Crash 及 APP 性能监控相关

卡顿处理 具体实践流程 (结合各种实际demo的总结)

针对不同的列表样式，使用不同的优化策略

预排版+预渲染+异步绘制

这种主要针对比较复杂的cell，比如富文本较多的类似微博这种列表。

- 首先**预排版**：拿到json之后先计算主要控件的布局，比如头像、内容、按钮等frame， 将其放到内存中。这里计算布局也可以使用Core Text的CTFramesetterRef来进行计算，性能消耗更小：

```
- (CGSize)sizeWithConstrainedToSize:(CGSize)size fromFont:(UIFont *)font1
lineSpace:(float)lineSpace{
    CGFloat minimumLineHeight = font1.pointSize,maximumLineHeight =
minimumLineHeight, linespace = lineSpace;
    CTFontRef font = CTFontCreateWithName((__bridge
CFStringRef)font1.fontName,font1.pointSize,NULL);
    CTLineBreakMode lineBreakMode = kCTLineBreakByWordWrapping;
    //Apply paragraph settings
    CTTextAlignment alignment = kCTLeftTextAlignment;
    CTParagraphStyleRef style =
CTParagraphStyleCreate((CTParagraphStyleSetting[6]){
        {kCTParagraphStyleSpecifierAlignment, sizeof(alignment),
&alignment},

{kCTParagraphStyleSpecifierMinimumLineHeight,sizeof(minimumLineHeight),&mini
mumLineHeight},

{kCTParagraphStyleSpecifierMaximumLineHeight,sizeof(maximumLineHeight),&maxi
mumLineHeight},
        {kCTParagraphStyleSpecifierMaximumLineSpacing, sizeof(linespace),
&linespace},
        {kCTParagraphStyleSpecifierMinimumLineSpacing, sizeof(linespace),
&linespace},

{kCTParagraphStyleSpecifierLineBreakMode,sizeof(CTLineBreakMode),&lineBreakM
ode}
    },6);
    NSDictionary* attributes = [NSDictionary dictionaryWithObjectsAndKeys:
(__bridge id)font,(NSString*)kCTFontAttributeName,(__bridge id)style,
(NSString*)kCTParagraphStyleAttributeName,nil];
    NSMutableAttributedString *string = [[NSMutableAttributedString alloc]
initWithString:self attributes:attributes];
    //    [self clearEmoji:string start:0 font:font1];
    CFAttributedStringRef attributedString = (__bridge
CFAttributedStringRef)string;
```

```
    CTFramesetterRef framesetter =
CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attributedStr
ing);
    CGSize result =
CTFramesetterSuggestFrameSizeWithConstraints(framesetter, CFRangeMake(0,
[string length]), NULL, size, NULL);
    return result;
}
```

- 预渲染：在预排版的过程中，也可以做一些预渲染的部分工作，比如如果头像是圆形，可以使用CAShapeLayer和UIBezierPath先设置好圆角path。

```
UIBezierPath *maskPath = [UIBezierPath bezierPathWithRoundedRect:rect
byRoundingCorners:corners cornerRadii:CGSizeMake(radius, radius)];
CAShapeLayer *maskLayer = [CAShapeLayer layer];
maskLayer.frame = rect;
maskLayer.path = maskPath.CGPath;
self.layer.mask = maskLayer;
```

- 在头像的UIImageView上加了一层薄薄的CALayer灰度图层。CALayer设置为shouldResterize = YES，也就是开启了光栅化，CALayer会被光栅化为bitmap，也可以有效提高性能，但是使用前提是在内容不变的前提下。
- Cell中的用于显示富文本的相关内容使用一个自定义的UIView子类AsyncLayer。AsyncLayer含有text文本内容、文本颜色、字体、字间距、居中样式等信息。然后在setText方法中异步绘制文本。大概的绘制流程就是：
    - 使用异步串行队列开启绘制；
    - 使用UIGraphicsBeginImageContextWithOptions创建Context上下文；
    - 创建CoreText，注意要先转换坐标；
    - 设置好文本样式，生成NSMutableAttributedString；
    - 使用NSMutableAttributedString创建CTFrame；
    - 使用CTFrame在CGContextRef上下文上绘制；CTFrameDraw(frame, context);
    - 获取上下文当中的image，UIGraphicsGetImageFromCurrentImageContext();
    - 回到主线程，给UIView的layer.contents属性进行赋值。
- cell中非富文本相关区域。比如整个背景、固定的头像、点赞这些固定的属性，也可以使用异步绘制的方式，在setModel的时候进行绘制。大概的流程就是：
    - 使用异步串行队列开启绘制；
    - 使用UIGraphicsBeginImageContextWithOptions创建Context上下文；
    - 绘制整个cell区域，及各种小空间；
    - 获取上下文中的Image；
    - 回到主线程，给最底层的layer.contents进行赋值；(注意这里不是直接给cell的layer的contents进行赋值，因为像富文本相关也是添加到cell的layer上面的)
- cell的图片，首先cell的图片就是使用SDWebImage这种方式去设置。其次，要对一些图片做一些特殊的设置，比如降低采样率等。

按需加载

按需加载就是说在滚动停止的时候进行加载。这种方式有个弊端就是滑动的过程中可能会出现空白页。但是也是一种解决滑动时卡顿的方案，像一些课时列表页，它的排列顺序有可能是按章节从小到大的排序的。那么用户一进来就可能会想要滑到最底部，查看最新的课时，那么可以考虑使用这种方案。具体的实现方案：

- 监听scrollview滚动状态，主要是监听scrollViewWillEndDragging、scrollViewDidScroll、scrollViewDidEndDecelerating等函数。在scrollViewWillEndDragging函数中，计算tableview将要显示的范围(一般会上下多加三行cell)，然后将将要显示的indexPath添加进一个待绘制数组里。其次，在scrollViewDidScroll、scrollViewDidEndDecelerating监听滚动状态，设置一个bool值。
- 将cell的绘制方法独立开来，在cellForRowAtIndexPath:方法中，如果当前正在滚动，或者当前待绘制的数组有值切不包含cell，则不进行绘制。否则进行绘制操作。

