面试题总结：

GCD怎么用的？

• 1.串行队列，同步操作，不会新建线程，操作顺序执行；

• 串行队列，异步操作，会新建线程，操作顺序进行，使用场景：既不影响主线程，又需要顺序执行的操作；

• 2.并行队列，同步操作，不会新建县城，操作顺序执行；

• 并行队列，异步操作，会新建线程，操作无序进行，队列前如果有其他任务，会等待其他任务执行完毕再执行；

• 全局队列是系统的，直接get就可以用

• UI的更新工作必须在主线程进行，

• 全局队列异步操作，会新建对个子线程，操作无序执行，如果队列前有其他任务，会等待其他任务执行完毕在调用；

• 全局队列同步操作，不会新建线程，顺序执行

• 主队列所有的操作都是主线程顺序执行，没有异步概念，主队列添加的同步操作永远不会执行，会死锁

• 单例模式

• allocwithzone是对象分配内存空间时，最终会调用的方法，重写该方法，保证只会分配一块内存dispatch\_once是线程安全的，保证块代码中的内容只会执行一次

•

• 串行队列添加的同步操作会死锁，但是会执行嵌套同步操作之前的代码；

• 并行队列添加的同步操作不会死锁都在主线程执行；

• 全局队列添加的同步操作不会死锁。

•

• 同步操作 最主要的目的，阻塞并行队列任务的执行，只有当前的同步任务执行完毕之后，后边的任务才会执行，应用：用户登录

• >1 队列和线程的区别：

    队列：是管理线程的，相当于线程池,能管理线程什么时候执行。

    队列分为串行队列和并行队列

    串行队列：队列中的线程按顺序执行（不会同时执行）

    并行队列：队列中的线程会并发执行，可能会有一个疑问，队列不是先进先出吗，如果后面的任务执行完了，怎么出去的了。这里需要强调下，任务执行完毕了，不一定出队列。只有前面的任务执行完了，才会出队列，也就是说你即使执行完毕了，也必须等前面的任务执行完毕出队列，才可以出去。

• >2 主线程队列和GCD创建的队列也是有区别的。

    主线程队列和GCD创建的队列是不同的。在GCD中创建的队列优先级没有主队列高，所以在GCD中的串行队列开启同步任务里面没有嵌套任务是不会阻塞主线程，只有一种可能导致死锁，就是串行队列里，嵌套开启任务，有可能会导致死锁。

    主线程队列中不能开启同步，会阻塞主线程。只能开启异步任务，开启异步任务也不会开启新的线程，只是降低异步任务的优先级，让cpu空闲的时候才去调用。而同步任务，会抢占主线程的资源，会造成死锁。

• 3> 线程：里面有非常多的任务（同步，异步）

    同步与异步的区别：

    同步任务优先级高，在线程中有执行顺序，不会开启新的线程。

    异步任务优先级低，在线程中执行没有顺序，看cpu闲不闲。在主队列中不会开启新的线程，其他队列会开启新的线程。

• • \*  主线程队列注意：

 下面代码执行顺序

 1111

 2222

 主队列异步 <NSThread: 0x8e12690>{name = (null), num = 1}

 在主队列开启异步任务，不会开启新的线程而是依然在主线程中执行代码块中的代码。为什么不会阻塞线程？

 > 主队列开启异步任务，虽然不会开启新的线程，但是他会把异步任务降低优先级，等闲着的时候，就会在主线程上执行异步任务。

 在主队列开启同步任务，为什么会阻塞线程？

 > 在主队列开启同步任务，因为主队列是串行队列，里面的线程是有顺序的，先执行完一个线程才执行下一个线程，而主队列始终就只有一个主线程，主线程是不会执行完毕的，因为他是无限循环的，除非关闭应用程序。因此在主线程开启一个同步任务，同步任务会想抢占执行的资源，而主线程任务一直在执行某些操作，不肯放手。两个的优先级都很高，最终导致死锁，阻塞线程了。

• - (void)main\_queue\_deadlock

{

    dispatch\_queue\_t q = dispatch\_get\_main\_queue();

