大师班第二次考试介绍

正常考试 分为四种题型. 总分 200分 (`不要问我为什么不是100分,诶...就是玩!!!!🐱`)

- 1、选择题 (每题5分, 共7道 35分)
- 2、判断题 (每题5分, 共6道 30分)
- 3、简单题 (每题12分 共10道 120分)
- 4、拓展满分题 (15分)

> *下面我就贴出题目吧,如果你有时间的,不妨也拿笔本子测试一下,看看能做多少分,在文章留言我会第一时间发你答案! 或者加我微信:* `KC_Cooci`

一、选择题(每题5分) ⚠️ 有单选有多选哦⚠️

> - *1.* `LGTeacher` *继承于* `LGPerson` *，下面代码输出为什么 (a) 分值5分*

```
LGTeacher *t = [[LGTeacher alloc] init];

- (instancetype)init{
    self = [super init];
    if (self) {
        NSLog(@"%@ - %@",[self class],[super class]);
    }
    return self;
}
```

- ☐ A: LGTeacher – LGTeacher
- ☐ B: LGTeacher – LGPerson
- ☐ C: LGTeacher – NSObject
- ☐ D: LGTeacher – 它爱输出啥就输出啥,我不清楚

> - *2. 下面代码能否正常执行以及输出 (c) 分值5分*

```
@interface LGPerson : NSObject
@property (nonatomic, retain) NSString *kc_name;
- (void)saySomething;
@end

@implementation LGPerson
- (void)saySomething{
    NSLog(@"%s - %@",__func__,self.kc_name);
}
@end


- (void)viewDidLoad {
    [super viewDidLoad] ;

    Class cls = [LGPerson class];
```

```
    void  *kc = &cls;
    [(__bridge id)kc saySomething];
}
```

- [ ] A: 能 – ViewController
- [ ] B: 能 – null
- [ ] C: 能 – ViewController: 0x7ff8d240ad30
- [ ] D: 能不能自己运行一下不就知道,非要问我 – 它爱输出啥就输出啥,我不清楚

> - *3. 下面代码执行,控制台输出结果是什么 (d) 分值5分*

```
NSObject *objc = [NSObject new];
NSLog(@"%ld",CFGetRetainCount((__bridge CFTypeRef)(objc)));

void(^block1)(void) = ^{
    NSLog(@"---%ld",CFGetRetainCount((__bridge CFTypeRef)(objc)));
};
block1();

void(^__weak block2)(void) = ^{
    NSLog(@"---%ld",CFGetRetainCount((__bridge CFTypeRef)(objc)));
};
block2();

void(^block3)(void) = [block2 copy];
block3();

__block NSObject *obj = [NSObject new];
void(^block4)(void) = ^{
    NSLog(@"---%ld",CFGetRetainCount((__bridge CFTypeRef)(obj)));
};
block4();
```

- [ ] A: 1 2 2 2 2
- [ ] B: 1 2 3 3 2
- [ ] C: 1 3 3 4 1
- [ ] D: 1 3 4 5 1

> - *4. 下面代码执行,控制台输出结果是什么 (d) 分值5分*

```
- (void)MTDemo{
    while (self.num < 5) {
        dispatch_async(dispatch_get_global_queue(0, 0), ^{
            self.num++;
        });
    }
    NSLog(@"KC : %d",self.num);
}

- (void)KSDemo{
```

```
    for (int i= 0; i<10000; i++) {
        dispatch_async(dispatch_get_global_queue(0, 0), ^{
            self.num++;
        });
    }
    NSLog(@"Cooci : %d",self.num);
}
```

- [ ] A: 0 , 10000
- [ ] B: 0 , <10000
- [ ] C: <=5 , <10000
- [ ] D: >=5 , <10000

> - *5. 下面代码执行,控制台输出结果是什么 (a) 分值5分*

```
- (void)textDemo2{
    dispatch_queue_t queue = dispatch_queue_create("cooci", DISPATCH_QUEUE_CONCURRENT);
    NSLog(@"1");
    dispatch_async(queue, ^{
        NSLog(@"2");
        dispatch_sync(queue, ^{
            NSLog(@"3");
        });
        NSLog(@"4");
    });
    NSLog(@"5");
}

- (void)textDemo1{

    dispatch_queue_t queue = dispatch_queue_create("cooci", NULL);
    NSLog(@"1");
    dispatch_async(queue, ^{
        NSLog(@"2");
        dispatch_sync(queue, ^{
            NSLog(@"3");
        });
        NSLog(@"4");
    });
    NSLog(@"5");
}
```