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NeedLoadNewsCell *cell = [tableView
dequeueReusableCellWithIdentifier:cellId];
    [self drawCell:cell withIndexPath:indexPath];
    cell.delegate = self;   // VC作为Cell视图的代理对象
    return cell;
}

// 按需绘制
- (void)drawCell:(NeedLoadNewsCell *)cell withIndexPath:(NSIndexPath
*)indexPath{
    NewsModel *model = [self.dataSource objectAtIndex:indexPath.row];
    cell.selectionStyle = UITableViewCellSelectionStyleNone;
    [cell clear];
    cell.model = model;
    // 如果当前 cell 不在需要绘制的cell 中,则直接 pass。
    // 注意第一屏肯定可以显示出来，因为第一屏不会触发滚动，待绘制数组里也没有数据，其次滚
动过程中，可保证需要绘制的cell进行绘制。
    if (self.needLoadArray.count > 0 && [self.needLoadArray
indexOfObject:indexPath] == NSNotFound) {
        [cell clear];
        //不在待绘制列表，不绘制。
        return;
    }
    if (_scrollToToping) {
    //正在滚动不绘制
        return;
    }
    //绘制方法
    [cell draw];
}
```

- 在每一次touchBegin函数中将需要绘制的cell清空。
- 在scrollViewDidEndScrollingAnimation和scrollViewDidScrollToTop方法中开始绘制tableview的visibleCells。

延时加载

延时加载主要针对图片，这个SDWebImage也可以进行设置（SDWebImage是可以设置在滚动的时候不加载）； 监听RunLoop的状态，这里和卡顿检测检测的状态就不一样了。这里主要是监听runloop的 DefaultMode下的KCFRunLoopBeforeWaiting。首先列表滑动时是UITrackingRunLoopMode，其次KCFRunLoopBeforeWaiting正好是defaultMode即将进入休眠的一个空闲状态。所以监听DefaultMode的BeforeWaiting状态就正好表示cpu有空闲处理问题。具体实现步骤：

- 1、将cell中图片绘制的代码独立成一个方法。比如：

```
/// 绘制图片
- (void)drawImg {
    if (_model.imgs.count > 0) {
        __block float posX = 0;
        [_model.imgs enumerateObjectsUsingBlock:^(NSString *obj, NSUInteger
idx, BOOL *stop) {
            UIImageView *imgView = [[UIImageView alloc]
initWithFrame:CGRectMake(posX, 0, kImgViewWH, kImgViewWH)];
            [imgView sd_setImageWithURL:[NSURL URLWithString:obj]];
            imgView.layer.cornerRadius = 5;
            imgView.layer.masksToBounds = YES;

            [self.imgListView addSubview:imgView];
            posX += (5 + kImgViewWH);
            if (idx >= 3) {
                *stop = YES;
            }
        }];
    }
}
```

- 2、在tableview或者vc上添加监听方法，监听RunLoop的defaultMode状态下的beforeWaiting状态。

```
// 监听 runloop 状态
- (void)addRunloopObserver {
    // 获取当前 runloop
    //获得当前线程的runloop，因为我们现在操作都是在主线程，这个方法就是得到主线程的
runloop
    CFRunLoopRef runloop = CFRunLoopGetCurrent();

    //定义一个观察者,这是一个结构体
    CFRunLoopObserverContext context = {
        0,
        (__bridge void *)(self),
        &CFRetain,
        &CFRelease,
        NULL
    };
```

```
    // 定义一个观察者
    static CFRunLoopObserverRef defaultModeObsever;
    // 创建观察者
    defaultModeObsever = CFRunLoopObserverCreate(NULL,
                                                 kCFRunLoopBeforeWaiting,
// 观察runloop等待的时候就是处于NSDefaultRunLoopMode模式的时候
                                                 YES,   // 是否重复观察
                                                 NSIntegerMax - 999,
                                                 &Callback, // 回掉方法，就是处于NSDefaultRunLoopMode时候要执行的方法
                                                 &context);

    // 添加当前 RunLoop 的观察者 注意防止DefaultMode中，正好是不在滑动时的mode。
    CFRunLoopAddObserver(runloop, defaultModeObsever, kCFRunLoopDefaultMode);
    //c语言有creat 就需要release
    CFRelease(defaultModeObsever);
}

// 每次 runloop 回调执行代码块
static void Callback(CFRunLoopObserverRef observer, CFRunLoopActivity activity, void *info) {
    DelayLoadImgViewController *vc = (__bridge DelayLoadImgViewController *)(info);  // 这个info就是我们在context里面放的self参数

    if (vc.tasks.count == 0) {
        return;
    }

    BOOL result = NO;
    while (result == NO && vc.tasks.count) {
        NSLog(@"开始执行加载图片总任务数:%d",vc.tasks.count);
        // 取出任务
        RunloopBlock unit = vc.tasks.firstObject;
        // 执行任务
        result = unit();
        // d干掉第一个任务
        [vc.tasks removeObjectAtIndex:0];
    }
}
```

- 3、使用一个tasks装cell的图片绘制任务

```
//cellForRowAtIndexPath函数中：
    DelayLoadImgCell *cell = [tableView dequeueReusableCellWithIdentifier:cellId];
    cell.model = model;
    // 将加载绘制图片操作丢到任务中去
```

```
    [self addTask:^BOOL{
        [cell drawImg];
        return YES;
    }];
    return cell;

// 添加任务
- (void)addTask:(RunloopBlock)unit {
    [self.tasks addObject:unit];
    // 保证之前没有显示出来的任务,不再浪费时间加载
    if (self.tasks.count > self.max) {
        [self.tasks removeObjectAtIndex:0];
    }
}
```

解析一下：这里刚拿到数据的时候，主线程RunLoop就是defaultMode。所以不会影响第一屏数据的绘制。这也是为什么RunLoop可以用来优化UITableView的原因。只不过在滑动的时候，同样会出现图片位置空白的情况。可以结合第一种方案来一起用。

Crash处理

面试常考题：介绍你碰到过的印象较深刻的外网crash，并介绍发现、定位、解决的过程。

常见的Crash及处理

- 找不到方法：unrecognized selector sent to instance： eg：比如说未实现的代理方法；对可变集合使用了copy后，对其进行修改操作。 解决方案：给NSObject写个分类，截获这种未实现的方法：