    NSLog(@"1111");

    dispatch\_async(q, ^{

        NSLog(@"主队列异步 %@", [NSThread currentThread]);

    });

    NSLog(@"2222");

    // 下面会造成线程死锁

    //    dispatch\_sync(q, ^{

    //        NSLog(@"主队列同步 %@", [NSThread currentThread]);

    //    });

}

• 并行队列里开启同步任务是有执行顺序的，只有异步才没有顺序；

• 串行队列开启异步任务，是有顺序的

• 串行队列开启异步任务后嵌套同步任务造成死锁

• - (void)serial\_queue\_deadlock2

• {

•     dispatch\_queue\_t q = dispatch\_queue\_create("cn.itcast.gcddemo", DISPATCH\_QUEUE\_SERIAL);

•

•

•     dispatch\_async(q, ^{

•         NSLog(@"异步任务 %@", [NSThread currentThread]);

•         // 下面开启同步造成死锁：因为串行队列中线程是有执行顺序的，需要等上面开启的异步任务执行完毕，才会执行下面开启的同步任务。而上面的异步任务还没执行完，要到下面的大括号才算执行完毕，而下面的同步任务已经在抢占资源了，就会发生死锁。

•         dispatch\_sync(q, ^{

•             NSLog(@"同步任务 %@", [NSThread currentThread]);

•         });

•

•     });

• 串行队列开启同步任务后嵌套同步任务造成死锁

• - (void)serial\_queue\_deadlock1

• {

•     dispatch\_queue\_t q = dispatch\_queue\_create("cn.itcast.gcddemo", DISPATCH\_QUEUE\_SERIAL);

•

•     dispatch\_sync(q, ^{

•         NSLog(@"同步任务 %@", [NSThread currentThread]);

•         // 下面开启同步造成死锁：因为串行队列中线程是有执行顺序的，需要等上面开启的同步任务执行完毕，才会执行下面开启的同步任务。而上面的同步任务还没执行完，要到下面的大括号才算执行完毕，而下面的同步任务已经在抢占资源了，就会发生死锁。

•         dispatch\_sync(q, ^{

•             NSLog(@"同步任务 %@", [NSThread currentThread]);

•         });

•

•     });

•     NSLog(@"同步任务 %@", [NSThread currentThread]);

• }

• 串行队列开启同步任务后嵌套异步任务不造成死锁

网络：

PUT方法

    // PUT

    //    1) 文件大小无限制

    //    2) 可以覆盖文件

    // POST

    //    1) 通常有限制2M

    //    2) 新建文件,不能重名

BASE 64是网络传输中最常用的编码格式 - 用来将二进制的数据编码成字符串的编码方式

     BASE 64的用法:

     1> 能够编码,能够解码

     2> 被很多的加密算法作为基础算法

Session,全局单例(我们能够给全局的session设置代理吗?如果不能为什么?)

    // sharedSession是全局共享的,因此如果要设置代理,需要单独实例化一个Session

NSURLSessionConfiguration(会话配置)

     defaultSessionConfiguration;       // 磁盘缓存,适用于大的文件上传下载

     ephemeralSessionConfiguration;     // 内存缓存,适用于小的文件交互,GET一个头像

     backgroundSessionConfiguration:(NSString \*)identifier; // 后台上传和下载

下载的位置,沙盒中tmp目录中的临时文件,会被及时删除

document       备份,下载的文件不能放在此文件夹中

cache          缓存的,不备份,重新启动不会被清空,如果缓存内容过多,可以考虑新建一条线程检查缓存目录中的文件大小,自动清理缓存,给用户节省控件

tmp            临时,不备份,不缓存,重新启动iPhone,会自动清空

直接通过文件名就可以加载图像,图像会常驻内存,具体的销毁有系统负责

// [UIImage imageNamed:@"”];

// 从网络下载下来的是二进制数据

   NSData \*data = [NSData dataWithContentsOfURL:location];

/ 这种方式的图像会自动释放,不占据内存,也不需要放在临时文件夹中缓存

// 如果用户需要,可以提供一个功能,保存到用户的相册即可

UIImage \*image = [UIImage imageWithData:data];