- [ ] A: 1 5 2 3 4 , 1 5 2
- [ ] B: 1 5 2 4 3 , 死锁奔溃
- [ ] C: 1 5 2 3 4 , 死锁奔溃
- [ ] D: 1 5 2 4 3 , 死锁奔溃

> - *6. 下面代码执行,控制台输出结果是什么 (b) 分值5分*

```
@property (nonatomic, strong) NSMutableArray      *mArray;

- (NSMutableArray *)mArray{
    if (!_mArray) _mArray = [NSMutableArray arrayWithCapacity:1];
    return _mArray;
}
- (void)viewDidLoad {
    [super viewDidLoad];

    NSMutableArray *arr = [NSMutableArray arrayWithObjects:@"1",@"2", nil];
    self.mArray = arr;

    void (^kcBlock)(void) = ^{
        [arr addObject:@"3"];
        [self.mArray addObject:@"a"];
        NSLog(@"KC %@",arr);
        NSLog(@"Cooci: %@",self.mArray);
    };
    [arr addObject:@"4"];
    [self.mArray addObject:@"5"];

    arr = nil;
    self.mArray = nil;

    kcBlock();
}
```

- ☐ A: 1 2 4 5 3 , nil
- ☐ B: 1 2 4 5 3 , a
- ☐ C: 1 2 4 5 3 , 1 2 4 5 3 a
- ☐ D: 1 2 4 5 3 a , 1 2 4 5 3 a

二、判断题 (每题5分)

> - *1. 可变数组线程是安全 (❎) 分值5分*

- ☐ 对
- ☐ 错

> - *2. 主队列搭配同步函数就会产生死锁 (❎) 分值5分*

- ☐ 对
- ☐ 错

> - *3. 下面代码执行不会报错 (✅) 分值5分*

```
int a = 0;
void(^ __weak weakBlock)(void) = ^{
    NSLog(@"-----%d", a);
};

struct _LGBlock *blc = (__bridge struct _LGBlock *)weakBlock;
```

```
id __strong strongBlock = [weakBlock copy];
blc->invoke = nil;
void(^strongBlock1)(void) = strongBlock;
strongBlock1();
```

- ☐ 对
- ☐ 错

> - *4. 下面代码执行不会报错 (❎) 分值5分*

```
NSObject *a = [NSObject alloc];
void(^__weak block1)(void) = nil;
{
    void(^block2)(void) = ^{
        NSLog(@"---%@", a);
    };
    block1 = block2;
    NSLog(@"1 - %@ - %@",block1,block2);
}
block1();
```

- ☐ 对
- ☐ 错

> - *5. 下面代码会产生循环引用 (✅) 分值5分*

```
__weak typeof(self) weakSelf = self;
self.doWork = ^{
    __strong typeof(self) strongSelf = weakSelf;
    weakSelf.doStudent = ^{
        NSLog(@"%@", strongSelf);
    };
   weakSelf.doStudent();
};
self.doWork();
```

- ☐ 对
- ☐ 错

> - *6. 下面代码是否有问题 (✅) 分值5分*

```
- (void)demo3{
    dispatch_queue_t concurrentQueue = dispatch_get_global_queue(0, 0);

    for (int i = 0; i<5000; i++) {
        dispatch_async(concurrentQueue, ^{
            NSString *imageName = [NSString stringWithFormat:@"%d.jpg", (i % 10)];
            NSURL *url = [[NSBundle mainBundle] URLForResource:imageName withExtension:nil];
            NSData *data = [NSData dataWithContentsOfURL:url];
            UIImage *image = [UIImage imageWithData:data];

            dispatch_barrier_async(concurrentQueue, ^{
                [self.mArray addObject:image];
            });
```

```
        });
    }
}
```

- ☐ 对
- ☐ 错

> - *7. 下面代码不会产生循环引用 (❎) 分值5分*

```
static ViewController *staticSelf_;

- (void)blockWeak_static {
    __weak typeof(self) weakSelf = self;
    staticSelf_ = weakSelf;
}
```

- ☐ 对
- ☐ 错

三、简单题 (每题 10分 合计 120分)

请把它当成一场面试,认真对待 希望大家耐心 切忌浮躁 (和谐学习 不急不躁)

