线程监控——WatchDog 死锁

1. 线程监控组件（SNKThreadMonitor）

1.1 ThreadMonitor

```swift
public final class ThreadMonitor {
    // 单例
    public static let shared = ThreadMonitor()

    // 监控频率（不建议配置太高的频率，使用过高的频率会导致过高的CPU和资源占用）
    public var frequency: TimeInterval = 3

    // 活跃线程
    @Protected
    internal var _activeThreadInfo: [MachInfoProvider] = [MachInfoProvider]()
    public var activeThreadInfo: [MachInfoProvider] {
        $_activeThreadInfo.wrappedValue
    }

    // 开始监控
    public func startMonitoring() {
        startThreadMonitorring()
        startMonitorringTimer()
        registerThreadStateNotify()
    }

    // 注册代理
    public func registerDelegate(_ delegate: ThreadMonitorDelegate) {
        delegates.add(delegate)
    }

    // 停止监控
    public func stopMonitoring() {
        stopMonitorringTimer()
        stopThreadMonitorring()
        unregisterThreadStateNotify()
    }

    // 监控队列
    internal lazy var monitorQueue: DispatchQueue = {
        let queue = DispatchQueue(label: "com.snake.thread-monitor",
```

```
                                              qos: .default,
                                              attributes: .concurrent,
                                              autoreleaseFrequency: .workItem)
        return queue
    }()

    // 定时器，用于定期更新线程信息
    internal lazy var timer: DispatchSourceTimer = {
        DispatchSource.makeTimerSource(queue: monitorQueue)
    }()

    internal var delegates: NSHashTable<AnyObject> = NSHashTable.weakObjects()

    private init() {}
}
```

> 目前会通过三种方式监测当前进程的线程信息：
>
> 1. ThreadMonitor内部维护一个并发队列 `monitorQueue` ，通过 `timer` 使用 `frequency` 频率，定时监测当前进程中的全部线程。（Mach/Darwin框架）
> 2. 向进程注册线程状态变化的hook函数。（POSIX框架）
> 3. 向应用程序注册线程状态变化的回调通知。（Foundation）

1.2 线程信息

1.2.1 ThreadInfoProviding

```
public protocol ThreadInfoProviding {
    associatedtype T
    // 线程句柄/端口
    var thread: T { get set }
    // 是否活跃
    var isActive: Bool { get }
    // 描述
    var description: String { get }
}

public struct MachInfoProvider: ThreadInfoProviding, Hashable {
    public typealias T = MachThread
    // 由操作系统内核维护管理的线程端口号
    public var thread: MachThread
    // 线程基础信息
```

```
    public var basicInfo: MachBasicInfo { thread.basicInfo }
    // 线程身份信息
    public var identifierInfo: MachIdentifierInfo { thread.identifierInfo }
    // 线程扩展信息
    public var extendInfo: MachExtendedInfo { thread.extendInfo }
    // 系统内核标记的线程状态
    public var machState: ThreadMachState? {
        return ThreadMachState(rawValue: Int(basicInfo.run_state))
    }
    public var machStateDesc: String {
        guard let machState = machState else { return "Unknown(\(basicInfo.run_state))" }
        return machState.desc
    }
    // 系统对线程的状态标记
    public var flag: ThreadFlagsType? { ThreadFlagsType(rawValue: Int(basicInfo.flags)) }
    public var flagDesc: String {
        guard let flag = flag else { return "Unknown(\(basicInfo.flags))" }
        return flag.desc
    }
    // CPU占用情况
    public var cpuUsage: String {
        return "\(Float(basicInfo.cpu_usage)*100/Float(TH_USAGE_SCALE))%"
    }
    // 线程名称
    public var name: String { extendInfo.name }

    // 调用堆栈信息
    public var backTraceDesc: String {
        return SNKBackTrace(thread).symbolsDescription
    }

    public var isActive: Bool { machState == .running || machState == .uninterruptible }

    public var description: String {
        let basicInfo = basicInfo
        let identifierInfo = identifierInfo
        let extendInfo = extendInfo
        let string = "\nMach-State:\(machStateDesc)\nQueue Address:\(identifierInfo.dispatch_qaddr)\nThread ID:\(identifierInfo.thread_id)\nThread Mach-Port:\(thread)\nFlag:\(flagDesc)\nSuspend Count:\(basicInfo.suspend_count)\nSleep Time:\(basicInfo.sleep_time)\nName:\(extendInfo.name)\nQueue Name:\(identifierInfo.queueName)\nCPU Usage:\(Float(basicInfo.cpu_usage)*100/Float(TH_USAGE_SCALE))%\n\(backTraceDesc)"
        return string
```

```swift
    }
    public init(_ value: MachThread) {
        self.thread = value
    }
}

public struct POSIXInfoProvider: ThreadInfoProviding {
    public typealias T = POSIXThread
    // POSIX框架下的线程句柄
    public var thread: POSIXThread
    // 线程内省状态
    public let introspectionState: ThreadIntrospectionState?

    public var isActive: Bool { introspectionState == .create || introspectionState == .start }

    public var description: String {
        let string = "\nState:\(introspectionState?.desc ?? "Unknown")\nPOSIX Address:\(thread)"
        return string
    }
    public init(_ value: POSIXThread, state: ThreadIntrospectionState?) {
        self.thread = value
        self.introspectionState = state
    }
}
```