要使用常规的AFN网络访问

 1. AFHTTPRequestOperationManager \*manager = [AFHTTPRequestOperationManager manager];

    所有的网络请求,均有manager发起

 2. 需要注意的是,默认提交请求的数据是二进制的,返回格式是JSON

    1> 如果提交数据是JSON的,需要将请求格式设置为AFJSONRequestSerializer

    2> 如果返回格式不是JSON的,

 3. 请求格式

     AFHTTPRequestSerializer            二进制格式

     AFJSONRequestSerializer            JSON

     AFPropertyListRequestSerializer    PList(是一种特殊的XML,解析起来相对容易)

 4. 返回格式

     AFHTTPResponseSerializer           二进制格式

     AFJSONResponseSerializer           JSON

     AFXMLParserResponseSerializer      XML,只能返回XMLParser,还需要自己通过代理方法解析

     AFXMLDocumentResponseSerializer (Mac OS X)

     AFPropertyListResponseSerializer   PList

     AFImageResponseSerializer          Image

     AFCompoundResponseSerializer       组合

所有网络请求,统一使用异步请求!

在今后的开发中,如果使用简单的get/head请求,可以用NSURLConnction异步方法

GET查/POST增/PUT改/DELETE删/HEAD

GET

1> URL

2> NSURLRequest

3> NSURLConnction 异步

POST

1> URL

2> NSMutableURLRequest

.httpMethod = @"POST";

str 从 firebug直接粘贴,或者自己写

变量名1=数值1&变量名2=数值2

.httpData = [str dataUsingEncoding:NSUTF8StringEncoding];

3> NSURLConnction 异步

Connection

// 1> 登录完成之前,不能做后续工作!

// 2> 登录进行中,可以允许用户干点别的会更好!

// 3> 让登录操作在其他线程中进行,就不会阻塞主线程的工作

// 4> 结论:登陆也是异步访问,中间需要阻塞住

数据解析：

从iOS 5开始，使用NSJSONSerialization对JSON解析

反序列化

[NSJSONSerialization JSONObjectWithData:data options:0 error:NULL];

序列化

[NSJSONSerialization dataWithJSONObject:array options:0 error:NULL];

1> PUT方法

// PUT

// 1) 文件大小无限制

// 2) 可以覆盖文件

// POST

// 1) 通常有限制2M

// 2) 新建文件,不能重名

// 2> 安全认证

// admin:123456

// result base64编码

// Basic result

/\*\*

BASE 64是网络传输中最常用的编码格式 - 用来将二进制的数据编码成字符串的编码方式

BASE 64的用法:

1> 能够编码,能够解码

2> 被很多的加密算法作为基础算法

3. Session,全局单例(我们能够给全局的session设置代理吗?如果不能为什么?)

// sharedSession是全局共享的,因此如果要设置代理,需要单独实例化一个Session

/\*\*

NSURLSessionConfiguration(会话配置)

defaultSessionConfiguration; // 磁盘缓存,适用于大的文件上传下载

ephemeralSessionConfiguration; // 内存缓存,适用于小的文件交互,GET一个头像

backgroundSessionConfiguration:(NSString \*)identifier; // 后台上传和下载

/\*\*

AFNetworkReachabilityStatusUnknown = -1, // 未知

AFNetworkReachabilityStatusNotReachable = 0, // 无连接

AFNetworkReachabilityStatusReachableViaWWAN = 1, // 3G 花钱

AFNetworkReachabilityStatusReachableViaWiFi = 2, // 局域网络,不花钱

\*/

// 如果要检测网络状态的变化,必须用检测管理器的单例的startMonitoring

[[AFNetworkReachabilityManager sharedManager] startMonitoring];

// 检测网络连接的单例,网络变化时的回调方法

[[AFNetworkReachabilityManager sharedManager] setReachabilityStatusChangeBlock:^(AFNetworkReachabilityStatus status) {

NSLog(@"%d", status);

}];

