CustomLoggers

How to write your own custom loggers.

Introduction

Loggers allow you to direct log messages wherever you want. For general information about loggers, see the architecture page. The best part is that it's relatively easy to write your own custom loggers.

The DDLog header file defines the DDLogger protocol. It consists of only 3 mandatory methods:

```
@protocol DDLogger <NSObject>
@required

- (void)logMessage:(DDLogMessage *)logMessage;

/**
 * Formatters may optionally be added to any logger.
 * If no formatter is set, the logger simply logs the message as it is given in logMessage.
 * Or it may use its own built in formatting style.
**/
- (id <DDLogFormatter>)logFormatter;
- (void)setLogFormatter:(id <DDLogFormatter>)formatter;

@optional

/**
 * Since logging is asynchronous, adding and removing loggers is also asynchronous.
 * In other words, the loggers are added and removed at appropriate times with regards to log messages.
 *
 * - Loggers will not receive log messages that were executed prior to when they were added.
 * - Loggers will not receive log messages that were executed after they were removed.
 *
 * These methods are executed in the logging thread.
 * This is the same thread that will execute every logMessage:withLevel: invocation.
 * Loggers may use these methods for proper thread synchronization or any other setup.
**/

- (void)didAddLogger;
- (void)willRemoveLogger;

@end
```

Furthermore, there is a base logger implementation one can extend that will automatically implement the logFormatter getter and setter. So implementing a logger can be pretty straight-forward.

Skeleton Implementation

Let's assume we want to write a custom logger. It doesn't take much to write the skeleton code:

MyCustomLogger.h:

```
#import <Foundation/Foundation.h>
#import "DDLog.h"

@interface MyCustomLogger : DDAbstractLogger <DDLogger>
{
}
@end
```

MyCustomLogger.m

```
#import "MyCustomLogger.h"

@implementation MyCustomLogger

- (void)logMessage:(DDLogMessage *)logMessage
{
    NSString *logMsg = logMessage->logMsg;

    if (logFormatter)
        logMsg = [logFormatter formatLogMessage:logMessage];
    if (logMsg)
    {
        // Write logMsg to wherever...
    }
}

@end
```

Pretty simple huh?

Details

The logFormatter is designed to be an optional component for loggers. This is for simplicity. And the separation between loggers and formatters is for reusability. A single formatter can be applied to multiple loggers.

However, you are obviously free to do whatever you want. If it doesn't make sense to support formatters for your custom logger, you don't have to. (This may be the case with database loggers.) And if your custom logger is to use a single pre-defined format, then you can simply do the formatting directly within the logger itself, and forego the formatter. It is completely up to you.

The DDLogMessage object encapsulates the information about a log message. It is also defined in DDLog.h:

```
@interface DDLogMessage : NSObject
{
  @public
        int logLevel;
        NSString *logMsg;
        NSDate *timestamp;
        // ...
}

// ...

@end
```

Threading

Most of the multi-threading issues are solved for you. The following 3 methods are **always** invoked on the loggingThread:

```
- (void)logMessage:(DDLogMessage *)logMessage;

- (void)didAddLogger;
- (void)willRemoveLogger;
```

Using these 3 methods you can setup resources, perform logging, and teardown resources without worrying about multi-threaded complications.

Furthermore, the DDAbstractLogger implements thread-safe getters and setters logFormatter. And it does so in such a way that allows you to access the logFormatter variable directly from within your logMessage method! (This is for performance reasons.)

However, if your custom logger has custom configuration variables, you may need to make them atomic and/or thread-safe.