  ```
  - (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector    {
      if ([self respondsToSelector:aSelector]) {
      //已经实现不做处理
          return [self methodSignatureForSelector:aSelector];
      }
      return [NSMethodSignature signatureWithObjCTypes:"v@:"];
  }
  - (void)forwardInvocation:(NSInvocation *)anInvocation {
      NSLog(@"在 %@ 类中, 调用了没有实现的实例方法: %@
  ",NSStringFromClass([self
  class]),NSStringFromSelector(anInvocation.selector));
  }
  ```

  其次，尽量少使用performselector这种API；

- KCV造成的Crash： eg:给NSObject添加KVC；key为nil；key不存在。一句话：**给不存在的key（包括nil）设置value**； 解决方案：重写类的setValue:forUndinedKey:和valueForUndefinedKey:

```
-(void)setValue:(id)value forUndefinedKey:(NSString *)key{
    NSLog(@"在 %@ 类中给不存在的key设置value",NSStringFromClass([self
class]));
}
-(id)valueForUndefinedKey:(NSString *)key{
    return nil;
}
```

- EXC_BAD_ACCESS: eg:使用没有实现的block；对象未初始化；访问野指针(比如assign、unsafe_unretained修饰对象类型，关联属性修饰使用不对)；地址越界； 解决方案：这类对象大多数是由于操作不当导致的，所以可以利用xcode的一些工具
    - 1、使用XCode，在Debug模式下开启僵尸模式，Release时关闭；
    - 2、使用XCode的Address Sanitizer检查地址访问越界；
    - 3、对象属性修改方式要使用正确；
    - 4、使用block要先做判断；
- KVO造成的Crash： eg:观察者是局部变量；被观察者是局部变量；没有实现observeValueForKeyPath:；重复移除观察者；另外需要注意一个不会崩溃的现象：如果重复添加观察者，不会导致崩溃，但是一次改变会被观察多次； 解决方案：
    - 1、尽可能合理使用；
    - 2、add和remove一定要成对出现；
- 集合类Crash： eg:数组越界；向数组添加nil对象；在数组遍历时进行移除操作；字典使用setObject:forKey:时，key为nil；字典使用setObject:forKey:时，objct为nil;字典使用setValue:forKey:时，key为nil。这里要注意setValue和setObjct的区别：

```
NSMutableDictionary * dic = [NSMutableDictionary dictionary];
 [dic setObject:nil forKey:@"1"]; //crash
 [dic setValue:nil forKey:@"1"]; // not crash
 [dic setObject:@"1" forKey:nil]; //crash
 [dic setValue:@"1" forKey:nil]; //crash
```

  解决方案：
    - 1、可以给集合类添加category，method swizzling原来方法，判断后再处理；
    - 2、使用可变字典添加元素时，尽可能使用setValue:forKey:
    - 3、NSMutableArray、NSMutableDictionary不是线程安全的，所以在多线程下要保证写操作的原子性。可以使用加锁、信号量、串行队列、dispatch_barrier_async等；或者使用NSCache代替；
- 多线程Crash： eg:子线程更新UI；Dispatch_Group中level比enter次数更多；Dispatch_semaphore使用时重新复制或置空；多线程下非线程安全的可变集合的使用； 解决方案：
    - 1、使用集合时要确保操作的原子性；可以使用加锁、信号量、串行队列、dispatch_barrier_async等；或者使用NSCache代替；
    - 2、熟练使用Dispatch_Group、Dispatch_semaphore等；
    - 3、多线程发送Crash时，会收到SIGSEGV信号，表面视图访问未分配给自己的内存、或视图往没有写权限的地址写数据；

- Socket造成的Crash： 原因：当服务器close一个连接时，若client端接着发数据。根据TCP协议的规定，会收到一个RST响应，client若再往这个服务器发送数据时，系统会发出一个SIGPIPE信号给进程，告诉进程这个连接已断开，不要再写数据了。而根据信号的默认处理规则，SIGPIPE信号的默认执行动作是terminate(终止、退出),所以client会退出。 eg：长连接socket或重定向管道进入后台，没有关闭导致崩溃的解决方法； 解决方案：
    - 1、切换到后台是，关闭长连接和管道，回到前台重新创建；
    - 2、使用signal(SIGPIPE,SIG_IGN)将SIGPIPE设置为SIG_IGN,相当于将SIGPIP交给系统处理，客户端不执行默认操作，即不退出。
- Watch Dog超时造成的Crash： 原因：主线程执行耗时操作，导致主线程被卡超过一定时间。一般异常编码是0x8babf00d，表示应用发送超时而被iOS系统终止。 eg:长期卡顿导致崩溃； 解决方案： 尽可能将耗时操作异步放到后台线程。主线程只负责更新UI和事件响应。
- 后台返回NSNull导致的崩溃： eg:后台返回NSNull，解析完成后，使用时会crash；

```
NSNull *nullStr = [[NSNull alloc] init];
NSMutableDictionary* dic = [NSMutableDictionary dictionary];
[dic setValue:nullStr forKey:@"key"]; //not crash
NSNumber* number = [dic valueForKey:@"key"];// crash 相当于调用了get方法，会报“unrecognized selector”
```

  解决方案：NSNull用于OC对象的占位，一般会作为集合类型的占位元素，给NSNull对象发送消息会crash。
    - 可以使用NullSafe第三方库。

[iOS中常见Crash总结]

Crash捕获

iOS的主要崩溃分为三大类，一类是OC抛出的Exception异常，可以通过注册NSUncaugthExceptionHandler来捕获；一类是Mach异常，比如说野指针访问、线程问题，这一类异常会被转换成Signal信号，可以通过注册signalHandler来捕获。还有一类则是无法使用信号和Exception捕获的，比如像后台任务超时、内存被爆、主线程卡顿超阈值等。

前两类异常的捕获：

这里需要注意的是，避免与Bugly这种工具冲突覆盖掉handler的问题，所以使用之前要先进行判断。再处理完自己的handler之后再抛出去。