音频处理

依赖的框架：AVFoundation、AudioToolbox框架

播放长音乐：AVAudioPlayer

播放短音效：加载音频文件生成SystemSoundID

录音：AVAudioRecord

较为底层、高级的音频\视频处理

CoreAudio、CoreVideo框架

XMPP工作原理

节点连接到服务器

服务器利用本地目录系统中的证书对其认证

节点指定目标地址，让服务器告知目标状态

服务器查找、连接并进行相互认证

节点之间进行交互

XMPP框架提供的主要扩展功能

XMPPReconnect：如果意外中断，自动重连XMPP流

XMPPRoster：标准的XMPP花名册

XMPPRoom：提供多人聊天支持

XMPPPubSub：提供公共订阅支持

通信类别及公共XML属性

使用XMPP的实时消息传递系统包含三大通信类别：

消息传递，其中数据在有关各方之间传输

联机状态，允许用户广播其在线状态和可用性

信息/查询请求，它允许XMPP实体发起请求并从另一个实体接收响应

以上三种类型的XMPP节都拥有以下公共属性：

from：源XMPP实体的JID

to：目标接收者的JID

id：当前对话的可选标识符

type：节的可选子类型

xml:lang：如果内容是人们可读的，则为消息语言的描述

XMPP核心文件

XMPPStream：是开发过程中最主要交互的类，所有扩展和自定义代码均要基于此类进行

XMPPParser：供XMPPStream解析使用

XMPPJID：提供了一个不可变JID的实现，遵守NSCopying协议和NSCoding协议

XMPPElement：以下三个XMPP元素的基类

XMPPIQ :请求

XMPPMessage :消息

XMPPPresence :出席

XMPPModule：开发XMPP扩展时使用

XMPPLogging：XMPP的日志框架

XMPPInternal：整个XMPP框架内部使用的核心和高级底层内容

XMPP框架常用扩展

XEP-0045: 多用户聊天

XEP-0060: 发布-订阅

XEP-0065: SOCKS5字节流

XEP-0085: 聊天状态通知

XEP-0096: 文件传输

XEP-0172: 用户昵称

XEP-0184: 消息送达

CoreDataStorage: 数据存储

Reconnect：重新连接

Roster：花名册

XMPP一栏的框架

CocoaLumberjack：日志框架

CocoaAsyncSocket：底层网络框架，实现异步Socket网络通讯

需要添加CFNetwork&Security框架依赖

KissXML：XML解析框架

需要添加libxml2.dylib框架依赖

需要指定如下编译选项：

OTHER\_LDFLAGS = -lxml2

HEADER\_SEARCH\_PATHS = /usr/include/libxml2

libidn

网络面试：

TCP：安全的协议，能保证数据顺序和正确性，服务器和客户端能随时互发数据。如果服务器要主动发送数据给客户端，可以用这个协议

UDP：非安全的协议，容易丢失数据，一般用于联机对战的游戏

XMPP：基于XML通讯的协议，基于TCP发送XML数据，一般用于即时通讯（比如QQ、微信）

HTTP：一般用于非实时连接的请求，只有客户端主动向服务器发送请求时，服务器才能返回数据给客户端

SOCKET：套接口，可以使用TCP/UDP/XMPP通讯

200 表示是一个正确的请求，206表示请求只加载了一部分，404表示网络请求的页面不存在；状态编码，503表示服务器超时，400请求出错

断点续传：客户端软件断点续传值的时在下载或者上传时，将下载或者上传的文件认为的划分成几个部分，每个部分一个线程进行上传或者下载的，如果网络异常，可以从上传或者下载的部分重新上传或者下载未上传下载的部分，提高速度，节省时间。

创建串行队列 加入异步任务

生成文件名,用该文件名和存放路径 生成文件路径

发送网络请求获取待生成文件文件大小

设定每次下载的字节数,循环下载 (循环判断是剩余字节是否大于循环下载字节)

