bugly收集和分析

- crash log 种类
  - Mach异常(NSException)
    - Mach是Mac OS和iOS操作系统的微内核核心，Mach异常是指最底层的内核级异常 。每个thread，task都有一个异常端口数组，Mach的部分API暴露给了开发者，开发者可以直接通过Mach API设置thread，task，host的异常端口，来监听捕获Mach异常，抓取Crash事件。所以当APP中产生异常时，最先能监听到异常的就是Mach
    - NSException
      - reason　崩溃原因
      - callStackSymbols　堆栈信息,获取方法执行的类和顺序
      - userInfo　崩溃信息
      - name　崩溃名称
  - Signal信号
    - 最先捕获到异常的Mach在接下来会将所有的异常转换为相应的Unix信号，并投递到出错的线程。之后就可以注册想要监听的signal类型，来捕获信号
    - 信号（英语：Signals）是Unix、类Unix以及其他POSIX兼容的操作系统中进程间通讯的一种有限制的方式。它是一种异步的通知机制，用来提醒进程一个事件已经发生。当一个信号发送给一个进程，操作系统中断了进程正常的控制流程，此时，任何非原子操作都将被中断。如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数
    - 要使用 Signal 时:　#include <sys/signal.h>
    - 监听了SIGSEGV信号：signal(SIGSEGV,mySignalHandler)　当有SIGSEGV信号产生时，就会回调mySignalHandler方法
    - sys/signal 文件内定义了大量的系统信号标识:
      - 进入sys/signal.h系统文件, 发现Signal信号定义31种类型(int 类型枚举):　https://www.jianshu.com/p/3a9dc6bd5e58
      - SIGABRT　程序中止命令中止信号
      - SIGALRM　程序超时信号
      - SIGFPE　程序浮点异常信号
      - SIGILL　程序非法指令信号
      - SIGHUP　程序终端中止信号
      - SIGINT　程序键盘中断信号
      - SIGKILL　程序结束接收中止信号
      - SIGTERM　程序kill中止信号
      - SIGSTOP　程序键盘中止信号
      - SIGSEGV　程序无效内存中止信号
      - SIGBUS　程序内存字节未对齐中止信号
      - SIGPIPE　程序Socket发送失败中止信号
- log
  - 崩溃crash log
    - 立刻上报服务器　使用runloop添加循环,阻塞当前线程,获得一点时间
    - 定量或定时上报服务器　时机：下次启动或者进入后台时
  - 非崩溃
    - 数据收集log
      - 固定功能的日志收集/统计
      - 用户打开页面偏好记录
      - 用户停留时间记录
      - 页面PV(page view),即页面浏览量
    - 问题跟踪log
      - 线上
        - 根据用户反馈,通过后台开启该用户上传日志状态,让用户再复现一遍流程(正常用户默认关闭)
        - 线上账号模拟用户流程分析原因
      - 测试环境　默认开启log打印, 上传
  - 定时清理过期的日志
- code
  - log.h

```
#define NSLog(...)
//宏定义
#define PKLog(frmt,...) [Log logWithLine:__LINE__ method:[NSString stringWithFormat:@"%s", __FUNCTION__] time:[NSDate date] format:[NSString stringWithFormat:frmt, ## __VA_ARGS__]]

@interface Log : NSObject
+ (void)setFileLogOnOrOff:(BOOL)on; //开启状态
+ (void)logWithLine:(NSUInteger)line method:(NSString *)methodName
            time:(NSDate *)timeStr format:(NSString *)format; //log上报
@end
```