```
//一、OC异常处理函数
// OC层中未被捕获的异常，通过注册NSUncaughtExceptionHandler捕获异常信息
void InstallUncaughtExceptionHandler(void) {
//注册
    if(NSGetUncaughtExceptionHandler() != custom_exceptionHandler)
    oldhandler = NSGetUncaughtExceptionHandler();
    uncaught_exception_handler(&custom_exceptionHandler);
}
static void uncaught_exception_handler (NSException *exception) {
    //获取exception的异常堆栈，NSThread也对应一个类方法的callStackSymbols
    NSArray *stackArray = [exception callStackSymbols];
```

```
    //出现异常的原因
    NSString *reason = [exception reason];
    //异常名称
    NSString *exceptionInfo = [NSString stringWithFormat:@"Exception reason:
%@\nException name: %@\nException stack: %@",
                               name,
                               reason,
                               stackArray];
    NSMutableArray *tmpArr = [NSMutableArray arrayWithArray:stackArray];
    [tmpArr insertObject:reason atIndex:0];
    //保存到本地，以便下次启动查看
    [exceptionInfo writeToFile:[NSString
stringWithFormat:@"%@/Documents/error.log",NSHomeDirectory()]
                    atomically:YES
                      encoding:NSUTF8StringEncoding
                         error:nil];
}

//二、Unix标准的signal机制处理函数
// 内存访问错误，重复释放等错误就无能为力了，因为这种错误它抛出的是Signal，所以必须要专门
做Signal处理。 OC中层不能转换的Mach异常，利用Unix标准的signal机制，注册SIGABRT,
SIGBUS, SIGSEGV等信号发生时的处理函数。
void registerSignalHandler(void) {
    signal(SIGSEGV, handleSignalException);
    signal(SIGFPE, handleSignalException);
    signal(SIGBUS, handleSignalException);
    signal(SIGPIPE, handleSignalException); //这个就是上文中socket长连接,服务器关
闭之后系统发出的终止进程的信号，如果想不退出，则可以signal(SIGPIPE, SIG_IGN)，然后重定
向服务器。
    signal(SIGHUP, handleSignalException);
    signal(SIGINT, handleSignalException);
    signal(SIGQUIT, handleSignalException);
    signal(SIGABRT, handleSignalException);
    signal(SIGILL, handleSignalException);
    }

void handleSignalException(int signal) {
    NSMutableString *crashString = [[NSMutableString alloc]init];
    void* callstack[128];
    int i, frames = backtrace(callstack, 128);
    char** traceChar = backtrace_symbols(callstack, frames);
    for (i = 0; i <frames; ++i) {
        [crashString appendFormat:@"%s\n", traceChar[i]];
    }
    NSLog(crashString);
}
```

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    InstallUncaughtExceptionHandler();
    registerSignalHandler();
    return YES;
}
```

对于无法捕获的崩溃怎么处理

- 后台崩溃： 当程序被退到后台后，只有几秒的时间可以执行代码，接着会被挂起，挂起后会暂停所有线程。但是如果是数据读写的线程则无法暂停只能被中断，中断的话，系统会主动杀掉APP，而且中断时数据容易被损坏，。 APP退到后台后，默认是使用Background Task方式，就是系统提供了beginBackgroundTaskWithExpirationHandler方法来延长后台执行时间，可以给到3分钟左右时间去处理退到后台还需处理的任务。3分钟未执行完成的话，还是会被杀掉。
    - 如何避免呢？对于要在后台处理的数据要严格把控大小，太大的数据可以考虑下次启动的时候处理。
    - 怎么监控呢？在Background Task里进行计时，如果时间接近3分钟，任务还在执行，那么就可以判断它即将后台崩溃，然后记录下内容，进行上报。
- 内存被爆 和 主线程超时： 内存被爆和主线程超时，都是由于Watch Dog检测到超时，而向系统杀掉导致的。那这类监控和后台崩溃一样，需要先找到它的阈值，然后临近阈值时进行收集和上报。

各种检测

子线程UI检测

原理：Hook UIView的三个必须在主线程操作的绘制方法：setNeedsDisplay、setNeedsLayout、setNeedsDisplayRect。然后判断他们是否在子线程中操作，如果是在子线程中进行的话，打印出当前代码调用堆栈。

帧率监测

原理：帧率FPS检测主要是检测APP的界面卡顿，判断流畅性。通常的做法是基于CADisplayLin做FPS计算，CADisplayLink是Core Animation提供的一个类似NSTimer的定时器，它会在屏幕每次刷新回调一次，所以它也是以runloop的帧率为标准。所以只需统计每秒方法执行的次数，次数/时间就可以得出帧率了。但是它无法真正定位到性能。

```
-(void)starRecord{
    if(_link) {
        _link.paused = NO;
    }else{
        _link = [CADisplayLink displayLinkWithTarget:self
selector:@selector(trigger:)];
        [_link addToRunLoop:[NSRunLoop mainRunLoop]
forMode:NSRunLoopCommonModes];
    }
}
- (void)trigger:(CADisplayLink * link) {
    if ( lastTime == 0 ){
```

```
        lastTime = link.timestamp;
        return;
    }

    count ++;
    NSTimeInterval delta = link.timestamp - lastTime;
    if (delta < 1) return;
    lastTime = link.timestamp;
    CGFloat fps = count / delta;
    count = 0;
}
```

