Glog使用文档

Yeolar 2014-12-20 18:07

来自Google的Glog是一个应用程序的日志库。它提供基于C++风格的流的日志API，以及各种辅助的宏。打印日志只需以流的形式传给 `LOG(level)`，例如：

```
#include <glog/logging.h>

int main(int argc, char* argv[]) {
  // Initialize Google's logging library.
  google::InitGoogleLogging(argv[0]);

  // ...
  LOG(INFO) << "Found " << num_cookies << " cookies";
}
```

Glog定义了一系列的宏来简化记录日志的工作。你可以：按级别打印日志，通过命令行控制日志行为，按条件打印日志，不满足条件时终止程序，引入自定义的日志级别，等等。

目录

日志级别

可以指定下面这些级别（按严重性递增排序）： `INFO`, `WARNING`, `ERROR`, and `FATAL` 。打印 `FATAL` 消息会在打印完成后终止程序。和其他日志库类似，级别更高的日志会在同级别和所有低级别的日志文件中打印。

`DFATAL` 级别会在调试模式（没有定义 `NDEBUG` 宏）中打印 `FATAL` 日志，但是会自动降级为 `ERROR` 级别，而不终止程序。

如果不指定的话，Glog输出到文件 /tmp/<program name>.<hostname>.<user name>.log.<severity level>.<date>-<time>.<pid> （比如 /tmp/hello_world.example.com.hamaji.log.INFO.20080709-222411.10474）。默认情况下，Glog对于 ERROR 和 FATAL 级别的日志会同时输出到stderr。

设置flag

一些flag会影响Glog的输出行为。如果安装了GFlags库，编译时会默认使用它，这样就可以在命令行传递flag（别忘了调用 ParseCommandLineFlags 初始化）。比如你想打开 --logtostderr flag，可以这么用：

```
./your_application --logtostderr=1
```

如果没有安装GFlags，那可以通过环境变量来设置，在flag名前面加上前缀 GLOG_ 。比如：

```
GLOG_logtostderr=1 ./your_application
```

常用的flag有：

logtostderr （bool，默认为 false）
日志输出到stderr，不输出到日志文件。

colorlogtostderr （bool，默认为 false）
输出彩色日志到stderr。

stderrthreshold （int，默认为2，即 ERROR）
将大于等于该级别的日志同时输出到stderr。日志级别 INFO，WARNING，ERROR，FATAL 的值分别为0、1、2、3。

minloglevel （int，默认为0，即 INFO）
打印大于等于该级别的日志。日志级别的值同上。

log_dir （string，默认为 ""）
指定输出日志文件的目录。

v （int，默认为0）
显示所有 VLOG(m) 的日志， m 小于等于该flag的值。会被 --vmodule 覆盖。

vmodule （string，默认为 ""）
每个模块的详细日志的级别。参数为逗号分隔的一组 <module name>=<log level>。 <module name> 支持通配（即gfs*代表所有gfs开头的名字），匹配不包含扩展名的文件名（忽略 .cc/.h./-inl.h 等）。 <log level> 会覆盖 --v 指定的值。

logging.cc 中还定义了其他一些flag。grep一下 DEFINE_ 可以看到全部。

也可以通过修改 FLAGS_* 全局变量来改变flag的值。

```
LOG(INFO) << "file";
// Most flags work immediately after updating values.
FLAGS_logtostderr = 1;
LOG(INFO) << "stderr";
FLAGS_logtostderr = 0;
// This won't change the log destination. If you want to set this
// value, you should do this before google::InitGoogleLogging .
FLAGS_log_dir = "/some/log/directory";
LOG(INFO) << "the same file";
```

按条件/次数打印日志

有时你可能只想在满足一定条件的时候打印日志。可以使用下面的宏来按条件打印日志：

```
LOG_IF(INFO, num_cookies > 10) << "Got lots of cookies";
```

上面的日志只有在满足 `num_cookies > 10` 时才会打印。

另一种情况，如果代码被执行多次，可能只想对其中某几次打印日志。

```
LOG_EVERY_N(INFO, 10) << "Got the " << google::COUNTER << "th cookie";
```

上面的代码会在执行的第1、11、21、...次时打印日志。 `google::COUNTER` 用来表示是哪一次执行。

可以将这两种日志用下面的宏合并起来。

```
LOG_IF_EVERY_N(INFO, (size > 1024), 10) << "Got the " << google::COUNTER
                                        << "th big cookie";
```

不只是每隔几次打印日志，也可以限制在前n次打印日志：

```
LOG_FIRST_N(INFO, 20) << "Got the " << google::COUNTER << "th cookie";
```

上面会在执行的前20次打印日志。

调试模式

调试模式的日志宏只在调试模式下有效，在非调试模式会被清除。可以避免生产环境的程序由于大量的日志而变慢。

```
DLOG(INFO) << "Found cookies";

DLOG_IF(INFO, num_cookies > 10) << "Got lots of cookies";

DLOG_EVERY_N(INFO, 10) << "Got the " << google::COUNTER << "th cookie";
```