  - Log.m
    - 实现日志文件存储,上传等功能
    - 日志默认路径,设置名称　新的日志可以是追加写入这个txt文件里
- 当APP崩溃时:　UncaughtExceptionHandler里面是利用 iOS SDK中提供的现成函数 NSSetUncaughtExceptionHandler 加上 注册想要监听的signal类型，来做异常处理的，通过抛出的Signal，专门对Signal处理
- crash日志的系统方法捕获
  - https://blog.csdn.net/u013602835/article/details/80485331
  - 1. 创建一个类专门处理崩溃日志拦截
  - 2.监听崩溃
    - 在程序启动(didFinishLaunch)调用该方法(监听两种类型):

```
void InstallUncaughtExceptionHandler(void) {
    NSSetUncaughtExceptionHandler(&HandleException); //1.系统的方法
    //2.添加想要监听的signal类型，当发出相应类型的signal时，会回调SignalHandler方法
    signal(SIGABRT, SignalHandler); //解释：调用signal系统方法, 监听SIGABRT信号状态,返回到SignalHandler 方法里
    signal(SIGILL, SignalHandler);
    signal(SIGSEGV, SignalHandler);
    signal(SIGFPE, SignalHandler);
    signal(SIGBUS, SignalHandler);
    signal(SIGPIPE, SignalHandler);
}
```

  - 3.Mach异常方法:
    - Mach异常(NSException) 方法实现:

```
void HandleException(NSException *exception)
{
    // 递增的一个全局计数器，很快很安全，防止并发数太大,此处注意OSAtomicIncrement32的使用，它此处是一个递增的一个全局计数器，效果又快又安全，是为了防止并发数太大出现错误的情况。
    int32_t exceptionCount = OSAtomicIncrement32(&UncaughtExceptionCount);
    if (exceptionCount > UncaughtExceptionMaximum) return;
    // 获取 堆栈信息的数组
    NSArray *callStack = [UncaughtExceptionHandler backtrace];
    // 设置该字典
    NSMutableDictionary *userInfo =
    [NSMutableDictionary dictionaryWithDictionary:[exception userInfo]];
    // 给 堆栈信息 设置 地址 Key
    [userInfo
     setObject:callStack
     forKey:UncaughtExceptionHandlerAddressesKey];

    // 假如崩溃了执行 handleException:，并且传出 NSException(名称,原因, 堆栈信息等等)
    [[[UncaughtExceptionHandler alloc] init] performSelectorOnMainThread:@selector(handleException:) withObject:
[NSException exceptionWithName:[exception name] reason:[exception reason]
      userInfo:userInfo]
     waitUntilDone:YES];
}
```

  - 4.Signal信号方法:

```
void SignalHandler(int signal)
{
    // 递增的一个全局计数器，很快很安全，防止并发数太大
    int32_t exceptionCount = OSAtomicIncrement32(&UncaughtExceptionCount);
    if (exceptionCount > UncaughtExceptionMaximum) return;
    // 设置是哪一种 single 引起的问题
    NSMutableDictionary *userInfo =
    [NSMutableDictionary
     dictionaryWithObject:[NSNumber numberWithInt:signal]
     forKey:UncaughtExceptionHandlerSignalKey];
    // 获取堆栈信息数组
    NSArray *callStack = [UncaughtExceptionHandler backtrace];
    // 写入地址
    [userInfo setObject:callStack forKey:UncaughtExceptionHandlerAddressesKey];
    //  假如崩溃了执行 handleException:，并且传出 NSException
    [[[UncaughtExceptionHandler alloc] init] performSelectorOnMainThread:@selector(handleException:)
     withObject: [NSException exceptionWithName:UncaughtExceptionHandlerSignalExceptionName reason:
[NSString stringWithFormat:
       NSLocalizedString(@"Signal %d was raised.", nil),signal]
      userInfo:[NSDictionary dictionaryWithObject:[NSNumber numberWithInt:signal]
forKey:UncaughtExceptionHandlerSignalKey]] waitUntilDone:YES];
}
```

  - 5.取当前线程调用堆栈的信息，并且转化为字符串数组

```
+ (NSArray *)backtrace
{
    void* callstack[128];
    //  该函数用来获取当前线程调用堆栈的信息,获取的信息将会被存放在buffer中(callstack),它是一个指针数组。
    int frames = backtrace(callstack, 128);
    //  backtrace_symbols将从backtrace函数获取的信息转化为一个字符串数组.
    char **strs = backtrace_symbols(callstack, frames);
    NSMutableArray *backtrace = [NSMutableArray arrayWithCapacity:frames];
    for (
         int i = UncaughtExceptionHandlerSkipAddressCount;
         i < UncaughtExceptionHandlerSkipAddressCount +
UncaughtExceptionHandlerReportAddressCount;
         i++)
    {
        [backtrace addObject:[NSString stringWithUTF8String:strs[i]]];
    }
    free(strs); // 记得free
    return backtrace;
}
```