CPU使用监测

原理：CPU长时间处于高消耗的状态，会使手机发热，耗电量加剧，导致APP产生卡顿。所以要对APP的CPU使用进行监测； 方案就是：使用task_threads函数，获取当前APP所有的线程列表，然后遍历每一个线程，通过thread_info函数获取每一个非闲置线程的cpu使用。

```
+ (CGFloat)cpuUsageForApp {
    // mach_port_t 类似的指针数组，用于存放从线程
    thread_array_t         thread_list;
    //unsigned int 存放获取的线程数
    mach_msg_type_number_t thread_count;
    thread_info_data_t     thinfo;
    mach_msg_type_number_t thread_info_count;
    /*
    线程基本信息：thread_basic_info_t
    struct thread_basic_info {
       time_value_t    user_time;  //用户运行时长
       time_value_t    system_time;  //系统运行时长
       integer_t       cpu_usage;  //cpu使用率，应该是指占整体的百分百
       policy_t        policy;     //调度策略
       integer_t       run_state;   //运行状态
       integer_t       flags;        //各种标记
       integer_t       suspend_count;  //暂停线程时的计数，不知道有啥用
       integer_t       sleep_time;    //休眠时长
    };
    */
    thread_basic_info_t basic_info_th;

    /* get threads in the task
    获取当前进程中 线程列表
    mach_task_self() 返回 mach_port_t 类型，应该是指当前进程
    c语言中引用一般是用于做返回值的，所以这个函数的作用就是从当前进程中获取所有线程数组，及线程数。返回是否成功的Bool值（C语言中非0就是true）。
    */
    kern_return_t kr = task_threads(mach_task_self(), &thread_list,
&thread_count);
    if (kr != KERN_SUCCESS)
```

```
        return -1;
    float tot_cpu = 0;
    for (int j = 0; j < thread_count; j++) {
        thread_info_count = THREAD_INFO_MAX;
        //thread_info用来获取当前这个线程的具体信息。
        kr = thread_info(thread_list[j], THREAD_BASIC_INFO,
                         (thread_info_t)thinfo, &thread_info_count);
        if (kr != KERN_SUCCESS)
            return -1;
        basic_info_th = (thread_basic_info_t)thinfo;
        //这个与操作就是判断线程是否是一个空闲线程
        if (!(basic_info_th->flags & TH_FLAGS_IDLE)) {
            //宏定义TH_USAGE_SCALE返回CPU处理总频率
            tot_cpu += basic_info_th->cpu_usage / (float)TH_USAGE_SCALE;
            //如果要获取线程堆栈应该放在这里。
            ... ...
        }
    }

    // 注意方法最后要调用 vm_deallocate，防止出现内存泄漏
    kr = vm_deallocate(mach_task_self(), (vm_offset_t)thread_list,
thread_count * sizeof(thread_t));
    assert(kr == KERN_SUCCESS);
    return tot_cpu;
}
```

内存消耗监测：

内存消耗监测与CPU使用监测一样的，通过使用task_info来获取进程的虚拟信息，然后获取到phys_footprint。

```
-(NSInterger)useMemoryForApp {
    task_vm_info_data_t vmInfo;
    mach_msg_type_number_t count = TASK_VM_INFO_COUNT;
    kern_return_t kernelReturn = task_info(mach_task_self(), TASK_VM_INFO,
(task_info_t) &vmInfo, &count);
    if(kernelReturn == KERN_SUCCESS)
    {
        int64_t memoryUsageInByte = (int64_t) vmInfo.phys_footprint;
        return (NSInteger)(memoryUsageInByte/1024/1024);
    }
    else
    {
        return -1;
    }
}
```

监测卡顿：

一种检测卡顿的方法是：重写一个NSThread的子类，设置一个时间间隔和最大Watch Dog时间阈值。然后重写她的main函数，在函数里开启一个while循环，然后在时间间隔内查看主线程是否能够及时处理事件，如果主线程能及时处理消息，则说明不卡顿。如果不能处理，则获取主线程的线程堆栈，然后在主线程操作信号量发出之后或者超过最大Watch Dog时间阈值，则将获取到的线程堆栈及卡顿时长进行记录，视为一次卡顿，如果超过一个watch dog阈值则可以视为一次卡顿崩溃，先暂时存起来，如果后续有主线程的信号过来，则把假定的Watch Dog崩溃日志去除。

```
- (void)main {
  //在main函数里，只要没有取消当前子线程，就while循环，查看主线程是否可以响应事件。每次循环都sleep一个阈值的时间。
      while (!self.cancelled) {
          printf("\n");
          //查看线程是否活跃
          if (_isApplicationInActive) {
              //标志位，用于查看主线程是否有处理的标志
              self.mainThreadBlock = YES;
              //主线程堆栈信息
              self.reportInfo = @"";
              self.startTimeValue = [[NSDate date] timeIntervalSince1970];
              printf("1开始查看\n");
              dispatch_async(dispatch_get_main_queue(), ^{
                  self.mainThreadBlock = NO;
                  printf("2执行了主线程，发出信号\n");
                  verifyReport();
                  //如果主线程能够响应，则对信号量进行加一
                  dispatch_semaphore_signal(self.semaphore);
              });
              //阻塞当前线程threshold秒
              [NSThread sleepForTimeInterval:self.threshold];
              printf("3sleep醒来\n");
              if (self.isMainThreadBlock) {
                  printf("4主线程未来得急执行\n");
                  //如果发生了卡顿，则在threshold秒后，上面主线程肯定是没有执行的。
                  self.reportInfo = [DoraemonBacktraceLogger doraemon_backtraceOfMainThread]; //这里包含堆栈的查看信息
              }
              //如果信号量小于等于0，则阻塞，如果大于0，则先对信号量减一，再执行block操作。或者到最大卡顿崩溃时间之后自动执行，下面这个300可以视为watch Dog的查杀时间
              dispatch_semaphore_wait(self.semaphore, dispatch_time(DISPATCH_TIME_NOW, 300.0 * NSEC_PER_SEC));
              {
                  //卡顿超时情况;
                  if self.isMainThreadBlock {
                      //说明是超过了300.0秒
                      printf("5超过最大Watch Dog时间阈值：%f\n",[[NSDate date] timeIntervalSince1970]);
                  }else{
                      printf("5刚收到信号：%f\n",[[NSDate date] timeIntervalSince1970]);
```

```
                    }
                    verifyReport();
                }
                //如果不卡顿，执行顺序是1、2、3、5；如果卡顿了，则执行顺序是1、3、4、2、
5。主线程卡顿多长时间，wait就等待多长时间，或者主线程卡顿的时间超过了300秒。等主线程反应过
来之后，对信号量进行加1，则就可以走之后的流程了。但是这也存在一个问题，如果是卡顿太久，导致
了崩溃呢！这里就收不到信号了。
            } else {
                //阻塞当前线程threshold秒
                [NSThread sleepForTimeInterval:self.threshold];
            }
        }
    }
```

