FineGrainedLogging

When you need something more advanced than simple log levels.

Introduction

Most of us are familiar with log levels. They are simple to understand, and they're used in almost every logging framework. However, there are times when something more advanced is desired. Something more fine grained.

Example #1:

You have a large source code file that you are debugging. The code is logically separated into 7 different sub-components. One of these sub-components is having issues. So you'd like to enable some extra logging, but you don't want to see all the log statements from the other 6 sub-components. That would be too much junk to filter through. Wouldn't it be nice if your logging framework was tailored to fit the logical separations in your code?

Example #2:

You're adding a new feature to your application. The implementation of this feature spans across several different files. Something isn't quite working right and you'd like to enable some extra logging, but you only want to see log statements related to this new feature. The problem is, there are a bunch of other log statements in these files. You don't want to remove them. You just want a little bit extra fine-grained control.

Not a problem. Lumberjack can help!

Details

If you look closely at the definitions in the DDLog header file, you'll notice something interesting:

```
#define LOG_FLAG_ERROR    (1 << 0)  // 0...0001
#define LOG_FLAG_WARN     (1 << 1)  // 0...0010
#define LOG_FLAG_INFO     (1 << 2)  // 0...0100
#define LOG_FLAG_VERBOSE  (1 << 3)  // 0...1000

#define LOG_ERROR   (ddLogLevel & LOG_FLAG_ERROR) // Bitwise AND
```

Lumberjack is actually using a bitmask! And it is only using 4 bits. This means that every other bit is **reserved for you**.

The Lumberjack framework actually comes with a sample Xcode project that demonstrates fine-grained logging. The demo project is based upon example #2 above. Here is how it works:

Imagine you have a series of related timers. The calculation of these timers is complicated, and are thus spread across multiple files. However, they are related. In our example we have two different categories of timers. One for "food" and another for "sleep".

So we want all of our logging related to our food timers to be grouped together. This way it can be enabled/disabled independently of the rest of the logging within the various files. And we want the same solution applied to our sleep timers as well.

So we create our own logging header file to be used by our application. We'll call it "MYLog.h":

```
#import "DDLog.h"

// The first 4 bits are being used by the standard levels (0 - 3)
// All other bits are fair game for us to use.

#define LOG_FLAG_FOOD_TIMER    (1 << 4)  // 0...0010000
#define LOG_FLAG_SLEEP_TIMER   (1 << 5)  // 0...0100000

#define LOG_FOOD_TIMER  (ddLogLevel & LOG_FLAG_FOOD_TIMER)
#define LOG_SLEEP_TIMER (ddLogLevel & LOG_FLAG_SLEEP_TIMER)

#define DDLogFoodTimer(frmt, ...)   ASYNC_LOG_OBJC_MAYBE(ddLogLevel, LOG_FLAG_FOOD_TIMER,  frmt, ##__VA_ARGS__)
#define DDLogSleepTimer(frmt, ...)  ASYNC_LOG_OBJC_MAYBE(ddLogLevel, LOG_FLAG_SLEEP_TIMER, frmt, ##__VA_ARGS__)

// Now we decide which flags we want to enable in our application

#define LOG_FLAG_TIMERS (LOG_FLAG_FOOD_TIMER | LOG_FLAG_SLEEP_TIMER))
```

Now we have two new macros that can be used in our application.

- DDLogFoodTimer(...)
- DDLogSleepTimer(...)

We also defined LOG_FLAG_TIMERS which can be used to enable both loggers. LOG_FLAG_FOOD_TIMER | LOG_FLAG_SLEEP_TIMER = 0010000 | 0100000 = 0110000

Then in our source code files we just make sure that our ddLogLevel bitmask has the flags set to enable logging for our timers:

```
#import "MYLog.h"

static const int ddLogLevel = LOG_LEVEL_WARN | LOG_FLAG_TIMERS;

...

DDLogFoodTimer(@"TimerTwo: Hungry - Need Food");
```

How many fine-grained options are available?

As many as you want.

The ddLogLevel variable is declared as an int. This means that on most systems it will be 32 bits. Since the normal log level stuff is taking up 4 bits, this gives you 28 bits to work with.

But if that isn't enough, you could change it to be a 64-bit value, or you could simply add another variable. The macros have been designed to make it easy for you to plug in your own custom stuff.

For example, let's say you wanted to have 30 fine-grained log options. No problem. Just do something like this.

```
#define LOG_FLAG_CAT   (1 << 0)  // 0...00001
#define LOG_FLAG_DOG   (1 << 1)  // 0...00010

#define LOG_CAT  (ddLogLevel2 & LOG_FLAG_CAT)
#define LOG_DOG  (ddLogLevel2 & LOG_FLAG_DOG)

#define DDLogCat(frmt, ...)  ASYNC_LOG_OBJC_MAYBE(ddLogLevel2, LOG_FLAG_CAT, frmt, ##__VA_ARGS__)
#define DDLogDog(frmt, ...)  ASYNC_LOG_OBJC_MAYBE(ddLogLevel2, LOG_FLAG_DOG, frmt, ##__VA_ARGS__)
```

All we had to do was use a new variable - ddLogLevel2.

You should also note that you aren't restricted to the 4 pre-defined log levels. You can define them however you want. Learn more in the Custom Log Levels page.