glog源码解析三：glog功能扩展：如何按天切割、自定义日志输出等

glog

Jun 04, 2017

上篇笔记:glog源码解析二：从LOG(INFO)到写入日志的代码分析介绍了glog的部分源码，这篇笔记尝试从实用角度看下如何修改和扩展glog的功能。我在实际工作中想尝试解决以下几个问题：

1. glog按照大小 FLAGS_max_log_size 分隔，公司内的log是按天分隔的，对于追查问题更方便些，如何修改为按天分割日志?
2. glog是否有退场功能?
3. glog默认高级别的日志会打印到低级别日志文件里，如何修改为每种级别打印到各自的日志里?
4. 如何增加自定义的输出日志的方式?

1. 如何按天切割日志?

上篇笔记里介绍到了 LogFileObject ，该类负责日志文件的创建、关闭以及写入。

LogDestination 调用 LogFileObject::write 接口，看下 write 里的这几行代码：

```
//超过大小或者进程id改变
if (static_cast<int>(file_length_ >> 20) >= MaxLogSize() ||
    PidHasChanged()) {
  if (file_ != NULL) fclose(file_);
  file_ = NULL;
  file_length_ = bytes_since_flush_ = 0;
  rollover_attempt_ = kRolloverAttemptFrequency-1;
}
```

我们可以增加一个 DayHasChanged 的函数，判断是否属于同一天，如果不是则同样close文件，记录哪天同样使用static变量即可。

2. 日志如何退场?

有两个思路，仍然是修改源码的方式，按天切割日志为周的方式，每周7个文件，周而复始的替换。这里需要修改文件名不能带时间戳。

3. 如何修改高级别日志不输出到低级别日志文件?

上一篇笔记提到了标准输出日志的代码

```
inline void LogDestination::LogToAllLogfiles(LogSeverity severity,
                                             time_t timestamp,
                                             const char* message,
                                             size_t len) {

  if ( FLAGS_logtostderr ) {           // global flag: never log to file
    ColoredWriteToStderr(severity, message, len);
  } else {
    for (int i = severity; i >= 0; --i)
      LogDestination::MaybeLogToLogfile(i, timestamp, message, len);
  }
}
```

这里去掉for循环，直接

```
    LogDestination::MaybeLogToLogfile(severity, timestamp, message, len);
```

4. 如何增加自定义的输出日志的方式?

上面三种修改虽然方便，但是修改了日志库的源码，需要重新编译，不具有通用性，而且修改源码的做法维护比较复杂。

比如我们接手了一个模块，使用了glog但是并没有考虑上面几个问题，导致磁盘占用非常严重。如何将模块已经使用的glog，与公司内通用的 `comlog` 结合呢，我想到的最佳方案是利用glog优秀的可扩展性，在glog里实现自定义的 `comlog` 输出方式，这样改动成本最小，同时本质上使用了统一的日志库。

修改方式有两种：增加自定义的LogSink 或者 替换标准输出的base::Logger。

4.1 如何增加自定义的LogSink

上篇笔记介绍过，在分发日志时，会分发到标准的默认日志输出，以及用户自定义的LogSink输出。这一节我们具体看下。

首先实现一个继承自 `google::LogSink` 的子类：

```
class MyNewLogSink : public google::LogSink {
    virtual void send(google::LogSeverity severity, const char* full_filename,
            const char* base_filename, int line,
            const struct ::tm* tm_time,
            const char* message, size_t message_len) {
        std::cout << "MyNewLogSink::send |" << google::LogSink::ToString(
                severity,
                full_filename,
                line,
                tm_time,
                message,
                message_len) << std::endl;;
    }

    virtual void WaitTillSent() {
        std::cout << "MyNewLogSink::WaitTillSent" << std::endl;
    }
};
```

`MyNewLogSink` 里我们重新实现了 `send` 方法， `WaitTillSent` 会在每次 `send` 后调用，用于一些异步写的场景。 `ToString` 是glog实现的辅助static函数。

接下来就是add到sink里：

```
int main(int argc, char* argv[]) {
    FLAGS_logtostderr = 1;
    google::InitGoogleLogging(argv[0]);
    MyNewLogSink my_new_log_sink;
    google::AddLogSink(&my_new_log_sink);

    LOG(INFO) << "log info" << std::endl;

    return 0;
}
```

运行后可以看到glog默认的写日志方法和我们自定义的 `send` 方法都调用了：

```
$ ./new_log_sink
I0607 23:09:47.168449 14894 new_log_sink.cpp:45] log info
MyNewLogSink::send |I0607 23:09:47.000000 14894 src/new_log_sink.cpp:45] log info
MyNewLogSink::WaitTillSent
```

如果觉得不想调用glog默认方法，就使用另外一个宏 `LOG_TO_SINK_BUT_NOT_TO_LOGFILE` ：

```
int main(int argc, char* argv[]) {
    FLAGS_logtostderr = 1;
    google::InitGoogleLogging(argv[0]);
    MyNewLogSink my_new_log_sink;

    LOG_TO_SINK(&my_new_log_sink, INFO) << "LOG_TO_SINK";
    LOG_TO_SINK_BUT_NOT_TO_LOGFILE(&my_new_log_sink, INFO) << "LOG_TO_SINK_BUT_NOT_TO_LOGFILE1";

    return 0;
}
```

4.2 如何替换标准输出的base::Logger

基于 `base::Logger` 我们可以实现自定义的输出Logger，替换掉目前默认使用的 `LogFileObject`

```
class MyInfoLogger : public google::base::Logger {
public:
    virtual void Write(bool force_flush,
            time_t timestamp,
            const char* message,
            int message_len) {
        std::cout << "MyInfoLogger::Write "
            << "|" << force_flush
            << "|" << timestamp
            << "|" << message
            << "|" << message_len << std::endl;
    }

    virtual void Flush() {
        std::cout << "MyInfoLogger::Flush" << std::endl;
    }

    virtual google::uint32 LogSize() {
        return 0;
    }
};//MyInfoLogger
```

使用 `SetLogger` 替换

```cpp
int main(int argc, char* argv[]) {
    google::InitGoogleLogging(argv[0]);

    MyInfoLogger my_info_logger;
    google::base::SetLogger(google::GLOG_INFO, &my_info_logger);

    LOG(INFO) << "LOG(INFO)";
    LOG(WARNING) << "LOG(WARNING)";

    return 0;
}
```

输出：

```
$ ./new_inf_logger
MyInfoLogger::Write |0|1496848853|I0607 23:20:53.523600 12673 new_inf_logger.cpp:34] LOG(INFO)
|61
MyInfoLogger::Write |0|1496848853|W0607 23:20:53.523738 12673 new_inf_logger.cpp:35] LOG(WARNING)
|64
```

这样我们就可以使用 `LOG(xxx)` 的宏自定义输出了，注意不要设置 `FLAGS_logtostderr` ，否则不会生效，想了解的同学可以直接翻一下源码或者上篇笔记。

日志的分析和应用到这篇笔记分析完成，我们知道glog在 `FATAL` 级别的日志输出时会自动输出当前的调用栈，那么如何优雅的在cpp里获取当前的调用栈呢？下篇笔记我们继续看下。