另一种方法是：就是根据RunLoop的事件循环状态，因为RunLoop大部分处理的事件是在KCFRunLoopBeforeSources和KCFRunLoopBeforeWaiting之间以及KCFRunLoopAfterWaiting之后。所以开启一个子线程，监听RunLoop的状态，如果长期处于KCFRunLoopBeforeSources或KCFRunLoopAfterWaiting状态，则可以判断为卡顿状态。大概的模拟操作就是：

```
// 创建子线程监控
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    // 子线程开启一个持续的 loop 用来进行监控
    while (YES) {
        long semaphoreWait = dispatch_semaphore_wait(dispatchSemaphore,
dispatch_time(DISPATCH_TIME_NOW, 3 * NSEC_PER_SEC));
        if (semaphoreWait != 0) {
            if (!runLoopObserver) {
                timeoutCount = 0;
                dispatchSemaphore = 0;
                runLoopActivity = 0;
                return;
            }
            //BeforeSources 和 AfterWaiting 这两个状态能够检测到是否卡顿
            if (runLoopActivity == kCFRunLoopBeforeSources ||
runLoopActivity == kCFRunLoopAfterWaiting) {
                // 将堆栈信息上报服务器的代码放到这里
            } //end activity
        }// end semaphore wait
        timeoutCount = 0;
    }// end while
});
```

流量监控

iOS的类NSURLProtocol可以拦截NSURLConnect、NSURLSession、UIWebView中的所有网络请求，获取每一次请求的request和response对象。但是这个类没法拦截TCP请求。可以使用URLProtocol做如下事情：

- 重定向网络请求
- 忽略网络请求，使用本地缓存
- 自定义网络请求的返回结果
- 一些全局的网络请求设置

NSURLProtocol的使用

堆栈收集与分析。

堆栈获取的话，一种是**直接使用系统函数**，类似使用signal捕获的方式获取堆栈，但是它不太好定位到具体函数，堆栈信息符号化不好处理。还有一种就是直接使用**PLCrashReporter开源库**。PLCrashReporter开源库能够定位到具体代码位置，性能消耗也不是很大。

iOS Crash的捕获知识

crash崩溃堆栈分析(略) 移动监控体系之技术原理 了解和分析iOS Crash Report iOS crash日志堆栈解析 [iOS崩溃异常捕获](https://juejin.im/post/5a93c9385188257a7b5aba42

Core Text

上述异步绘制中设计到CoreText，所以这里简单介绍一下： 三个类：CTFrameRef: 画布；CTLineRef: 每一行；CTRunRef:每一小段。 每个画布(CTFrameRef)可以包含多行(CTLineRef)，每一行可以包含多个小段(CTRunRef)。 绘制步骤： 首先一般的绘制都是异步绘制，所以基本是在display函数或者drawRect函数中。因为这样才能拿到context

```
    //
    CGContextRef context = UIGraphicsGetCurrentContext();
    //变换坐标
    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);
    //设置绘制的路径
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddRect(path, NULL, self.bounds);
    /创建属性字符串
    NSMutableAttributedString * attStr = [[NSMutableAttributedString alloc]
initWithString:str4];

    //颜色
    [attStr addAttribute:(__bridge NSString
*)kCTForegroundColorAttributeName value:(__bridge id)[UIColor
redColor].CGColor range:NSMakeRange(5, 10)];

    //字体
    UIFont * font = [UIFont systemFontOfSize:25];
    CTFontRef fontRef = CTFontCreateWithName((__bridge
CFStringRef)font.fontName, 25, NULL);
```

```
    [attStr addAttribute:(__bridge NSString *)kCTFontAttributeName value:
(__bridge id)fontRef range:NSMakeRange(20, 10)];

    //空心字
    [attStr addAttribute:(__bridge NSString *)kCTStrokeWidthAttributeName
value:@(3) range:NSMakeRange(36, 5)];
    [attStr addAttribute:(__bridge NSString *)kCTStrokeColorAttributeName
value:(__bridge id)[UIColor blueColor].CGColor range:NSMakeRange(37, 10)];

    //下划线
    [attStr addAttribute:(__bridge NSString *)kCTUnderlineStyleAttributeName
value:@(kCTUnderlineStyleSingle | kCTUnderlinePatternDot)
range:NSMakeRange(45, 15)];

    CTFramesetterRef framesetter =
CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attStr);
    CTFrameRef frame = CTFramesetterCreateFrame(framesetter, CFRangeMake(0,
attStr.length), path, NULL);

    //绘制内容
    CTFrameDraw(frame, context);
```

Core Text编程指南

启动优化

- 冷启动是指APP不在后台，第一次打开
- 热启动是指APP在后台被唤起。

冷启动优化

APP启动主要分为三个阶段：main函数之前、main函数之后、首屏渲染完成：

- main函数之前： 主要工作：加载可执行文件；加载动态链接库；运行时处理(包括类的注册、category注册、selector唯一性检测等)；初始化(+load方法、创建c++静态全局变量)。 所以可以做的事情就包括：
    - 减少动态库的加载，苹果公司建议使用更大的动态库，可以考虑将多个库进行合并，数量上建议是6个非系统动态库；
    - 删减无用的代码和类；
    - +load方法尽量少用，或者将里面的内容挪到其他地方，比如+initialize()；控制c++全局变量的数量；
- main函数之后：