> 内部根据监控方式（框架）的不同，提供了两种线程信息结构体:
>
> 1. `MachInfoProvider`：Mach框架下的线程信息
> 2. `POSIXInfoProvider`：POSIX框架下的线程信息
>
> 可通过结构体获取线程对应的详细信息，如:
>
> 1. `backTraceDesc`:线程当前的调用栈
> 2. `cpuUsage`:CPU占用
> 3. 系统态/用户态下的执行耗时
> 4. `name`:线程名称
> 5. `queueName`:线程所属的队列名称
> 6. 线程当前的运行状态/标记

1.2.2 状态

```swift
/**
 PTHREAD_INTROSPECTION_THREAD
 POSIX线程内省状态
 */
@frozen
public enum ThreadIntrospectionState: Int {
    case create = 1
    case start
    case terminate
    case destroy
    var desc: String {
        switch self {
        case .destroy:
            return "Destroy"
        case .terminate:
            return "Terminate"
        case .start:
            return "Start"
        case .create:
            return "Create"
        }
    }
}
/**
 TH_STATE
 系统/CPU内核状态
 */
@frozen
public enum ThreadMachState: Int {
    case running = 1
    case stopped
    case wating
    case uninterruptible
    case halted
    var desc: String {
        switch self {
        case .running:
            return "Running"
        case .stopped:
            return "Stopped"
        case .wating:
            return "Waiting"
        case .uninterruptible:
            return "Uninterruptible"
        case .halted:
            return "Halted"
```

```
        }
    }
}
/**
 TH_FLAGS
 系统/CPU开放的标记位
 */
@frozen
public enum ThreadFlagsType: Int {
    case swapped = 0x1
    case idle = 0x2
    case forcedIdle = 0x4
    var desc: String {
        switch self {
        case .swapped:
            return "Swapped out"
        case .idle:
            return "Idle thread"
        case .forcedIdle:
            return "Global forced idle"
        }
    }
}
```

1.3 回调

1.3.1 注册回调

```
// 注册回调
ThreadMonitor.registerDelegate(_:)
```

1.3.2 ThreadMonitorDelegate

```
// ThreadMonitorDelegate
public protocol ThreadMonitorDelegate: AnyObject {
    // Foundation.NSThread定义的通知代理
    var threadNotifyDelegate: ThreadMonitorNotifyProviding? { get }
    // Mach/Darwin框架下的定时回调的代理
    var threadInfosDelegate: ThreadMonitorInfosProviding? { get }
    // POSIX框架下的线程内省状态变更的代理
    var threadStateDelegate: ThreadMonitorIntrospectionStateProviding? { get }
}
```

> 三种代理均为 `optional` ，按需实现

1.3.2.1 ThreadMonitorNotifyProviding

```swift
// NSThread定义的全局通知回调
public protocol ThreadMonitorNotifyProviding: AnyObject {
    func threadMonitorDidReceiveWillExit(thread: Thread?, info: (any ThreadInfoProviding)?)
    func threadMonitorDidReceiveWillBecomeMulti()
    func threadMonitorDidReceiveDidBecomeSingle()
}
```

