进阶班家庭作业

Runtime的家庭作业

一个对象的类方法查找流程

- A: 方法的本质就是消息，所有的类都会在编译初期加载进应用程序缓存
- B: `objc_msgSend` 的底层是汇编写的，所以会先通过***汇编快速***通过 `sel` 查找 `imp`
- C: 汇编找不到相关 `imp` ，我们就需要通过一个***慢速遍历查找***。根据 `isa` 的走位图,我们的类方法存在相关元类中，我们就会遍历元类-元类的父类-元类的父类的父类…nil 如果相关类找到了就会存储在缓存中，方便下次***快速查找***
- D: 如果第三步也没找到，就会开启***动态方法解析***（这是系统提供一次防止奔溃的方式）如果我们重写了 `+ (BOOL)resolveClassMethod:(SEL)sel` ,并且返回响应 `imp` ，程序会走到相关函数中去
- E: 如果没有实现 `+ (BOOL)resolveClassMethod:(SEL)sel` ，就查看是否有实现 `+ (BOOL)resolveInstanceMethod:(SEL)sel` ，拯救方式同 `D` 步骤
- F: `+ (id)forwardingTargetForSelector:(SEL)aSelector` 进行相关处理，防止奔溃
- G: `+ (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector` 获取相关签名，然后 `+ (void)forwardInvocation:(NSInvocation *)anInvocation` 进行消息转发
- H: 如果都没有处理，那么我们的消息就会 `crash` , `Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: '+[LGPerson walk]: unrecognized selector sent to class 0x100001310`

Runloop的家庭作业

runloop和线程的关系

- runloop与线程是一一对应的，一个runloop对应一个核心的线程，为什么说是核心的，是因为runloop是可以嵌套的，但是核心的只能有一个，他们的关系保存在一个全局的字典里。
- runloop是来管理线程的，当线程的runloop被开启后，线程会在执行完任务后进入休眠状态，有了任务就会被唤醒去执行任务。
- runloop在第一次获取时被创建，在线程结束时被销毁。
- 对于主线程来说，runloop在程序一启动就默认创建好了。
- 对于子线程来说，runloop是懒加载的，只有当我们使用的时候才会创建，所以在子线程用

定时器要注意：确保子线程的runloop被创建，不然定时器不会回调。

KVC的家庭作业

KVC的赋值过程

- 先找相关方法 set:, _set:, setIs:
- 若没有相关方法 + (BOOL)accessInstanceVariablesDirectly，判断是否可以直接方法成员变量
- 如果是判断是NO,直接执行KVC的setValue:forUndefinedKey:(系统抛出一个异常，未定义key)
- 如果是YES，继续找相关变量_ _is is
- 方法或成员都不存在，setValue:forUndefinedKey:方法，默认是抛出异常

KVO的家庭作业

KVO原理

- 动态生成子类-NSKVONotifying_xxx
- 动态添加setter方法
- 动态添加class方法
- 动态添加dealloc方法
- 开启手动观察
- 消息转发给我们的原类
- 消息发送-响应回调方法

多线程的家庭作业

多线程的原理

cpu在单位时间片里快速在各个线程之间切换

同步函数串行队列

- 不会开启线程，在当前线程执行任务
- 任务串行执行，任务一个接着一个
- 会产生堵塞

异步函数串行队列

- 开启一条线程，在当前线程执行任务
- 任务串行执行，任务一个接着一个

同步函数并发队列

- 不会开启线程，在当前线程执行任务
- 任务串行执行，任务一个接着一个

异步函数并发队列

- 开启线程，在当前线程执行任务
- 任务异步执行，没有顺序，CPU调度有关

下面代码的执行顺序

```
// 同步队列
dispatch_queue_t queue = dispatch_queue_create("cooci", DISPATCH_QUEUE_CONCUR
RENT);
NSLog(@"1");
// 异步函数
dispatch_async(queue, ^{
    NSLog(@"2");
    // 同步
    dispatch_sync(queue, ^{
    });
});
NSLog(@"5");
```

答： 152 死锁奔溃

下面代码的打印结果是怎么样的?

```
while (a<5) {
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        a++;
        NSLog(@"在里面a == %d -- %@",a,[NSThread currentThread]);
    });
}
NSLog(@"a == %d",a);
```

答：很大可能性： >= 5

网络家庭作业

为什么是三次握手，四次挥手?

- 在三次握手之后，A和B都能确定这么一件事： 我说的话，你能听到； 你说的话，我也能听到。 这样，就可以开始正常通信了。
- 注意： HTTP是基于TCP协议的，所以每次都是客户端发送请求，服务器应答，但是TCP还可以给其他应用层提供服务，即可能A、B在建立链接之后，谁都可能先开始通信。
- 如果两次，那么B无法确定B的信息A是否能收到，所以如果B先说话，可能后面的A都收不到，会出现问题 。
- 如果四次，那么就造成了浪费，因为在三次结束之后，就已经可以保证A可以给B发信息，A可以收到B的信息； B可以给A发信息，B可以收到A的信息。

为什么四次挥手最后还需要等待两个2MSL（最长报文段寿命）时间

- 为了保证A发送的最后一个ACK报文能够到达B。如果B没有收到，则会重传自己的FIN+ACK报文段，A在2MSL时间内收到B的报文段，接着A重新确认一次，重新启动2MSL计时器。
- 为了防止 “三次握手”中出现的 “已经失效的连接请求报文段”。在A发送完最后一个ACK报文段后，在经过时间2MSL就可以让本连接持续的时间内所产生的所有报文段都从网络中消失，确保下一个新的连接不会出现旧连接的报文段。