RunLoop

一个线程一次只能执行一个任务，执行完成后线程就会退出。Runloop提供让线程能随时处理事件，这种模式通常被称为Event Loop。关键处理以下几个问题：

- 如何管理事件/消息

- 如何让线程在空闲时休眠处理资源占用问题/有消息立即唤醒的问题

APM（OC与Swift机制一样）

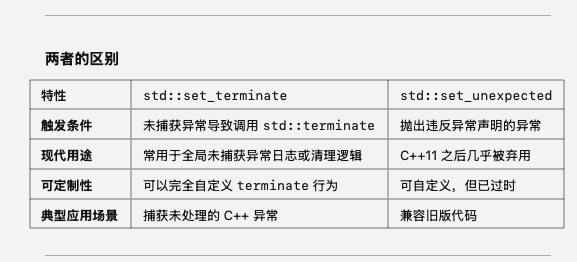
崩溃类型

- Mach异常

1. 内核回想当前task的某个专门处理异常的port发消息，该消息会被一次转为signal/NSException抛出，如果想在mach层捕获，则需要注册自己的port
   1. 不听循环通过mach\_msg()读取port传来的消息
   2. 读取成功后挂起所有线程（当前处理线程、白名单线程除外）
      1. 通过task\_threads()获取所有当前task的所有线程
      2. 通过thread\_suspend()挂起某个线程
      3. 后续还原其它线程恢复线程thread\_rsume()
   3. 清除所有的monitor(自定义）。恢复原来的port
      1. 避免自定义的监控机制干扰崩溃处理
      2. 清理已注册的异常处理器
      3. 防止重复的崩溃报告
      4. 减少内存泄漏或资源占用
      5. 程序恢复或恢复工作环境
   4. 抓取所有线程的信息保存到ksmachineContext结构体中
   5. 将各种信息交给crashContext（有可能就是一个独立的负责上报的线程）
   6. 恢复所有线程
   7. 通过mch\_msg()再发出一个消息告知没有处理这个异常
      1. 会员原本的mach处理端口restoreExceptionPorts()
      2. task\_set\_exception\_ports()
   8. 结束自己用用来处理异常的线程

- 信号Signal

1. 构造数组
   1. SIGABRT
   2. SIGBUS
   3. SIGFPE
   4. SIGILL
   5. SIGPIPE
   6. SIGSEGV
   7. SIGSYS
   8. SIGTRAP
2. 注册异常监听并保存原异常 sigaction(fatalSignals[i], &action, &g\_previousSignalHandlers[i]
3. 暂停线程
4. 保存context
5. 传递给外部函数
6. 恢复原来的环境
7. 信号抛出raise(sigNum)供其他人处理
8. 取消监听。使用sigaction将原本方法还原



- C++异常

1. 通过API std::set\_terminate（）/ std::set\_unexception设置自己的处理handler
2. 暂停线程
3. 拿崩溃type std::type\_info\*info = \_\_cxxabiv1::\_\_cxa\_current\_exception\_type()
4. 捕获crash后清空所有监听
5. 使用try catch 捕获当前异常信息
   1. 标准c++使用exc.what()获取描述
   2. 不同数据类型使用宏CATCH\_VALUE捕获异常并格式化为字符串
6. 生成崩溃上下文并收集信息
7. 处理崩溃
8. 恢复线程
9. 调用原始处理函数，供其他人处理

- NSException异常

1. 先通过NSGetUncaughtExceptionHandler()获取原异常处理函数
2. 然后通过NSSetUncaughtExceptionHandler()设置为自己的处理函数
3. 挂起其它线程
4. 获取调用堆栈地址 [exception callStackReturnAddress]不需要task获取堆栈，直接从NSException获取（apple包装好了）
5. 拿调用堆栈 uintptr\_t\* callstack = malloc(numFrames \* sizeof(\*callstack));
6. 转换为堆栈 callstack[i] = (uintptr\_t)[address[i] unsignedLongLongValue];
7. 构造要上报的崩溃事件
8. 环境还原，线程resume
9. 调用原来的处理函数，交还别人处理

- 主线程死锁监控

1. 通过子线程循环往复的判断一个标志位在主线程中是否被修改过，达到监听效果



- 用户主动抛出@throw

- 系统

- 应用状态

- 僵尸对象

如何统一处理信息上报？

1. 创建一个buffer将上报信息汇总
2. 将手机信息写入buffer
3. buffer满1024写入磁盘并刷新buffer
4. 写入image信息，如inageList
5. 写入线程信息
   1. 遍历所有调用线程
   2. 写入调用栈
   3. 写入寄存器
   4. 写入线程index
   5. 写入线程名称
   6. 写入dispatch\_queue名称
   7. 如果是崩溃线程
      1. 将stack pointer上的部分数据拷贝
      2. 将通过zombie记录的地址拷贝出来

