(start, dispatch_get_main_queue(), ^{

 //执行事件

})
```

8) 运行期间只执行一次代码

```
static dispatch_once_t onceToken;
dispatch_once(&onceToken,^{

});
```