  这个阶段主要是指从main函数开始，到APPDelegatD的didFinishLaunchingWithOptions方法里首屏渲染相关方法的执行；我的理解就是main函数到设置window的root结束。 可以做的事情包括：
    - 不要将各种无必要的类的初始化、配置文件的读写、首屏列表的数据获取和渲染相关等放到这个区间里面。比如说首页列表，通常会直接在viewdidload里做，这样是不好的。

- 首屏渲染完成： 这个阶段就是从首屏渲染到didFinishLaunchingWiithOptions方法作用域结束为止。这个阶段用户已经能看到首屏数据了，所以要注意的就是主线程卡顿相关的问题。

启动耗时检测：略

QQ问题

- 1、说一下哪个项目的技术点比较难。

  原则：所有这些抽象类的问题，都应该尽可能引导到自己熟悉的知识点上。其次切忌模棱两可回答一些不必要的东西。 比如说：最难的项目应该是我最新的那个项目吧，因为在那个项目上，我做了很多列表流畅度优化、卡顿检测、崩溃检测等相关的尝试。

- 2、在项目里都做了哪些卡顿的优化？（理论被断掉了，直接说自己做了哪些处理？） 原则：永远优先直接回答问题，如果是自己比较熟悉的知识点，可以在回答关键点之后，逐一进行解释。其次再看提问者会不会追问一些更深入的东西。 比如：主要对不同的列表做过一些：预排版、预渲染、异步绘制、按需加载、延迟加载、离屏渲染相关的处理。...

- 3、卡顿是怎么做的检测？为什么还要自己写检测工具？那检测的过程中是怎么定位到具体的方法的？ 首先之所以写卡顿的检测工具，主要是用于测试及产品等部门使用。在线的卡顿检测则是更依赖于Bugly来做的。 首先，我的工具里对于卡顿的检测，只做了两方面：

一是FPS帧率检测。PFS是通过CADisplayerLink来计算的。主要就是在主线程创建一个DisplayerLink，然后在回调函数里计算每秒回调的次数。CADIsplayerLink是RunLoop的回调帧率，也就是每一帧会回调一次。如果列表滑动的帧率保持在50到60帧的样子，就可以算是很流畅了。如果低于24帧，则基本有肉眼可见的卡顿。但是FPS只能显示出是否卡顿，但是没法定位到具体的函数。

二就是重写一个NSThread的子类，设置一个时间间隔和最大Watch Dog时间阈值。然后重写她的main函数，在函数里开启一个while循环，然后在时间间隔内查看主线程是否能够及时处理事件，如果主线程能及时处理消息，则说明不卡顿。如果不能处理，则获取主线程的线程堆栈，然后在主线程操作信号量发出之后或者超过最大Watch Dog时间阈值，则将获取到的线程堆栈及卡顿时长进行记录，视为一次卡顿，如果超过一个watch dog阈值则可以视为一次卡顿崩溃，先暂时存起来，如果后续有主线程的信号过来，则把假定的Watch Dog崩溃日志去除。

卡顿定位：刚才说过，FPS是没法定位到具体函数的。具体方法：一是使用Instrument的Timer Profile来分析代码的执行时间，基本可以定位到具体的函数。二就是通过内核线程信息获取线程堆栈，找到对应的函数。

- 4、主线程的卡顿是怎么检测到的？为什么要单独写一个CADisplayLink检测主线程上的帧率，如果是主线程已经卡顿的情况下岂不是会加重卡顿？ 回答如3题的第二种检测方案。只能说主线程放置CADisplayerLink是一种择中的方案。主要用于开发阶段。其次要把问题说出来，即CADisplayLink确实会消耗性能，而且也不是非常准备的一个判断。
- 5、为什么要使用RunLoop做tableView卡顿优化？一开始加载数据的时候岂不是没法使用？（简历里的坑） 使用runloop做tableview的优化主要是根据runloop的工作原理，将一些耗时的操作放到DefaultMode下的beforeWaiting状态下进行。
- 6、在项目里是如何检测Crash的，降低崩溃率都做了哪些事情？ 首先，根据是否可捕获，我们可以将崩溃记录大致分为三种：一是语言层面的Exception，二是mach异常，这两类是可以通过方法捕获到崩溃的，另外一种无法捕获到的崩溃就是后台查杀、内存被爆、主线程卡顿超时等导致的系统级别的查杀。

语言层面的Exception：可以通过注册uncaught_exception_handler，来截获崩溃，获取到Exception的callStackSymbols（异常堆栈列表）、reason、name；

mach异常则会发出Unix标准的Signal信号。所以可以通过注册SignalHandler来截获对应的线程，获取通过线程对应的backtrace_symbols获取堆栈信息。

另外一种则需要一些特殊的方式去避免或者监听崩溃的可能发生：

内存被爆和主线程卡顿，可以通过卡顿检测的最大时间阈值来模拟判断；

后台线程也可以通过监听后台线程运行时长来模拟判断。

- 7、Bugly的检测原理是什么？ Bugly的崩溃和卡顿的原理可能不是非常清楚，但是可以回答一下自己做的卡顿崩溃检测相关知识点。
- 8、是否了解直播的一些底层原理？还是说只停留在SDK的使用过程？ 可以说一下直播的组成结构，已经自己了解的怎么检测直播的卡顿相关知识点。这样即使不会直播的底层实现，也可以有一些拿得出手的东西。

注意： 1、可能是因为一开始紧张，有点语无伦次，所以理论被直接断掉了。然后说实践的时候，自己都觉得这些点很微不足道，没有底气。 2、面试官估计能力很强，他一直就觉得我使用的是系统API之类的，都应该是基础的东西。没什么难的。而且做的一些实践应该是必须要做的。(总之，被鄙视了) 不过最主要的是自己对知识点里的细节没有把握清楚。主要就是CADisplayLink检测主线程、runloop为什么用来优化tableview，是怎么做的。木有说清楚。 3、一定要思路清晰，把自己掌握的东西尽可能地表现出来。

其他看到的关系卡顿和Crash相关的面试问题

- 用户报卡顿，有哪些情况，该如何定位问题?