CHECK宏

常做状态检查以尽早发现错误是一个很好的编程习惯。 CHECK 宏和标准库中的 assert 宏类似，可以在给定的条件不满足时终止程序。

CHECK 和 assert 不同的是，它不由 NDEBUG 控制，所以一直有效。因此下面的 fp->Write(x) 会一直执行：

```
CHECK(fp->Write(x) == 4) << "Write failed!";
```

有各种用于相等/不等检查的宏： CHECK_EQ，CHECK_NE，CHECK_LE，CHECK_LT，CHECK_GE，CHECK_GT 。它们比较两个值，在不满足期望时打印包括这两个值的 FATAL 日志。注意这里的值需要定义了 operator<<(ostream, ...) 。

比如：

```
CHECK_NE(1, 2) << ": The world must be ending!";
```

每个参数都可以保证只用一次，所以任何可以做为函数参数的都可以传给它。参数也可以是临时的表达式，比如：

```
CHECK_EQ(string("abc")[1], 'b');
```

如果一个参数是指针，另一个是 NULL ，编译器会报错。可以给 NULL 加上对应类型的 static_cast 来绕过。

```
CHECK_EQ(some_ptr, static_cast<SomeType*>(NULL));
```

更好的办法是用 CHECK_NOTNULL 宏：

```
CHECK_NOTNULL(some_ptr);
some_ptr->DoSomething();
```

该宏会返回传入的指针，因此在构造函数的初始化列表中非常有用。

```
struct S {
  S(Something* ptr) : ptr_(CHECK_NOTNULL(ptr)) {}
  Something* ptr_;
};
```

因为该特性，这个宏不能用作C++流。如果需要额外信息，请使用 CHECK_EQ 。

如果是需要比较C字符串（ char* ），可以用 CHECK_STREQ，CHECK_STRNE，CHECK_STRCASEEQ，CHECK_STRCASENE 。 CASE 的版本是不区分大小写的。这里可以传入 NULL 。 NULL 和任何非 NULL 的字符串是不等的，两个 NULL 是相等的。

这里的参数都可以是临时字符串，比如 CHECK_STREQ(Foo().c_str(), Bar().c_str()) 。

CHECK_DOUBLE_EQ 宏可以用来检查两个浮点值是否等价，允许一点误差。 CHECK_NEAR 还可以传入第三个浮点参数，指定误差。

细节日志

当你在追比较复杂的bug的时候，详细的日志信息非常有用。但同时，在通常开发中需要忽略太详细的信息。对这种细节日志的需求，Glog提供了 VLOG 宏，使你可以自定义一些日志级别。通过 --v 可以控制输出的细节日志：

```
VLOG(1) << "I'm printed when you run the program with --v=1 or higher";
VLOG(2) << "I'm printed when you run the program with --v=2 or higher";
```

和日志级别相反，级别越低的 VLOG 越会打印。比如 --v=1 的话， VLOG(1) 会打印， VLOG(2) 则不会打印。对 VLOG 宏和 --v flag可以指定任何整数，但通常使用较小的正整数。 VLOG 的日志级别是 INFO 。

细节日志可以控制按模块输出：

```
--vmodule=mapreduce=2,file=1,gfs*=3 --v=0
```

会：

a. 为 mapreduce.{h,cc} 打印 VLOG(2) 和更低级别的日志
b. 为 file.{h,cc} 打印 VLOG(1) 和更低级别的日志
c. 为前缀为gfs的文件打印 VLOG(3) 和更低级别的日志
d. 其他的打印 VLOG(0) 和更低级别的日志

其中 (c) 给出的通配功能支持 * （0或多个字符）和 ? （单字符）通配符。

细节级别的条件判断宏 VLOG_IS_ON(n) 当 --v 大于等于n时返回true。比如：

```
if (VLOG_IS_ON(2)) {
  // do some logging preparation and logging
  // that can't be accomplished with just VLOG(2) << ...;
}
```

此外还有 VLOG_IF，VLOG_EVERY_N，VLOG_IF_EVERY_N ，和 LOG_IF，LOG_EVERY_N，LOF_IF_EVERY 类似，但是它们传入的是一个数字的细节级别。

```
VLOG_IF(1, (size > 1024))
   << "I'm printed when size is more than 1024 and when you run the "
      "program with --v=1 or more";
VLOG_EVERY_N(1, 10)
   << "I'm printed every 10th occurrence, and when you run the program "
      "with --v=1 or more. Present occurence is " << google::COUNTER;
VLOG_IF_EVERY_N(1, (size > 1024), 10)
   << "I'm printed on every 10th occurence of case when size is more "
      " than 1024, when you run the program with --v=1 or more. ";
      "Present occurence is " << google::COUNTER;
```