发送请求时设定http头的range范围, 根据每次循环 fromB 和 toB 来设定

每次下载成功返回的数据写入到之前设定好的文件中

Socket连接与HTTP连接

由于通常情况下Socket连接就是TCP连接，因此Socket连接一旦建立，通信双方即可开始相互发送数据内容，直到双方连接断开。但在实际网络应用中，客户端到服务器之间的通信往往需要穿越多个中间节点，例如路由器、网关、防火墙等，大部分防火墙默认会关闭长时间处于非活跃状态的连接而导致 Socket 连接断连，因此需要通过轮询告诉网络，该连接处于活跃状态。

而HTTP连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是Socket连接，服务器就可以直接将数据传送给客户端；若双方建立的是HTTP连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端，因此，客户端定时向服务器端发送连接请求，不仅可以保持在线，同时也是在“询问”服务器是否有新的数据，如果有就将数据传给客户端。

http基于socket做出来的，所有的网络功能都是基于socket做出来的，比如：即时通讯，ftp

//收到内存警告会自动调用

- (void)applicationDidReceiveMemoryWarning:(UIApplication \*)application

文件存储：

Plist文件存储

// 1.获得沙盒根路径

NSString \*home = NSHomeDirectory();

// 2.document路径

NSString \*docPath = [home stringByAppendingPathComponent:@"Documents"];

// 3.新建数据

NSArray \*data = @[@"jack", @10, @"ffdsf"];

// 4.将数据写入沙盒document 目录的data.plist文件中

NSString \*filepath = [docPath stringByAppendingPathComponent:@"data.plist"];

[data writeToFile:filepath atomically:YES];

// 5.读取数据

NSArray \*data = [NSArray arrayWithContentsOfFile:filepath];

NSLog(@"%@", data);

偏好设置存储

// 1.利用NSUserDefaults,就能直接访问软件的偏好设置(Library/Preferences)

NSUserDefaults \*defaults = [NSUserDefaults standardUserDefaults];

// 2.存储数据

[defaults setObject:@"mj" forKey:@"account"];

[defaults setObject:@"123" forKey:@"pwd"];

[defaults setInteger:10 forKey:@"age"];

[defaults setBool:YES forKey:@"auto\_login"];

// 3.立刻同步

[defaults synchronize];

// 4.读取数据

NSString \*account = [defaults objectForKey:@"account"];

BOOL autoLogin = [defaults boolForKey:@"auto\_login”];

NSKeyedArchiver 和 NSKeyedUnarchiver

MJStudent实现 <NSCoding > 协议的方法

/\*\*

\* 将某个对象写入文件时会调用

\* 在这个方法中说清楚哪些属性需要存储

\*/

- (void)encodeWithCoder:(NSCoder \*)encoder

{

[encoder encodeObject:self.no forKey:@"no"];

[encoder encodeInt:self.age forKey:@"age"];

[encoder encodeDouble:self.height forKey:@"height"];

}

/\*\*

\* 从文件中解析对象时会调用

\* 在这个方法中说清楚哪些属性需要存储

\*/

- (id)initWithCoder:(NSCoder \*)decoder

{

if (self = [super init]) {

// 读取文件的内容

self.no = [decoder decodeObjectForKey:@"no"];

self.age = [decoder decodeIntForKey:@"age"];

self.height = [decoder decodeDoubleForKey:@"height"];

}

return self;

}

MJStudent \*stu1 = [[MJStudent alloc] init];

stu1.no = @"42343254";

stu1.age = 20;

stu1.height = 1.55;

MJStudent \*stu2 = [[MJStudent alloc] init];

stu2.no = @"42343254";

stu2.age = 20;

stu2.height = 1.55;

// 新建一块可变数据区

NSMutableData \*data = [NSMutableData data];

// 将数据区连接到一个NSKeyedArchiver对象

NSKeyedArchiver \*archiver = [[NSKeyedArchiver alloc] initForWritingWithMutableData:data];

// 开始存档对象，存档的数据都会存储到NSMutableData中

[archiver encodeObject:stu1 forKey:@"person1"];

[archiver encodeObject:stu2 forKey:@"person2"];

// 存档完毕(一定要调用这个方法)

[archiver finishEncoding];