1.3.2.2 ThreadMonitorInfosProviding

```swift
// 定时刷新的全局线程信息回调
// infos: ThreadMonitor.recordedThreadInfo
public protocol ThreadMonitorInfosProviding: AnyObject {
    func threadMonitorDidReceiveInfosUpdated(_ infos: [MachInfoProvider])
    func threadMonitorDidReceiveInfosDeadLockDetached(_ infos: [MachInfoProvider], deadLockInfos: [MachInfoProvider: [MachInfoProvider]])
}
```

1.3.2.3 ThreadMonitorIntrospectionStateProviding

```swift
// 线程内省状态回调
public protocol ThreadMonitorIntrospectionStateProviding: AnyObject {
    func threadMonitorDidReceiveStateChanged(_ info: POSIXInfoProvider)
    func threadMonitorDidReceiveThreadCreated(_ info: POSIXInfoProvider)
    func threadMonitorDidReceiveThreadStarted(_ info: POSIXInfoProvider)
    func threadMonitorDidReceiveThreadFinished(_ info: POSIXInfoProvider)
    func threadMonitorDidReceiveThreadDestroied(_ info: POSIXInfoProvider)
}
```

1.4 APM指标

```swift
// 指标类型
public protocol IndicatorType {
    // 名称
```

```
    var name: String { get }
    // 标题
    var title: String { get }
    // 详细描述
    var description: String { get }
    // 调用栈
    var callStacks: [SNKBackTrace] { get }
}

// 锁类型
public enum DeadLockType {
    // 互斥锁
    case mutex(_ holding: SNKBackTrace, _ waitings: [SNKBackTrace])
}

// 指标
public enum Indicator: IndicatorType {
    case deadLock(_ type: DeadLockType)
    case priorityInversion
    case longWaiting
    case longRunning
    public var name: String {
        switch self {
        case .deadLock:
            return "DEAD_LOCK_DETACHED"
        case .priorityInversion:
            return "PROORITY_INVERSION_DETACHED"
        case .longWaiting:
            return "LONG_WAITING"
        case .longRunning:
            return "LONG_RUNNING"
        }
    }
    public var title: String {
        switch self {
        case let .deadLock(type):
            switch type {
            case .mutex(let h, _):
                guard let t = SNKBackTrace(h.thread).symbols.firstObject as? String else { return "Null" }
                return t
            }
        case .priorityInversion:
            return "优先级反转"
        case .longWaiting:
            return "长时间等待"
        case .longRunning:
```

```
                return "执行耗时过久"
            }
        }
        public var description: String {
            switch self {
            case let .deadLock(type):
                switch type {
                case let .mutex(h, ws):
                    let hp = MachInfoProvider(h.thread)
                    let wsp = ws.map{ MachInfoProvider($0.thread) }.reduce("") { $0 + $1.description + "\n" }
                    return "HoldingThreadInfo:\n\(hp.description)\nWaitingThreadInfos:\n\(wsp)"
                }
            case .priorityInversion:
                return ""
            case .longWaiting:
                return ""
            case .longRunning:
                return ""
            }
        }
        public var callStacks: [SNKBackTrace] {
            switch self {
            case let .deadLock(type):
                switch type {
                case let .mutex(holding, waitings):
                    var result = [holding]
                    result += waitings
                    return result
                }
            case .priorityInversion:
                return []
            case .longWaiting:
                return []
            case .longRunning:
                return []
            }
        }
}
```

2. WatchDog 死锁监控

2.1 卡死归因

> 卡死通常发生在冷启动阶段，在某线程（主要是主线程）出现长时间等待时，系统会强制回收当前进程的内存，导致应用程序闪退。即：用户可能等待了10 秒什么都没有做，这个 App 就崩溃了。