> - *1、请用GCD实现读写锁，解释为什么这么设计 分值10分*

GCD实现读写锁

读写锁具有以下特点：

- 同一时间，只能有一个线程进行写的操作。
- 同一时间，允许有多个线程进行读的操作。
- 同一时间，不允许既有写的操作，又有读的操作。

```
- (void)kc_safeWrite:(NSString *)result time:(int)time{
    dispatch_barrier_async(self.kc_queue, ^{
        sleep(time);
        self.mDict[@"kc"] = result;
        NSLog(@"写入: %@ -- %@",result,[NSThread currentThread]);
    });
}


- (NSString *)kc_safeRead:(int)time{
    __block NSString *result = @"还没有赋值";
    dispatch_sync(self.kc_queue, ^{
        result = self.mDict[@"kc"];
        NSLog(@"读取: %@ -- %@",result,[NSThread currentThread]);
    });
    return result;
}
```

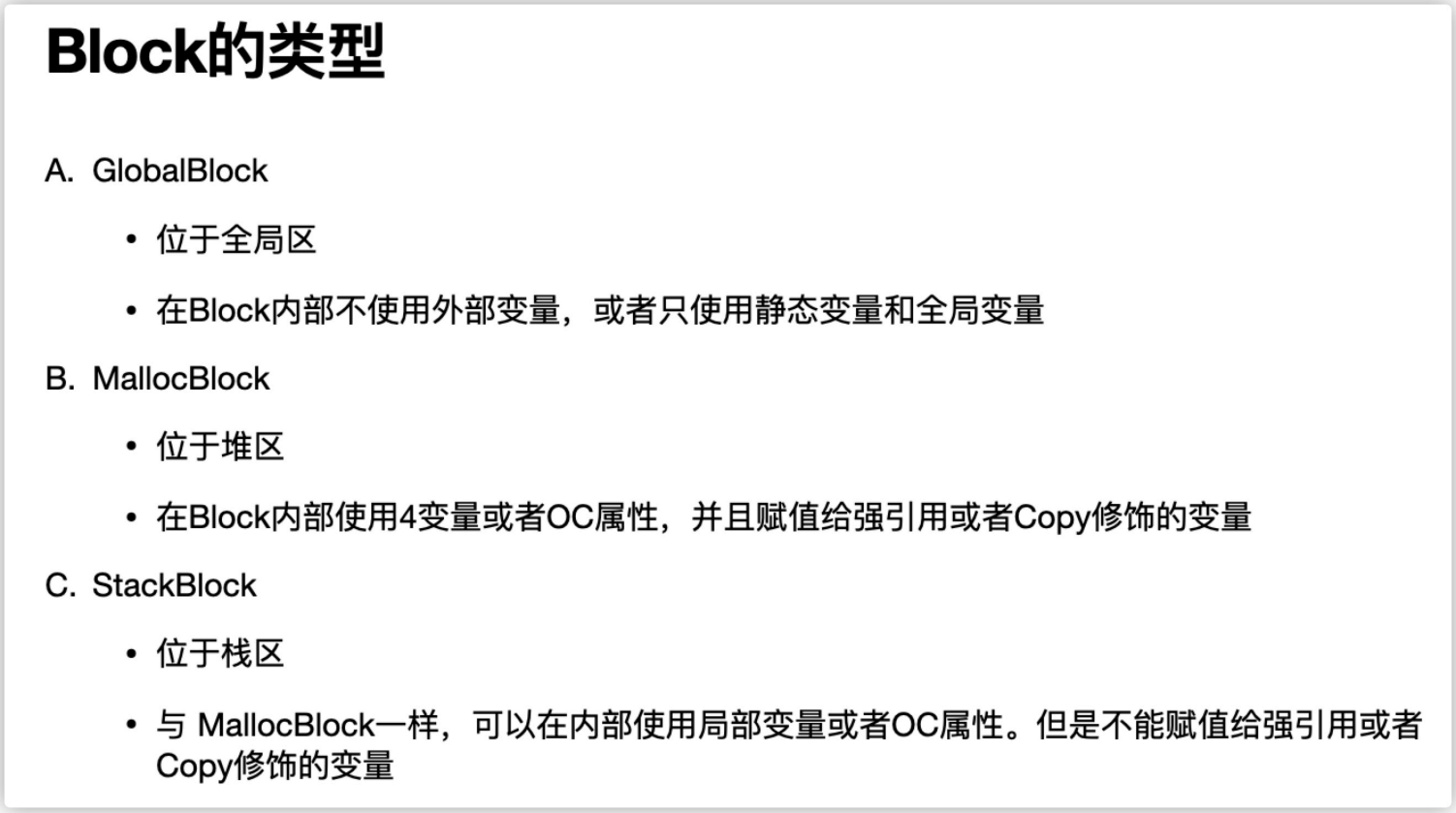
- `dispatch_barrier_async` 也被叫做栅栏块、同步点，简单的说就是在并行队列中的一个同步点，在 `dispatch_barrier_async` 之后的异步任务，需要等到 `dispatch_barrier_async` 执行完成后，才可以执行。

