DDLog 源码解析

一，使用流程

主要类是 DDLog 这个单例

1. 添加 logger

```
- (void)addLogger:(id <DDLogger>)logger withLevel:(DDLogLevel)level {
    if (!logger) {
        return;
    }
    dispatch_async(_loggingQueue, ^{ @autoreleasepool {
        [self lt_addLogger:logger level:level];
    } });
}
```

添加了logger，DDLog 才产生对应的日志，我们用得多的主要有两类 logger ,如下：

- DDTTYLogger： xcode 终端输出的 logger,根据需要可以设置自定义的 formatter，可以添加一些额外的信息，如时间，方法名，文件名等等

```
@protocol DDLogFormatter <NSObject>
@required
/**
 设置输出的字符串，可以在原 log 基础上添加一些特殊前缀或后缀
 **/
- (NSString * __nullable)formatLogMessage:(DDLogMessage *)logMessage
NS_SWIFT_NAME(format(message:));
@end
```

```
- (NSString *)formatLogMessage:(DDLogMessage *)logMessage {
    NSString *logLevel;
    switch (logMessage->_flag) {
        case DDLogFlagError    : logLevel = @"❌ERROR"; break;
        case DDLogFlagWarning  : logLevel = @"⚠️WARNING"; break;
        case DDLogFlagInfo     : logLevel = @"✅INFO"; break;
```

```
        case DDLogFlagDebug    : logLevel = @"🔨DEBUG"; break;
        default                : logLevel = @"📖VERBOSE"; break;
    }
    NSString *logMsg = logMessage->_message;
    return [NSString stringWithFormat:@"%@ %@ %@ | %@", logMessage.timestamp,
logLevel,logMessage.function, logMsg];
}
```

- DDFileLogger： 磁盘文件写入的 logger

```
/**
 *  Designated initializer, requires a `DDLogFileManager` instance
 */
- (instancetype)initWithLogFileManager:(id <DDLogFileManager>)logFileManager
NS_DESIGNATED_INITIALIZER;
```

需要传一个支持 DDLogFileManager 协议的对象，指定文件路径，文件个数，文件占用空间大小等信息

```
@protocol DDLogFileManager <NSObject>
@required
// Public properties
//最多存多少个文件
@property (readwrite, assign, atomic) NSUInteger maximumNumberOfLogFiles;
/**
 最大磁盘容量
 **/
@property (readwrite, assign, atomic) unsigned long long logFilesDiskQuota;
// Public methods
/**
 *  log文件存贮路径
 */
@property (nonatomic, readonly, copy) NSString *logsDirectory;
...
```

DDLogFileManagerDefault 是自带默认配置的遵循 DDLogFileManager 协议的类

```objc
//指定log文件的磁盘路径
- (instancetype)initWithLogsDirectory:(NSString *)aLogsDirectory {
    if ((self = [super init])) {
            //最大存5个文件
        _maximumNumberOfLogFiles = kDDDefaultLogMaxNumLogFiles;
        //最大占用20M
        _logFilesDiskQuota = kDDDefaultLogFilesDiskQuota;
        if (aLogsDirectory) {
            _logsDirectory = [aLogsDirectory copy];
        } else {
            _logsDirectory = [[self defaultLogsDirectory] copy];
        }
        NSKeyValueObservingOptions kvoOptions = NSKeyValueObservingOptionOld | NSKeyValueObservingOptionNew;
                    // 添加对最大文件个数和最大容量的监听
        [self addObserver:self forKeyPath:NSStringFromSelector(@selector(maximumNumberOfLogFiles)) options:kvoOptions context:nil];
        [self addObserver:self forKeyPath:NSStringFromSelector(@selector(logFilesDiskQuota)) options:kvoOptions context:nil];
        NSLogVerbose(@"DDFileLogManagerDefault: logsDirectory:\n%@", [self logsDirectory]);
        NSLogVerbose(@"DDFileLogManagerDefault: sortedLogFileNames:\n%@", [self sortedLogFileNames]);
    }
    return self;
}
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context {
    NSNumber *old = change[NSKeyValueChangeOldKey];
    NSNumber *new = change[NSKeyValueChangeNewKey];
    if ([old isEqual:new]) {
        // No change in value - don't bother with any processing.
        return;
    }
    if ([keyPath isEqualToString:NSStringFromSelector(@selector(maximumNumberOfLogFiles))] ||
        [keyPath isEqualToString:NSStringFromSelector(@selector(logFilesDiskQuota))]) {
        NSLogInfo(@"DDFileLogManagerDefault: Responding to configuration change: %@", keyPath);
                            //发现有修改，立马执行删除逻辑
        dispatch_async([DDLog loggingQueue], ^{ @autoreleasepool {
                                                    [self deleteOldLogFiles];
```

```
38                                                  } });
39      }
40  }
```

2. log 级别控制

- DDLogLevelOff：　　//关闭 log
- DDLogLevelError　　//支持 DDLogError() 的运行
- DDLogLevelWarning　//支持 DDLogWarning() 和 DDLogLevelError 级别的 log 的运行
- DDLogLevelInfo　　//支持 DDLogInfo() 和 DDLogLevelWarning 级别的 log 的运行
- DDLogLevelDebug　　//支持 DDLogDebug() 和 DDLogLevelInfo 级别的 log 的运行
- DDLogLevelVerbose　　//支持 DDLogVerbose() 和 DDLogLevelDebug 级别的 log 的运行
- DDLogLevelAll　　//支持以上5种 log 输出