符号化调用堆栈

1. 对每一层调用栈进行符号化
2. 把符号化的image/地址、symbol名，地址写入
   1. 根据adress获取符号名
      1. 判断address在第几个image ,imageIndexContainingAddress(adress)
      2. 获得image的header = \_dyld\_get\_image\_header(idx);
      3. 获取image基地址imageVMAddreSlid (uintptr\_t)\_dyld\_get\_image\_vmaddr\_slide(idx);
      4. 获取address相对于image的偏移 address - imageVMAddrSlid
      5. 获得segment在虚拟内存中的基地址 segmentBaseOfImageIndex(idx) + imageVMAddrSlide
      6. 获取image名称\_dyld\_get\_image\_name(idx)
      7. 获取header地址 (void \*) header
      8. 遍历ncmds，所有的加载
         1. 找到symbol tabl的load command (struct symtab\_command\*)cndPtr
         2. 通过load command 的offset + segment 基地址，得到symbol table实际地址 (STRUCT\_NLIST\*)(segementBase + symtabCmd -> symoff)
         3. 找到string table位置 segmentBase + symtabCmd->stroff
         4. 找到最佳符号，如果n\_value不是0
         5. 根据符号位置找到在string table的字符串

写入寄存器，从寄存器拿值

1. 列举所有可用寄存器
2. 根据寄存器序号在context取寄存器值context->machineContext.\_\_ss.\_\_rax

记录调用堆栈的部分信息

1. 拿到Stack pointer ，通过context->machineContext.\_\_ss\_\_rsp;
2. 拷贝站上信息范围为栈内20个，刚刚出栈的10个
3. 获取栈
   1. vm\_read\_overwrite(mach\_tsak\_self(), 源地址，比特数量，目标，存储地址）
4. 取出地址上的对象，将地址转化为对象

监控野指针

1. 判断当前address的object是否记录过对象

注册：NSSetUncaughtExceptionHandler捕获未处理的异常

- 先NSGetUncaughtExceptionHandler拿到异常处理函数，用完后再还回去给别人处理

- NSSetUncaughtExceptionHandler(&uncaughtExceptionHandler);

- void uncaughtExceptionHandler(NSException \*exception) {}

- exception

- [exception reason]

- exception callStackkSymbols

注册信号处理，处理各种异常信号mach

- 注册信号

- SIGABRT 中断信号

- SIGILL 非法指令信号

- SIGSEGV 无效内存访问信号

- SIGFPE 算数错误信号，比如除0

- SIGBUS 总线错误信号，访问已经释放内存

- SIGPIPE 管道错误信号

- 信号处理流程

- #include <signal.h>

- #include <execinfo.h>

- int frames = backtrace(callStack, 120);

- char \*\*symbols = backtrace\_symbols(callstack, frames;

- symbols[i]

- 上报

- signal(signal, SIG\_DFL) 程序崩溃后终止

- raise(signal) 继续程序行为

- OOM

- 无法直接捕获，通过标记和阈值来判断

- 检测applicationDidReceiveMemoryWarning

- 手动监控内存

- task\_info 和 mach\_task\_self()

- struct tsak\_basic\_info info;

- mach\_msg\_type\_number\_t size = TASK\_BASIC\_INFO\_COUNT

- kern\_return\_t kerr = task\_info(mach\_task\_self(), TASK\_BASIC\_INFO, (task\_info\_t)&info, &size);

- return (kerr == KERN\_SUCESS) ? info.resident\_size : 0; 反馈使用内存大小

- 启动时检测是否异常退出

OC try-catch

- 可以捕获OC的NSException异常，通常有@throw抛出。包括cocoa库或自己throw抛出的

- 不可以捕获

- c/c++异常，这个需要使用c++的try-catch实现

- posix信号，需要通过mach注册信号

- OOM，需要检测

- Signal信号异常，通过mach注册相关信号

- 有性能开销，谨慎使用

捕获mach异常时，为什么暂停非当前线程？

1. 防止其它线程修改内存或状态
   1. 确保可以完整捕获和记录崩溃信息
   2. 防止其它线程对程序状态做修改，
2. 确保崩溃线程和堆栈信息准确
   1. 程序崩溃时，为了准确获取堆栈，需要处于静止状态，其他线程可能会干扰当前线程的栈信息
3. 防止崩溃导致的并发问题
   1. 暂停其它线程，防止因数据竞争或静态条件导致的问题恶化
   2. 防止崩溃处理中发生新的竞争条件
4. 白名单线程：日志写入、异常处理、资源清理等操作，需要保留
5. 结束捕获后恢复所有线程

调试模式下，调试器会附加到进程，大多数框架不会上报此崩溃报告

使用sysctl函数实现运行时检测判断

- var info = kinfo\_proc()

- var mib = [CTL\_KERN, KERN\_PROC, KERN\_PROC\_PID, getpid()]

- var size = MemoryLayout<kinfo\_proc>.size

- let result = sysctl(&mib, u\_int(mib.count), &info, &size, nil, 0)

- return (info.kp\_proc.p\_flag & P\_TRACE) != 0

全埋点技术方案

- 应用启动事件

- 应用退出事件

- 页面浏览事件

- 控件点击事件

- Target-Action UIApplication的sendAction:to:from:for-Event

- 交换-didMoveToSuperview方法，添加Target-Action，并在action里触发事件

- 应用崩溃事件

- 采集手势

- 标记用户

- 设计上报结构

- 设计上报数据库与消化