- 通过 `dispatch_sync` 来堵塞保证任务同步

> - *2、 `@synchronized` 为什么应用频次最多 分值10分*

- `@synchronized` 底层封装了是一把递归锁，可以自动进行加锁解锁，这也是大家喜欢使用它的原因
- `@synchronized` 中 `lockCount` 控制递归，而 `threadCount` 控制多线程 解决了锁的**可重入递归性**
- `@synchronized` 锁的写法也是最简单,最粗暴的！简单方便应该是程序员追求的方向

> - *3、 `block` 的种类有几种 是符合区分的？分值10分*



- 4、 `KVC` 普通对象 `setter` 过程 分值10分

`KVC` 可以通过 `key` 直接访问对象的属性，或者给对象的属性赋值，这样可以在运行时动态的访问或修改对象的属性

当调用 **setValue：**属性值 `forKey: @"name"` 的代码时，，底层的执行机制如下：

- 1、程序优先调用 `set<Key>:` 属性值方法，代码通过 `setter方法` 完成设置。注意，这里的 `<key>` 是指成员变量名，首字母大小写要符合 `KVC` 的命名规则，下同
- 2、如果没有找到 `setName:` 方法，KVC机制会检查 `+ (BOOL)accessInstanceVariablesDirectly` 方法有没有返回 `YES`，默认该方法会返回 `YES`，如果你重写了该方法让其返回NO的话，那么在这一步 `KVC` 会执行 `setValue: forUndefinedKey:` 方法，不过一般开发者不会这么做。所以KVC机制会搜索该类里面有没有名为 `<key>` 的成员变量，无论该变量是在类接口处定义，还是在类实现处定义，也无论用了什么样的访问修饰符，只在存在以 `<key>` 命名的变量，KVC都可以对该成员变量赋值。
- 3、如果该类即没有 `set<key>:` 方法，也没有 `_<key>` 成员变量，KVC机制会搜索 `_is<Key>` 的成员变量。
- 4、和上面一样，如果该类即没有set：方法，也没有_和_is成员变量，KVC机制再会继续搜索和is的成员变量。再给它们赋值。

- 5、如果上面列出的方法或者成员变量都不存在，系统将会执行该对象的 `setValue: forUndefinedKey:` 方法，默认是抛出异常。

如果想禁用KVC，重写 `+ (BOOL)accessInstanceVariablesDirectly` 方法让其返回NO即可，这样的话如果KVC没有找到 `set<Key>:` 属性名时，会直接用 `setValue: forUndefinedKey:` 方法。

当调用 **valueForKey：@”name“** 的代码时，KVC对key的搜索方式不同于 `setValue:` 属性值 `forKey: @”name“` ，其搜索方式如下：