> 如果不对卡死崩溃做一层过滤的话，会加大OOM崩溃的误判几率，提高定位问题的难度。

- 卡顿监控：认为主线程响应时间超过3~5秒之后就是一次卡死

> 误判几率高，假如5s内主线程分别执行了3个任务，第3个任务完成时触发了卡顿阈值，但实际上的卡顿操作是在第2个任务，此时dump到的堆栈信息是不准确的

- 主线程死锁、长时间等待以及主线程I/O

> 定时监控当前进程下的全部线程信息，分不同的场景，对不同的锁类型进行校验与转存。再抛给上层通过其它方式进行上报等操作。

2.2 死锁检测

通过Mach/Darwin框架可拿到实时的全线程信息，通过分析线程的状态、标识以及CPU占用率产生了两种分析思路

```
void mach_check_thread_dead_lock(thread_t thread, NSMutableDictionary<NSNumber *,NSMutableArray<NSNumber *> *> *threadWaitDic) {
#ifndef __i386__
    thread_extended_info_data_t threadInfoData;
    mach_msg_type_number_t threadInfoCount = THREAD_EXTENDED_INFO_COUNT;
    thread_identifier_info_data_t threadIDData;
    mach_msg_type_number_t threadIDDataCount = THREAD_IDENTIFIER_INFO_COUNT;
    if (thread_info(thread, THREAD_EXTENDED_INFO, (thread_info_t)&threadInfoData, &threadInfoCount) == KERN_SUCCESS &&
        thread_info(thread, THREAD_IDENTIFIER_INFO, (thread_info_t)&threadIDData, &threadIDDataCount) == KERN_SUCCESS) {
        uint64_t thread_id = threadIDData.thread_id;
        integer_t cpu_usage = threadInfoData.pth_cpu_usage;
        integer_t run_state = threadInfoData.pth_run_state;
        integer_t flags = threadInfoData.pth_flags;
        // 情景1:
        // 线程处于等待状态且已被换出，CPU占用率为0
        if ((run_state & TH_STATE_WAITING) && (flags & TH_FLAGS_SWAPPED) && cpu_usage == 0) {
            checkMainEmptyCPUUsageWithWapped(thread, thread_id, threadWaitDic);
        }
        // 情景2:
        // 主线程的 CPU 占用一直很高 ，处于运行的状态，那么就应该怀疑主线程是否存在一些死循环等 CPU 密集型的任务。
        if ((run_state & TH_STATE_RUNNING) && cpu_usage > 800) {
            //怀疑死循环
            NSLog(@"怀疑死循环:%llu",thread_id);
        }
```

```
    }
#endif
}
```

2.2.1 死锁/锁等待

CPU占用为0 -> Waiting状态 -> Swapped Out：疑似死锁

```
// 主线程CPU占用为0，等待状态且已被换出
void checkMainEmptyCPUUsageWithWapped(thread_t thread, uint64_t thread_id, NSMutableDictionary<NSNumber *,NSMutableArray<NSNumber *> *> *threadWaitDic) {
#ifndef __i386__
    // 通过符号化判断它是否是一个锁等待的方法。
    _STRUCT_MCONTEXT machineContext;
    //通过 thread_get_state 获取完整的 machineContext 信息，包含 thread 状态信息
    mach_msg_type_number_t state_count = j_threadStateCountByCPU();
    kern_return_t kr = thread_get_state(thread, j_threadStateByCPU(), (thread_state_t)&machineContext.__ss, &state_count);
    if (kr != KERN_SUCCESS) {
        NSLog(@"Fail get thread: %u", thread);
        return;
    }
    //通过指令指针来获取当前指令地址
    SNKBackTrace *backTrace = [SNKBackTrace backTraceWith:thread];
    NSMutableArray *symbols = backTrace.symbols;
    for (int i = 0; i < symbols.count; i++) {
        const char *cString = [symbols[i] UTF8String];
        // https://github.com/apple-oss-distributions/libpthread/blob/d8c4e3c212553d3e0f5d76bb7d45a8acd61302dc/src/imports_internal.h#L47
        if (strcmp(cString, "__psynch_mutexwait") == 0) {
            // 认为`thread`正在等待锁
            uintptr_t firstParam = j_firstParamRegister(&machineContext);
            struct pthread_mutex_s *mutex = (struct pthread_mutex_s *)firstParam;
            uint32_t *tid = mutex->psynch.m_tid;
            uint64_t hold_lock_thread_id = *tid;
            //需要判断死锁
            NSMutableArray *array = threadWaitDic[@(hold_lock_thread_id)];
            if (!array) {
                array = [NSMutableArray array];
            }
            [array addObject:@(thread_id)];
            threadWaitDic[@(hold_lock_thread_id)] = array;
            break;
```

```
        }

    }
    // TODO: 其他锁情况
    //
    //__psynch_rw_rdlock   ReadWrite lock
    //__psynch_rw_wrlock   ReadWrite lock
    //__ulock_wait         UnfariLock lock
    //_kevent_id           GCD lock

    // psynch_cvwait, semwait_signal, psynch_mutexwait, psynch_mutex_trylock, dispatch_sync_f_slow
#endif
}
```