3. DDLogMessage 的创建

```
1  @property (readonly, nonatomic) NSString *message;
2  @property (readonly, nonatomic) DDLogLevel level;
3  @property (readonly, nonatomic) DDLogFlag flag;
4  @property (readonly, nonatomic) NSInteger context;
5  @property (readonly, nonatomic) NSString *file;
6  @property (readonly, nonatomic) NSString *fileName;
7  @property (readonly, nonatomic) NSString * __nullable function;
8  @property (readonly, nonatomic) NSUInteger line;
9  @property (readonly, nonatomic) id __nullable tag;
10 @property (readonly, nonatomic) DDLogMessageOptions options;
11 @property (readonly, nonatomic) NSDate *timestamp;
12 @property (readonly, nonatomic) NSString *threadID; // ID as it appears in
   NSLog calculated from the machThreadID
13 @property (readonly, nonatomic) NSString *threadName;
14 @property (readonly, nonatomic) NSString *queueLabel;
```

二，技术点

1. 同步队列同步执行任务

```
1  for (DDLoggerNode *loggerNode in self._loggers) {
```

```
    // skip the loggers that shouldn't write this message based on the log level
    if (!(logMessage->_flag & loggerNode->_level)) {
        continue;
    }
    // 在当前线程下，一个一个执行，不管 loggerNode->_loggerQueue是同步还是异步队列
    dispatch_sync(loggerNode->_loggerQueue, ^{ @autoreleasepool {
        [loggerNode->_logger logMessage:logMessage];
    } });
}
```

能在当前线程中执行任务，一个一个按顺序执行，也保证了线程安全

```
- (unsigned long long)maximumFileSize {
    __block unsigned long long result;
    dispatch_block_t block = ^{
        result = self->_maximumFileSize;
    };
    NSAssert(![self isOnGlobalLoggingQueue], @"Core architecture requirement failure");
    NSAssert(![self isOnInternalLoggerQueue], @"MUST access ivar directly, NOT via self.* syntax.");
    dispatch_queue_t globalLoggingQueue = [DDLog loggingQueue];
    dispatch_sync(globalLoggingQueue, ^{
        dispatch_sync(self.loggerQueue, block);
    });
    return result;
}
```

2. 当前线程创建子线程执行任务，用 Group 控制当前线程的完成

```
for (DDLoggerNode *loggerNode in self._loggers) {
    // skip the loggers that shouldn't write this message based on the log level
    if (!(logMessage->_flag & loggerNode->_level)) {
        continue;
    }
    // 在当前线程下，创建子线程，由 loggerNode->_loggerQueue 决定同步还是异步执行
    dispatch_group_async(_loggingGroup, loggerNode->_loggerQueue, ^{ @autoreleasepool {
        [loggerNode->_logger logMessage:logMessage];
```

```
    } });
}
// 在当前线程下的所有
dispatch_group_wait(_loggingGroup, DISPATCH_TIME_FOREVER);
```

能在当前线程中创建子线程去执行任务，同时阻塞当前线程直到当前线程所有任务完成

3. NSFileHandle 实现写入文件

```
- (NSFileHandle *)currentLogFileHandle {
    if (_currentLogFileHandle == nil) {
        NSString *logFilePath = [[self currentLogFileInfo] filePath];
        _currentLogFileHandle = [NSFileHandle
fileHandleForWritingAtPath:logFilePath];
        [_currentLogFileHandle seekToEndOfFile];
        if (_currentLogFileHandle) {
            [self scheduleTimerToRollLogFileDueToAge];
            // Here we are monitoring the log file. In case if it would be
deleted ormoved
            // somewhere we want to roll it and use a new one.
            _currentLogFileVnode = dispatch_source_create(
                    DISPATCH_SOURCE_TYPE_VNODE,
                    [_currentLogFileHandle fileDescriptor],
                    DISPATCH_VNODE_DELETE | DISPATCH_VNODE_RENAME,
                    self.loggerQueue
                    );
            dispatch_source_set_event_handler(_currentLogFileVnode, ^{
@autoreleasepool {

NSLogInfo(@"DDFileLogger: Current logfile was moved. Rolling it and creating a
new one");
                                                                    [self
 rollLogFileNow];
                                                                } });
            #if !OS_OBJECT_USE_OBJC
            dispatch_source_t vnode = _currentLogFileVnode;
            dispatch_source_set_cancel_handler(_currentLogFileVnode, ^{
                dispatch_release(vnode);
            });
            #endif
            dispatch_resume(_currentLogFileVnode);
        }
    }
    return _currentLogFileHandle;
}
```

- `dispatch_source_create`

可以设置一个 DISPATCH_SOURCE_TYPE_VNODE 类型的dispatch source，你可以从这个dispatch source中接收文件删除、写入、重命名等通知，`dispatch_source_set_event_handler` 设置收到通知的处理

4. NSLog 底层是 writev 实现的

NSLog