- 1、首先按 `get<Key>,<key>,is<Key>` 的顺序方法查找 `getter` 方法，找到的话会直接调用。如果是 `BOOL` 或者 `Int` 等值类型，会将其包装成一个 `NSNumber` 对象
- 2、如果上面的 `getter` 没有找到， `KVC` 则会查找 `countOf<Key>,objectIn<Key>AtIndex` 或 `<Key>AtIndexes` 格式的方法。如果 `countOf<Key>` 方法和另外两个方法中的一个被找到，那么就会返回一个可以响应NSArray所有方法的代理集合(它是 `NSKeyValueArray` ，是 `NSArray` 的子类)，调用这个代理集合的方法，或者说给这个代理集合发送属于 `NSArray` 的方法，就会以 `countOf<Key>,objectIn<Key>AtIndex或<Key>AtIndexes` 这几个方法组合的形式调用。还有一个可选的 `get<Key>:range:` 方法。所以你想重新定义KVC的一些功能，你可以添加这些方法，需要注意的是你的方法名要符合KVC的标准命名方法，包括方法签名。
- 3、如果上面的方法没有找到，那么会同时查找 `countOf<Key>, enumeratorOf<Key>,memberOf<Key>` 格式的方法。如果这三个方法都找到，那么就返回一个可以响应NSSet所的方法的代理集合，和上面一样，给这个代理集合发NSSet的消息，就会以 `countOf<Key>, enumeratorOf<Key>,memberOf<Key>` 组合的形式调用。
- 4、如果还没有找到，再检查类方法 `+ (BOOL)accessInstanceVariablesDirectly` ,如果返回 `YES` (默认行为)，那么和先前的设值一样，会按 `_<key>,_is<Key>,<key>,is<Key>` 的顺序搜索成员变量名，这里不推荐这么做，因为这样直接访问实例变量破坏了封装性，使代码更脆弱。如果重写了类方法 `+ (BOOL)accessInstanceVariablesDirectly` 返回 `NO` 的话，那么会直接调用 `valueForUndefinedKey:` 方法，默认是抛出异常

> - *5、 `KVO` 底层原理机制分析 分值10分*

`KVO` 是基于 `runtime机制` 实现的， `KVO` 运用了 `isa-swizzling技术` ， `isa-swizzling` 就是类型混合指针机制, 将2个对象的isa指针互相调换, 就是俗称的黑魔法.

- 当某个类的属性对象第一次被观察时，系统就会在运行期动态地创建该类的一个派生类: `NSKVONotifying_xxx` ，在这个派生类中重写基类中任何被观察属性的setter 方法。派生类在被重写的 `setter` 方法内实现真正的通知机制。

例如如果原类为 `LGPerson` ，那么生成的派生类名为 `NSKVONotifying_LGPerson`

- 每个类对象中都有一个 `isa指针` 指向当前类，当一个类对象的第一次被观察，那么系统会偷偷将 `isa指针` 指向动态生成的派生类，从而在给被监控属性赋值时执行的是派生类的 `setter方法`
- 键值观察通知依赖于 `NSObject` 的两个方法: `willChangeValueForKey:` 和 `didChangevlueForKey:` ；在一个被观察属性发生改变之前， `willChangeValueForKey:` 一定会被调用，这就 会记录旧的值。而当改变发生后， `didChangeValueForKey:` 会被调用，
- 继而 通过消息或者响应机制去调用 `observeValueForKey:ofObject:change:context:`
- `removeObserver:` 的时候 `isa指针` 指回来

> - *6、下面代码的栈帧入栈情况是什么的? 分值10分*

```
- (void)viewDidLoad {
    [super viewDidLoad] ;


    Class cls = [LGPerson class];
    void  *kc = &cls;
```

```
    [(__bridge id)kc saySomething];

    LGPerson *person = [LGPerson alloc];
}
```

答: `self对象 -> cmd (viewDidLoad) -> self 类入栈 -> self对象 -> LGPerson 类入栈 -> person对象`

注意1⚠️: 参数的入栈顺序 和 结构体的入栈顺序

注意2⚠️: 这道题也考了内存平移的问题

注意3⚠️: 这道题也考了对象和类的结构问题

> - *7、iOS 线程如何保活, 为什么要线程保活 分值10分*

1: 自定义线程

```
#import "KCThread.h"

@implementation KCThread
- (void)dealloc {
    NSLog(@"%s",__func__);
}
@end
```

2: 使用细节

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

- (void)sayNB {
    @autoreleasepool {
        for (int i = 0; i < 100; i++) {
           NSLog(@"KC-子线程任务 %d - %@",i,[NSThread currentThread]);
        }
        NSLog(@"KC-子线程任务结束 - %@",[NSThread currentThread]);
        // 一般情况下开启线程任务后，当任务执行完毕后，线程就会被销毁，如果想让线程不死掉的话，需要为线程添加一个RunLoop
        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        [runLoop addPort:[NSMachPort port] forMode:NSRunLoopCommonModes];
        [runLoop run];
    }
}
```

3: 线程的dealloc方法不会执行 处理方式

```
- (IBAction)start:(id)sender {
    [self performSelector:@selector(saySomethingChildThread) onThread:self.thread withObject:nil waitUntilDone:NO];
    //waitUntilDone:YES 等到子线程任务执行完再执行下面NSLog
    //NO 不用等到子线程执行完再执行下面NSLog(下面NSLog在主线程，test在子线程，同时执行)
    NSLog(@"cooci 🐱 🐱 🐱");
}
```

4: 保活的线程如何回收