1. 将线程头部栈帧的函数名与 `锁等待` 函数进行匹配，满足条件时，认为该线程正处在锁等待状态。将其保存在 `threadWaitDic` 中，方便后续校验。
2. 任何 `锁等待` 函数中必有一个参数表示持有的 `锁` 的结构体信息，此处以POSIX互斥锁为例：`pthread_mutex_t`
3. 从苹果官方开源文档中查询得到 `锁` 的结构体信息（`pthread_mutex_s`），和 `pthread_mutex_t` 一致
4. 根据 `锁等待` 函数中表示 `锁` 的参数位置，依据不同架构下的C函数传递规范，从对应架构的通用寄存器中拿到 `锁` 的内存信息（`pthread_mutex_s`），并直接强转
5. `pthread_mutex_s->psynch.m_tid` 即表示持有该锁的线程ID
6. 从对应架构的程序计数器 `__ss.__pc` 中拿到当前正在执行的指令的地址；从对应架构的帧指针寄存器 `__ss.__fp` 中拿到指向当前函数的堆栈帧，堆栈帧包含了函数的局部变量、参数和其他与函数调用相关的信息；从对应架构的链接寄存器 `__ss.__lr` 中拿到函数调用的返回地址；遍历帧指针寄存器，生成完整的调用堆栈信息。

2.3 测试

2.3.1 NSLock互斥锁

```
    _lockA = [[NSLock alloc]init];
    _lockA.name = @"I am LockA";

    _lockB = [[NSLock alloc]init];
    _lockB.name = @"I am LockB";

```

```
    _lockC = [[NSLock alloc]init];
    _lockC.name = @"I am LockC";

    _holdLockAThread = [[NSThread alloc]initWithTarget:self selector:@selector(holdLockA) object:nil];
    [_holdLockAThread setName:@"I hold LockA!"];
    [_holdLockAThread start];

    _holdLockBThread = [[NSThread alloc]initWithTarget:self selector:@selector(holdLockB) object:nil];
    [_holdLockBThread setName:@"I hold LockB!"];
    [_holdLockBThread start];

    _holdLockCThread = [[NSThread alloc]initWithTarget:self selector:@selector(holdLockC) object:nil];
    [_holdLockCThread setName:@"I hold LockC!"];
    [_holdLockCThread start];

    - (void)holdLockA {
    [_lockA lock];

    NSLog(@"AThread hold lockA success");
    sleep(2);

    NSLog(@"AThread want lockB");
    [_lockB lock];
    NSLog(@"AThread hold lockB success");
}

- (void)holdLockB {
    [_lockB lock];

    NSLog(@"BThread hold lockB success");
    sleep(2);

    NSLog(@"BThread want lockC");
    [_lockC lock];
    NSLog(@"BThread hold lockC success");
}

- (void)holdLockC {
    [_lockC lock];

    NSLog(@"CThread hold lockC success");
    sleep(2);

    NSLog(@"CThread want lockA");
```

```
    [_lockA lock];
    NSLog(@"CThread hold lockA success");
}
```