失败信号处理

Glog库还提供了一个信号处理器，能够在 `SIGSEGV` 之类的信号导致的程序崩溃时导出有用的信息。使用 `google::InstallFailureSignalHandler()` 加载信号处理器。下面是它输出的一个例子。

```
*** Aborted at 1225095260 (unix time) try "date -d @1225095260" if you are using GNU date ***
*** SIGSEGV (@0x0) received by PID 17711 (TID 0x7f893090a6f0) from PID 0; stack trace: ***
PC: @           0x412eb1 TestWaitingLogSink::send()
    @     0x7f892fb417d0 (unknown)
    @           0x412eb1 TestWaitingLogSink::send()
    @     0x7f89304f7f06 google::LogMessage::SendToLog()
    @     0x7f89304f35af google::LogMessage::Flush()
    @     0x7f89304f3739 google::LogMessage::~LogMessage()
    @           0x408cf4 TestLogSinkWaitTillSent()
    @           0x4115de main
    @     0x7f892f7ef1c4 (unknown)
    @           0x4046f9 (unknown)
```

注意： `InstallFailureSignalHandler()` 在x86_64系统架构上可能会引发退栈的死锁，导致递归地调用 `malloc` 。这是内置的退栈的bug，建议在安装Glog之前安装libunwind。更多解释可以看Glog的 INSTALL 文件。

```
# apt-get install libunwind libunwind-dev
```

默认情况，信号处理器把失败信息导出到stderr。可以用 `InstallFailureWriter()` 定制输出位置。

其他

支持CMake

Glog并不自带CMake支持，如果想在CMake脚本中使用它，可以把 FindGlog.cmake (/media/note/2014/12/20/glog/FindGlog.cmake) 添加到CMake的模块目录下。然后像下面这样使用：

```
find_package (Glog REQUIRED)
include_directories (${GLOG_INCLUDE_DIR})

add_executable (foo main.cc)
target_link_libraries (foo glog)
```

性能

Glog提供的条件日志宏（比如 `CHECK`，`LOG_IF`，`VLOG`，`...`）在条件判断失败时，不会执行右边表达式。因此像下面这样的检查不会牺牲程序的性能。

```
CHECK(obj.ok) << obj.CreatePrettyFormattedStringButVerySlow();
```

自定义失败处理函数

`FATAL` 级别的日志和 `CHECK` 条件失败时会终止程序。可以用 `InstallFailureFunction` 改变该行为。

```
void YourFailureFunction() {
  // Reports something...
  exit(1);
}

int main(int argc, char* argv[]) {
  google::InstallFailureFunction(&YourFailureFunction);
}
```

默认地，Glog会导出stacktrace，程序以状态1退出。stacktrace只在Glog支持栈跟踪的系统架构（x86和x86_64）上导出。

原始日志

`<glog/raw_logging.h>` 可用于要求线程安全的日志，它不分配任何内存，也不加锁。因此，该头文件中定义的宏可用于底层的内存分配和同步的代码。

谷歌风格的perror()

`PLOG()`，`PLOG_IF()`，`PCHECK()` 和对应的 `LOG*` 和 `CHECK` 类似，但它们会同时在输出中加上当前 `errno` 的描述。如：

```
PCHECK(write(1, NULL, 2) >= 0) << "Write NULL failed";
```

下面是它的输出：

```
F0825 185142 test.cc:22] Check failed: write(1, NULL, 2) >= 0 Write NULL failed: Bad address [14]
```

Syslog

SYSLOG，SYSLOG_IF，SYSLOG_EVERY_N 宏会在正常日志输出的同时输出到syslog。注意输出日志到syslog会大幅影响性能，特别是如果syslog配置为远程日志输出。所以在用它们之前一定要确定影响，一般来说很少使用。

跳过日志

打印日志的代码中的字符串会增加可执行文件的大小，而且也会带来泄密的风险。可以通过使用 GOOGLE_STRIP_LOG 宏来删除所有低于特定级别的日志：

比如使用下面的代码：

```
#define GOOGLE_STRIP_LOG 1    // this must go before the #include!
#include <glog/logging.h>
```

编译器会删除所有级别低于该值的日志。因为 VLOG 的日志级别是 INFO （等于0），设置 GOOGLE_STRIP_LOG 大于等于1会删除所有 VLOG 和 INFO 日志。

Windows用户的注意事项

Glog定义的 ERROR 日志级别，和 windows.h 中的定义有冲突。可以在引入 glog/logging.h 之前定义 GLOG_NO_ABBREVIATED_SEVERITIES ，这样Glog就不会定义 INFO，WARNING，ERROR，FATAL 。不过你仍然可以使用原来的宏：

```
#define GLOG_NO_ABBREVIATED_SEVERITIES
#include <windows.h>
#include <glog/logging.h>

// ...

LOG(ERROR) << "This should work";
LOG_IF(ERROR, x > y) << "This should be also OK";
```

但是你不能再在 glog/logging.h 中的函数中使用 INFO，WARNING，ERROR，FATAL 了。

```
#define GLOG_NO_ABBREVIATED_SEVERITIES
#include <windows.h>
#include <glog/logging.h>

// ...

// This won't work.
// google::FlushLogFiles(google::ERROR);

// Use this instead.
google::FlushLogFiles(google::GLOG_ERROR);
```

如果不需要使用 windows.h 中定义的 `ERROR`， 那么也可以尝试下面的方法：

- 在引入 windows.h 之前 `#define WIN32_LEAN_AND_MEAN` 或 `NOGDI` 。
- 在引入 windows.h 之后 `#undef ERROR` 。

http://www.yeolar.com/note/2014/12/20/glog/