```
- (IBAction)stop:(id)sender {
    [self performSelector:@selector(exitThread) onThread:self.thread withObject:nil waitUntilDone:NO];
}
```

5: 解决循环引用问题

```
/如果使用如下方式创建thread，self会引用thread，thread会引用self，会造成循环引用。
KCThread *thread = [[KCThread alloc] initWithTarget:self selector:@selector(sayNB) object:nil];

//需要在exitThread中，进行如下设置
- (void)exitThread {
   // 设置标记为NO
    self.stopped = YES;
    // 停止RunLoop
    CFRunLoopStop(CFRunLoopGetCurrent());
    [self.thread cancel];
//解决循环引用问题
    self.thread = nil;
    NSLog(@"%s %@", __func__, [NSThread currentThread]);
}
```

线程保活: 在实际开发中经常会遇到一些耗时，且需要频繁处理的工作，这部分工作与UI无关，比如说大文件的下载，后台间隔一段时间进行数据的上报，APM中开启一个 `watch dog` 线程等。

> - *8、循环引用,为什么要在 `block` 中加 `strong` ,不加会怎样 分值10分*

通常我们在解决循环引用的时候是利用: `__weak typeof(self) weakSelf = self;` 虽然通过弱引用的 `weakSelf` 解决循环引用的无法释放的问题,但是会存在释放过早的问题!

例如我在 `block` 内部加入延时(这个很正常 请求延时😸) 但是用户操作过快,导致 `self` 提前释放 那么接下来的数据也就没有意义!

我们通过 `__strong __typeof(weakSelf)strongSelf = weakSelf;` 延长了生命周期,这样在 `strongSelf` 在作用空间能够有效.并且出了作用域也能及时的回收(`strongSelf` 临时变量而已)

> - *9、你使用过 `dispatch_once` 吗？了解底层吗？让你实现一个应该怎么操作？ 分值10分*

进入 `dispatch_once_f` 源码,其中的val是外界传入的onceToken静态变量,而func是 `_dispatch_Block_invoke(block)` ,其中单例的底层主要分为以下几步

- 将val,也就是静态变量转换为 `dispatch_once_gate_t` 类型的变量l
- 通过 `os_atomic_load` 获取此时的任务的标识符v
- 如果v等于 `DLOCK_ONCE_DONE` ,表示任务已经执行过了,直接return
- 如果 任务执行后,加锁失败了,则走到 `_dispatch_once_mark_done_if_quiesced` 函数,再次进行存储,将标识符置为 `DLOCK_ONCE_DONE`
- 反之,则通过 `_dispatch_once_gate_tryenter` 尝试进入任务,即解锁,然后执行 `_dispatch_once_callout` 执行 `block回调`
- 如果此时有任务正在执行,再次进来一个任务2,则通过 `_dispatch_once_wait` 函数让任务2进入无限次等待

> - *10、iOS 多线程原理和线程生命周期是什么样的 分值10分*

对于单核 `CPU` ,同一时间, `CPU` 只能处理一条线程,即只有一条线程在工作, `iOS` 中的多线程同时执行的本质是 `CPU` 在多个任务直接进行

快速的切换,由于 `CPU` 调度线程的时间足够快,就造成了多线程的“同时”执行的效果。其中切换的时间间隔就是时间片

对于多核cpu,具有真正意义上的并发!

1.新建状态:

用 `new关键字` 建立一个线程后，该线程对象就处于新建状态。处于新生状态的线程有自己的内存空间，通过调用 `start()` 方法进入就绪状态。

2.就绪状态:

处于就绪状态线程具备了运行条件，但还没分配到CPU，处于线程就绪队列，等待系统为其分配CPU。当系统选定一个等待执行的线程后，它就会从就绪状态进入运行状态，该动作称为“CPU调度”。

3.运行状态

在运行状态的线程执行自己的run方法中代码,直到等待某资源而阻塞或完成任何而死亡。如果在给定的时间片内没有执行结束，就会被系统给换下来回到就绪状态。

4.阻塞状态

处于运行状态的线程在某些情况下，如执行了sleep(睡眠)方法，或等待I/O设备等资源，将让出CPU并暂时停止自己运行，进入阻塞状态。

在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的I/O设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续执行。

5.死亡状态

死亡状态是线程生命周期中的最后一个阶段。线程死亡的原因有三个，一个是正常运行的线程完成了它的全部工作；另一个是线程被强制性地终止，如通过 `exit方法` 来终止一个线程【不推荐使用】；三是线程抛出未捕获的异常。