2.3.2 Console Log

```
threadMonitorDidReceiveInfosDeadLockDetached:💩💩💩💩

Holding Info:

Mach-State:Waiting
Queue Address:6137835904
Thread ID:66494673
Thread Mach-Port:31235
Flag:Swapped out
Suspend Count:0
Sleep Time:0
Name:I hold LockB!
Queue Name:Null
CPU Usage:0.0%
libsystem_kernel.dylib          0x1b0544c04 __psynch_mutexwait  +  8
libsystem_pthread.dylib         0x1b059b7c0 _pthread_mutex_firstfit_lock_wait  +  80
libsystem_pthread.dylib         0x1b059926c _pthread_mutex_firstfit_lock_slow  +  244
SNKThreadMonitor_Example        0x100001bc8   +  100
Foundation                      0x180bf4750 __NSThread__start__  +  704
libsystem_pthread.dylib         0x1b059e3b4 _pthread_start  +  116

Waiting Infos:


Mach-State:Waiting
Queue Address:6137262464
Thread ID:66494672
Thread Mach-Port:31491
Flag:Swapped out
Suspend Count:0
Sleep Time:0
Name:I hold LockA!
Queue Name:Null
CPU Usage:0.0%
libsystem_kernel.dylib          0x1b0544c04 __psynch_mutexwait  +  8
libsystem_pthread.dylib         0x1b059b7c0 _pthread_mutex_firstfit_lock_wait  +  80
```

```
libsystem_pthread.dylib        0x1b059926c _pthread_mutex_firstfit_lock_slow + 244
SNKThreadMonitor_Example       0x100001b4c   +  100
Foundation                     0x180bf4750 __NSThread__start__  +  704
libsystem_pthread.dylib        0x1b059e3b4 _pthread_start  +  116


Holding Info:

Mach-State:Waiting
Queue Address:6138409344
Thread ID:66494674
Thread Mach-Port:29955
Flag:Swapped out
Suspend Count:0
Sleep Time:0
Name:I hold LockC!
Queue Name:Null
CPU Usage:0.0%
libsystem_kernel.dylib         0x1b0544c04 __psynch_mutexwait  +  8
libsystem_pthread.dylib        0x1b059b7c0 _pthread_mutex_firstfit_lock_wait + 80
libsystem_pthread.dylib        0x1b059926c _pthread_mutex_firstfit_lock_slow + 244
SNKThreadMonitor_Example       0x100001c44   +  100
Foundation                     0x180bf4750 __NSThread__start__  +  704
libsystem_pthread.dylib        0x1b059e3b4 _pthread_start  +  116

Waiting Infos:


Mach-State:Waiting
Queue Address:6137835904
Thread ID:66494673
Thread Mach-Port:31235
Flag:Swapped out
Suspend Count:0
Sleep Time:0
Name:I hold LockB!
Queue Name:Null
CPU Usage:0.0%
libsystem_kernel.dylib         0x1b0544c04 __psynch_mutexwait  +  8
libsystem_pthread.dylib        0x1b059b7c0 _pthread_mutex_firstfit_lock_wait + 80
libsystem_pthread.dylib        0x1b059926c _pthread_mutex_firstfit_lock_slow + 244
SNKThreadMonitor_Example       0x100001bc8   +  100
```

```
Foundation                      0x180bf4750 __NSThread__start__  +  704
libsystem_pthread.dylib         0x1b059e3b4 _pthread_start  +  116


Holding Info:

Mach-State:Waiting
Queue Address:6137262464
Thread ID:66494672
Thread Mach-Port:31491
Flag:Swapped out
Suspend Count:0
Sleep Time:0
Name:I hold LockA!
Queue Name:Null
CPU Usage:0.0%
libsystem_kernel.dylib          0x1b0544c04 __psynch_mutexwait  +  8
libsystem_pthread.dylib         0x1b059b7c0 _pthread_mutex_firstfit_lock_wait  +  80
libsystem_pthread.dylib         0x1b059926c _pthread_mutex_firstfit_lock_slow  +  244
SNKThreadMonitor_Example        0x100001b4c   +  100
Foundation                      0x180bf4750 __NSThread__start__  +  704
libsystem_pthread.dylib         0x1b059e3b4 _pthread_start  +  116

Waiting Infos:


Mach-State:Waiting
Queue Address:6138409344
Thread ID:66494674
Thread Mach-Port:29955
Flag:Swapped out
Suspend Count:0
Sleep Time:0
Name:I hold LockC!
Queue Name:Null
CPU Usage:0.0%
libsystem_kernel.dylib          0x1b0544c04 __psynch_mutexwait  +  8
libsystem_pthread.dylib         0x1b059b7c0 _pthread_mutex_firstfit_lock_wait  +  80
libsystem_pthread.dylib         0x1b059926c _pthread_mutex_firstfit_lock_slow  +  244
SNKThreadMonitor_Example        0x100001c44   +  100
Foundation                      0x180bf4750 __NSThread__start__  +  704
libsystem_pthread.dylib         0x1b059e3b4 _pthread_start  +  116
```

成功拦截到了处于锁等待的线程，并获取对应的线程信息。

仓库地址

> 完成以上操作，就初步实现了对应用程序的死锁监控并上报异常死锁日志的功能。
>
> 目前需要补充一下情景2中的检测内容，并对锁类型进行扩充。