  离屏渲染、一次性创建大量对象、执行耗时操作等等。通过time Profile查看函数运行时长，然后定位到一些基本的函数。

- 卡顿检测有哪些方案？如果要监控每个函数的耗时如何实现？页面停留时间的检测该如何实现?
- 页面直接卡死，导致无法获取FPS，怎么解决这个问题?
- 为什么有时候FPS很高，但还是感觉卡顿?
- 获取FPS后，怎么定位到具体函数?
- 怎么做图片、数组、字符串的无差别存储，key怎么确定，怎么删除数据。如何保证取出的数据顺序?
- 常见的Crash有哪些？如何对这些Crash堆栈进行收集？如何捕获Crash?
- 捕获的堆栈如何进行符号还原呢？UUID是什么？怎么获取？ UUID 是通用设备唯一标识符，它保证了设备在全世界的唯一性。通过崩溃日志的UUID和dysm文件的UUID比较来获取对应设备的堆栈日志。
- dysm文件是什么，有什么作用?

  Dysm文件是保存函数地址映射信息的中转文件，它里边包含着函数与符合地址的映射，可以用它来做符号还原。

- 如果你的项目中既有Bugly的Crash监控系统，又有自己的监控系统，可能存在什么问题？怎么解决?

可能导致Crash监控出现冲突的问题，解决方案就是在使用自己的监控系统的时候先判断是否有其他的监控，如果有的话，先用一个全局变量保存下来，等自己的监控系统处理完，再把异常抛出去。

```
//Mach-O signal异常
typedef void (*SignalHandler)(int signo, siginfo_t *info, void *context);
//用于保存就的监控Handler
static SignalHandler previousSignalHandler = NULL;
+ (void)installSignalHandler {
    struct sigaction old_action;
    //判断是否已有监控
    sigaction(SIGABRT, NULL, &old_action);
    if (old_action.sa_flags & SA_SIGINFO) {
        //有的话进行保存
        previousSignalHandler = old_action.sa_sigaction;
    }
    //处理自己的监控
    LDAPMSignalRegister(SIGABRT);
    .......
}
static void LDAPMSignalRegister(int signal) {
    struct sigaction action;
    action.sa_sigaction = LDAPMSignalHandler;
    action.sa_flags = SA_NODEFER | SA_SIGINFO;
    sigemptyset(&action.sa_mask);
    sigaction(signal, &action, 0);
}
static void LDAPMSignalHandler(int signal, siginfo_t* info, void* context) {
    //  获取堆栈，收集堆栈
    ........

    LDAPMClearSignalRigister();
    // 处理前者注册的 handler
    if (previousSignalHandler) {
        previousSignalHandler(signal, info, context);
    }
}
```

```
//OC 异常
static NSUncaughtExceptionHandler *previousUncaughtExceptionHandler;
static void LDAPMUncaughtExceptionHandler(NSException *exception) {
    // 获取堆栈，收集堆栈
    // ......
    //  处理前者注册的 handler
    if (previousUncaughtExceptionHandler) {
        previousUncaughtExceptionHandler(exception);
    }
}
```

```
+ (void)installExceptionHandler {
    previousUncaughtExceptionHandler = NSGetUncaughtExceptionHandler();
    NSSetUncaughtExceptionHandler(&LDAPMUncaughtExceptionHandler);
}
```

- 针对无堆栈的Crash，比如out of memory有什么定位思路?

  主要是内存被爆、后台任务被杀、Watch Dog被杀导致的Crash无法获取。只能使用一些特殊的阈值来评估释放可能产生Crash。

- 常见的野指针问题有哪些?

  比如使用assign修饰对象。block回调方法里使用已经dealloc掉的对象。

- dyld是什么？动态链接和静态链接的区别？dyld是什么时候执行的？dyld如何把对应dylib中函数实现链接到另一个库中的?

  dyld 是动态链接加载器，用来加载动态库到内存中。动态链接不会复制到可执行文件中，是运行时动态加载到内存，系统只加载一次，多个程序共用。静态链接会被拷贝进可执行文件中，使用多少次就拷贝多少份。dyld通过symbol符号表来获取对应函数地址。

- FPS相关：

  FPS就是根据CADisplayerLink来计算的。CADisplayerLink是RunLoop回调的帧率，也就是每一帧会回调一次。虽然CADisplayerLink尽可能保持帧率的连续，但是如果因为卡顿出现了丢帧的情况，CADisplayerLink自然就不会被回调。所以FPS只是一个相对流畅度的提现。（注意：未完成绘制的帧会被丢掉，但是runloop并不会丢弃操作，耗时的操作依旧会被执行）

  所以最科学地**检测卡顿的方式**：直接监听RunLoop的BefoeWaiting或Exit通知的时间间隔。当然也有人通过子线程定时地ping主线程，然后根据pong的返回间隔来确定是否卡顿问题。这两个方法的原理都是一样的，就是监听RunLoop本身事件循环的周期，当前最关键的问题是时间阈值的确定。卡顿的阈值一般设置为16.7ms * 40（也就是卡了40帧左右）、卡顿崩溃的阈值大概设为3min。 对于有时候FPS很高，但是仍然卡顿，没怎么明白原因。

质量保障体系相关体系：

- 内存泄漏检测
- 内存大图检测
- 图片主线程解压缩检测
- 卡顿检测
- 帧率检测
- 网络性能检测
- Crash检测
- Abort检测(jetsam杀死进程、watchdog杀死进程、后台崩溃)
- 内存消耗检测
- DNS解析检测
- 启动时间检测

想要处理列表等流程度，是肯定摆脱不了YYKit和AsyncDisplayKit这两个库的 iOS保持界面流程的技巧 AsyncDisplayKit详解 FPS及具体定位 // 了解和分享Crash Report iOS Crash日志堆栈解析

iOS CALayer及UI显示原理与优化