> - *11、请简述信号量和调度组的原理 分值10分*

信号量 作用一般是用来使任务同步执行,类似于互斥锁,用户可以根据需要控制GCD最大并发数!

1: 创建信号量, 并控制通行的额值 `dispatch_semaphore_t sem = dispatch_semaphore_create(1);`

2: `dispatch_semaphore_wait` 底层源码中是一个 `原有信号量 -1` 的一个操作. 当我们信号量值小于0 的时候会进入判断等待的时间. 如果设定为 `DISPATCH_TIME_FOREVER` 就进入 `do-while` 死循环. 等待信号发起

3: `dispatch_semaphore_signal` 原理

- 里面 `os_atomic_inc2o` 原子操作 `自增加1` ，然后会判断，如果value > 0，就会返回0。
- 例如 `value加1` 之后还是 `小于0` ，说明是一个负数，也就是调用 `dispatch_semaphore_wait` 次数太多了，加一次后依然小于0,并且 `== LONG_MIN` 就报异常 `Unbalanced call to dispatch_semaphore_signal()`
- 然后会调用 `_dispatch_semaphore_signal_slow` 方法的，做容错的处理， `_dispatch_sema4_signal` 是一个 `do while` 循环调用 `semaphore_signal` ，这样就能响应 `dispatch_semaphore_wait` 陷入的死循环等待

调度组

1: `dispatch_group_create` 创建组控制所有调度组的状态
2: 进组和出组

- `dispatch_group_enter` : 调度组 `value-1` .等待信号
- `dispatch_group_enter` 中 `old_value == DISPATCH_GROUP_VALUE_MAX` 证明

`Too many nested calls to dispatch_group_enter()` 也会报错

- `dispatch_group_leave` : 调度组 `value+1` 达到最开始设定的值就会调用 `_dispatch_group_wake` 去唤醒 `dispatch_group_enter` 堵塞的状态 进而去执行 `_dispatch_continuation_async` 进而执行任务
- `dispatch_group_leave` +1 还是 `value=0` 证明进组和出组不匹配

3: `dispatch_group_async` 进组任务 封装了 `dispatch_group_enter 进组` + `dispatch_group_leave 出组`

4: `dispatch_group_notify` : 当 `old_state == 0` 的时候调用 `_dispatch_group_wake` , 也就是调用 `block的callout` 。与 `leave` 调用了同一个方法. 其实起到一个监听通知执行的效果.

> - *12、请简述 __block 修饰变量被 block 捕获之后的情况 分值10分*

- 第一层拷贝对象本身,从栈区copy到堆区. 因为被 `__block` 修饰所以中间会产生 `Block_byref` 这样的结构体. 其中带有相关信息的成员变量
- 第二层: 因为被 `__block` 修饰所以中间会产生 `Block_byref` 这样的结构体. 其中带有相关信息的成员变量!

```
struct Block_byref {
    void * __ptrauth_objc_isa_pointer isa;
    struct Block_byref *forwarding;
    volatile int32_t flags; // contains ref count
    uint32_t size;
};

struct Block_byref_2 {
    // requires BLOCK_BYREF_HAS_COPY_DISPOSE
    BlockByrefKeepFunction byref_keep; //= __Block_byref_id_object_copy_131
    BlockByrefDestroyFunction byref_destroy; // = __Block_byref_id_object_dispose_131
};

struct Block_byref_3 {
    // requires BLOCK_BYREF_LAYOUT_EXTENDED
    const char *layout;
};
```

其中在真正执行的的时候就会调用: `BlockByrefKeepFunction byref_keep` 函数进行 `object_copy` 进而调用: `_Block_object_assign` 对外界对象的拷贝过程!

- 第三层: 外界 `Block_byref` 会被 `Block` 捕获之后,会赋值(copy) 一份! 通过 `_Block_object_assign` 函数调用: `*dest = _Block_byref_copy(object);` 发起对 `Block_byref` 的拷贝

四、拓展满分题 (15分)

1. 作为一名iOS中高级开发人员, 你的加分项和优势是什么? 分值20分