  - 6.崩溃信息处理

```
- (void)handleException:(NSException *)exception
{
    //1. 打印或弹出框
    //UIAlert *alert = [UIAlert ...
    //2.将信息保存或者上报 ....
    // 3.接到程序崩溃时的信号进行自主处理(通过runloop生成一个死循环, 让APP不要马上退出,给时间去处理日志)
    CFRunLoopRef runLoop = CFRunLoopGetCurrent();
    CFArrayRef allModes = CFRunLoopCopyAllModes(runLoop);
    while (!dismissed)
    {
        //循环 - 生成一个死循环
        for (NSString *mode in (NSArray *)allModes) {
            CFRunLoopRunInMode((CFStringRef)mode, 0.001, false);
        }
    }
    CFRelease(allModes);

    //4. 下面等同于清空之前设置的
    NSSetUncaughtExceptionHandler(NULL);
    signal(SIGABRT, SIG_DFL);
    signal(SIGILL, SIG_DFL);
    signal(SIGSEGV, SIG_DFL);
    signal(SIGFPE, SIG_DFL);
    signal(SIGBUS, SIG_DFL);
    signal(SIGPIPE, SIG_DFL);
    // 杀死 或 唤起
    if ([[exception name] isEqual:UncaughtExceptionHandlerSignalExceptionName]) {
        kill(getpid(), [[[exception userInfo] objectForKey:UncaughtExceptionHandlerSignalKey]
intValue]);
    } else {
        [exception raise];
    }
}
```

- 两个开源框架:
  - https://blog.csdn.net/GGGHub/article/details/71430037
  - 日志收集主要用了两个开源框架来实现：plcrashreporter与CocoaLumberjack。plcrashreporter主要用来崩溃日志收集，CocoaLumberjack用来非崩溃日志收集。
  - CocoaLumberjack：
    - 如果设置DDLogLevelError等级，那么只会看到Error语句
    - 如果设置DDLogLevelWarn等级，那么会看到Error和Warn语句
    - 如果设置DDLogLevelInfo等级，那么会看到Error，Warn，Info语句
    - 如果设置DDLogLevelDebug等级，那么会看到Error，Warn，Info，Debug语句
    - 如果设置DDLogLevelVerbose等级，会看到所有的DDLog语句
    - 如果设置DDLogLevelOff等级，不会看到任何DDLog语句
- 日志分析:
  - bugly文档介绍:　https://bugly.qq.com/docs/user-guide/symbol-configuration-ios/?v=1492997248592#_2
  - 符号表(dSYM文件里)?
    - 符号表是内存地址与函数名、文件名、行号的映射表。符号表元素如下所示：<起始地址> <结束地址> <函数> [<文件名:行号>]
    - 提取dSYM文件的符号表文件并上传到服务器
  - 为什么要配置符号表?　能快速并准确地定位用户APP发生Crash的代码位置, 还原堆栈, 还原问题代码
  - 自动配置：XCode + sh脚本　在Xcode中配置sh脚本, 自动上传符号表(或者dSYM文件), 默认release环境上传
  - 上传　HTTPS接口支持上传dSYM文件（需要压缩成Zip文件）和符号表文件（Symbol）。
  - XCode编译后没有生成dSYM文件?
    - XCode Release编译默认会生成dSYM文件，而Debug编译默认不会生成，对应的Xcode配置如下：
    - XCode -> Build Settings -> Code Generation -> Generate Debug Symbols -> Yes
    - XCode -> Build Settings -> Build Option -> Debug Information Format -> DWARF with dSYM File
  - 开启Bitcode之后需要注意哪些问题?
    - 在点"Upload to App Store"上传到App Store服务器的时候需要选中 符号文件（dSYM文件）的生成选项
    - 通过Xcode找dSYM：　步骤：Xcode 顶部菜单栏 -> Window -> Organizer 窗口 -> Archive -> 右键点击对应归档包，选择Show in Finder操作 -> 显示包内容
  - 如何判断dSYM文件是否与Crash的UUID匹配?
    - 还原Crash堆栈时，需要根据UUID来匹配符号表文件，因此只有上传的符号表文件的UUID与Crash对应APP的UUID一致时，才能准确地对堆栈进行还原。
    - 查看符号表文件的UUID？
      - 通过命令查看UUID：　xcrun dwarfdump --uuid <dSYM文件>
      - 通过符号表文件查看UUID：
        - 符号表文件的UUID与dSYM文件的UUID是一致的，因此可以通过符号表工具生成的符号表文件来查看dSYM文件的UUID：
        - 步骤： 生成符号表文件(.zip) -> 解压符号表文件(.symbol) -> 使用文本编辑器打开符号表文件 -> 搜索UUID
    - 每次构建或者发布APP版本的时候，备份App对应的dSYM文件！
- 手动分析崩溃日志:
  - 下载生成的崩溃文件：xxx2018.crash
  - 创建文件夹：crashLog
  - 将 .crash，dSYM，app的二进制包 放到crashLog文件夹目录下
  - 复制symbolicatecrash文件到crashLog：　/应用程序/Xcode.app/Contents/SharedFrameworks/DVTFoundation.framework/Versions/A/Resources/symbolicatecrash
  - 生成log文件:

```
开始执行命令生成log文件:
cd /Users/niexiaobo/Downloads/Crach
./symbolicatecrash ./.crash ./.dSYM > crash20180308.log

如果报错了,执行下面命令:
export DEVELOPER_DIR=/Applications/Xcode.app/Contents/Developer

再执行:
./symbolicatecrash ./.crash ./.dSYM > crash20180308.log

成功的话文件夹里有个crash20180308.log
```