// 将存档的数据写入文件

[data writeToFile:path atomically:YES]

// 从文件中读取数据

NSData \*data = [NSData dataWithContentsOfFile:path];

// 根据数据，解析成一个NSKeyedUnarchiver对象

NSKeyedUnarchiver \*unarchiver = [[NSKeyedUnarchiver alloc] initForReadingWithData:data];

MJStudent \*stu11 = [unarchiver decodeObjectForKey:@"stu1"];

MJStudent \*stu22= [unarchiver decodeObjectForKey:@"stu2"];

// 恢复完毕

[unarchiver finishDecoding];

如果父类也遵守了NSCoding协议，请注意：应该在encodeWithCoder:方法中加上一句

[super encodeWithCode:encode];确保继承的实例变量也能被编码，即也能被归档

应该在initWithCoder:方法中加上一句self = [super initWithCoder:decoder];

确保继承的实例变量也能被解码，即也能被恢复

利用解归档实现深复制

通过解归档, 被归档的对象 ,再被解档后,内存地址已经不一样了,即实现了深复制

数据库的线程安全：

如果是coredata，需要将context放在主线程上；因为context统一负责数据库的读写操作

1. 全局队列与并行队列的区别

dispatch\_queue\_t q =

dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0);

1> 不需要创建，直接GET就能用

2> 两个队列的执行效果相同

3> 全局队列没有名称，调试时，无法确认准确队列

4> 全局队列有高中默认优先级

2. 并行队列

dispatch\_queue\_t q =

dispatch\_queue\_create("ftxbird", DISPATCH\_QUEUE\_CONCURRENT);

3. 串行队列

dispatch\_queue\_t t = dispatch\_queue\_create("ftxbird",DISPATCH\_QUEUE\_SERIAL);

4. 开发中，跟踪当前线程

[NSThread currentThread]

5. 并行队列的任务嵌套例子

    dispatch\_queue\_t q = dispatch\_queue\_create("ftxbird", DISPATCH\_QUEUE\_CONCURRENT);

    // 任务嵌套

        dispatch\_sync(q, ^{

          NSLog(@"1 %@", [NSThread currentThread]);

            dispatch\_sync(q, ^{

                NSLog(@"2 %@", [NSThread currentThread]);

                dispatch\_sync(q, ^{

                    NSLog(@"3 %@", [NSThread currentThread]);

                });

            });

            dispatch\_async(q, ^{

                NSLog(@"4 %@", [NSThread currentThread]);

            });

        NSLog(@"5 %@", [NSThread currentThread]);

        });

// 运行结果是: 12345 或12354

6. 主队列(线程)

1>每一个应用程序都只有一个主线程

2>所有UI的更新工作，都必须在主线程上执行！

3>主线程是一直工作的，而且除非将程序杀掉，否则主线程的工作永远不会结束！

dispatch\_queue\_t q = dispatch\_get\_main\_queue();

7.在主队列上更新UI的例子

    //创建代码块

    void (^TaskOne)(void) = ^(void)

    {

        NSLog(@"Current thread = %@", [NSThread currentThread]);

        NSLog(@"Main thread = %@", [NSThread mainThread]);

        [[[UIAlertView alloc] initWithTitle:@"GCD"

                                    message:@"Great Center Dispatcher"

                                   delegate:nil

                          cancelButtonTitle:@"OK"

                          otherButtonTitles:nil, nil] show];

    };

    //取得分发队列

    dispatch\_queue\_t mainQueue = dispatch\_get\_main\_queue();

    //提交任务

    dispatch\_async(mainQueue, TaskOne);

}

    //简便写法

   dispatch\_async( dispatch\_get\_main\_queue(), ^(void)

    {

       NSLog(@"Current thread = %@", [NSThread currentThread]);

       NSLog(@"Main thread = %@", [NSThread mainThread]);

       [[[UIAlertView alloc] initWithTitle:@"GCD"

                                   message:@"Great Center Dispatcher"

                                  delegate:nil

                         cancelButtonTitle:@"OK"

                         otherButtonTitles:nil, nil] show];

    });

//输出结果

//2014-05-02 20:34:27.872 serirl[835:60b] Current thread = <NSThread: 0x8e24540>{name = (null), num = 1}

//2014-05-02 20:34:27.873 serirl[835:60b] Main thread = <NSThread: 0x8e24540>{name = (null), num = 1}

NSOperation 多线程技术

8. NSBlockOperation 简单使用

//开发中一般给自定义队列定义为属性

@property (nonatomic, strong) NSOperationQueue \*myQueue;

self.myQueue = [[NSOperationQueue alloc] init];

1>在自定义队列

 NSBlockOperation \*block = [NSBlockOperation blockOperationWithBlock:^{

        NSLog(@"%@", [NSThread currentThread]);

    }];

  所有的自定义队列，都是在子线程中运行.

  [self.myQueue addOperation:block];

  或者:

   [self.myQueue addOperationWithBlock:^{

            NSLog(@"%@", [NSThread currentThread]);

        }];

2>在主队列中执行

    [[NSOperationQueue mainQueue] addOperationWithBlock:^{

        NSLog(@"%@", [NSThread currentThread]);

    }];

3> NSBlockOperation 的使用例子

  NSBlockOperation \*op1 = [NSBlockOperation blockOperationWithBlock:^{

        NSLog(@"下载图片 %@", [NSThread currentThread]);

    }];

    NSBlockOperation \*op2 = [NSBlockOperation blockOperationWithBlock:^{

        NSLog(@"修饰图片 %@", [NSThread currentThread]);

    }];

    NSBlockOperation \*op3 = [NSBlockOperation blockOperationWithBlock:^{

        NSLog(@"保存图片 %@", [NSThread currentThread]);

    }];

    NSBlockOperation \*op4 = [NSBlockOperation blockOperationWithBlock:^{

        NSLog(@"更新UI %@", [NSThread currentThread]);

    }];

    // 设定执行顺序, Dependency依赖，可能会开多个，但不会太多

    // 依赖关系是可以跨队列的！

    [op2 addDependency:op1];

    [op3 addDependency:op2];

    [op4 addDependency:op3];

    // GCD是串行队列，异步任务，只会开一个线程

    [self.myQueue addOperation:op1];

    [self.myQueue addOperation:op2];

    [self.myQueue addOperation:op3];

    // 所有UI的更新需要在主线程上进行

    [[NSOperationQueue mainQueue] addOperation:op4];

9. NSInvocationOperation 简单使用

 NSInvocationOperation \*op = [[NSInvocationOperation alloc] initWithTarget:self selector:@selector(demoOp:) object:@"hello op"];

- (void)demoOp:(id)obj

{

    NSLog(@"%@ - %@", [NSThread currentThread], obj);

}

10. performSelectorOnMainThread 方法使用

        // 1> 模拟下载，延时

        [NSThread sleepForTimeInterval:1.0];

        // 2> 设置图像，苹果底层允许使用performSelectorInBackground方法

        // 在后台线程更新UI，强烈不建议大家这么做！

        // YES会阻塞住线程，直到调用方法完成

        // NO不会阻塞线程，会继续执行

  [self performSelectorOnMainThread:@selector(setImage:) withObject:[UIImage imageNamed:imagePath] waitUntilDone:NO];

      // 1. 图像

    - (void)setImage:(UIImage \*)image

      {

        self.imageView.image = image;

        [self.imageView sizeToFit];

      }

11.

提问：代码存在什么问题？如果循环次数非常大，会出现什么问题？应该如何修改？

 // 解决办法1：如果i比较大，可以在for循环之后@autoreleasepool

 // 解决方法2：如果i玩命大，一次循环都会造成

          自动释放池被填满,一次循环就@autoreleasepool

    for (int i = 0; i < 10000000; ++i) {

        @autoreleasepool {

            // \*

            NSString \*str = @"Hello World!";

            // new \*

            str = [str uppercaseString];

            // new \*

            str = [NSString stringWithFormat:@"%@ %d", str, i];

            NSLog(@"%@", str);

        }

    }