  - 查找分析:
    - 相关设备信息等等
    - 搜索关键词Last Exception Backtrace查看日志
- 常见崩溃:
  - 非主线程刷新UI
  - KVO
    - 添加删除不对称, 漏删或者重复删除
    - 添加监听的属性字符串错误, 没有对应的属性
  - KVC　Key不能为nil　Value设置错误的值类型
  - nil /null 值问题
    - 设置text值是Null崩溃
      - 字典value有null存储时　model=nil直接取属性
      - 数据集合类型，如字典、数组中插入元素
    - 后台返回null,iOS前端没有判断或处理
    - 富文本初始化值不能为nil
  - 数组越界　不可变数组使用可变数组方法
  - 调用不存在的方法 unrecognized selector　极光推送证书过期
  - NaN崩溃　使用0做被除数(NaN)　字典value有@(NaN)
  - APP权限设置崩溃　Info么有配置相册,相机,定位,保存,麦克风,健康 权限等等
  - 类型不一致　参数id使用时没有判断只类型
  - 野指针引起的崩溃 EXC_BAD_ACCESS
    - 访问内存被释放的对象　SIGILL
    - 僵尸对象检查NSZombie
  - 信号量错误
    - SIGSEGV: 当硬件出现错误、访问不可读的内存地址或向受保护的内存地址写入数据时，就会发生这个错误
    - SIGBUS: 总线错误信号（SIGBUG）代表无效内存访问，即访问的内存是一个无效的内存地址
    - SIGILL：非法指令时, 访问一段已经释放的内存或是一个数据段，有时EXC_BAD_INSTRUCTION
    - SIGABRT：当操作系统发现不安全的情况时
    - 看门狗超时(错误码固定是0x8badf00d)：它经常出现在执行一个同步网络调用而阻塞主线程的情况
  - 循环引用导致对象/内存不释放　block
  - tableView等复用对象没有注册
    - tableView的局部刷新问题,抖动问题　Cell删除崩溃
    - Xcode在项目更名之后，把项目转移到其他机器出现ld: file not found:
  - ld: library not found for 关联库　Pod源无法下载问题, 路径替换
  - 数据异步操作, 资源竞争导致bug　卡顿/堵塞: 主线程下载图片,视频等操作
  - 系统适配问题　Socket长连接，进入后台没有关闭