Вычислительные технологии **Том 12, № 6, 2007**

EFFICIENT LOG FILTER

A. A. Fedotov
Intel Corporation, Moscow, Russia
e-mail: `alexei.fedotov@gmail.com`

Сервис журналирования используется в программных продуктах для описания и сохранения внутреннего состояния программы и происходящих в ней процессов для последующего анализа. Использование подобного сервиса широко распространено при разработке, тестировании, поддержке и обслуживании, администрировании и ускорении сложных программных продуктов.

Выдача журнала может сильно замедлить реальную программу в связи с обилием происходящих в ней событий. Поэтому очень важной задачей при организации журналирования является фильтрация происходящих событий во время выполнения программы. Данная статья предлагает эффективный способ фильтрации журнала, не зависящий от сложности правил фильтрации в том случае, если эти правила редко изменяются.

А именно, в любой программе найдется ограниченное количество вызовов системы журналирования и эти места вызовов могут быть перечислены в списке. В случае изменения фильтра алгоритм просматривает этот список и кэширует для каждого места вызова системы, активен ли этот вызов.

Introduction

A log is an output stream where an application sequentially reports the application-related events. Developers often use logging to make cross-platform debugging easier. Part of the logging infrastructure remains in the released version of the application, which helps resolving the possible cause of errors.

Logging is important for testers or advanced users who don't have access to the source code. In this case a clear log message can help to locate a problem, to decipher its cause, to find a workaround, or even a bug resolution [1]. Analyzing logs can help to improve future product releases [2]. Logging can be used by customer support and those who are responsible for application code maintenance.

When monitoring distributed systems such as a grid computational cluster, network latencies and failures prove a serious problem [3, 4]. One possible solution is to use a local log cache and deliver the newest log chunks to the main server when network capacity permits. Isolating log delivery in a separate OS process or a thread increases overall system reliability.

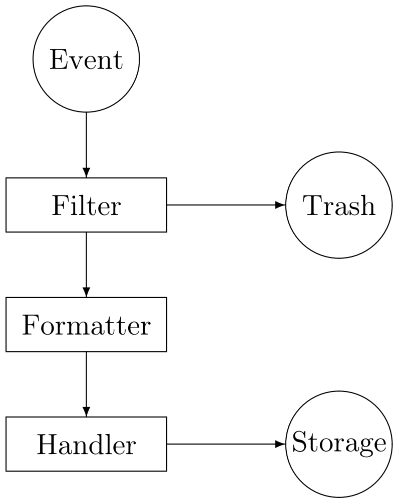

Fig. 1. Logging Design

Logging helps to insure reliability, security, and accountability of file systems [5, 6] and databases [7]. After a failure the system can recover state based on historical information contained in the log. These very important applications are out of the scope of the article.

Recently generic logging API [8] and internal virtual machine logging management [9] became part of Java[1] specification. The Apache Commons Logging project has a pre-built support for five logging implementations [10]. That an accepted programming language would integrate a logging mechanism would seem to indicate that logging has become an important aspect of modern applications.

There is a new programming technique that allows programmers to modularize crosscutting concerns such as logging. This technique is called Aspect-oriented programming (AOP) [11], and it is shown that this technique significantly improves code locality and reusability of software [12]. Application of AOP ideas to logging could be simply reformulated as follows [13]. A developer describes the logging functionality using some descriptive language, for example, he wants to check a value of some variable each time an execution thread enters the function `foo()`. The implementation does the rest of the job by parsing instructions and placing appropriate code to the function `foo()` call sites.

Languages with dynamic class loading such as Java greatly benefit from AOP. Efficient AOP-based implementation patches classes in runtime by adding logging functionality where it is requested. AspectJ, which applies AOP principles to Java, have become the most popular AOP implementation [14].

For statically compiled languages such as C or C++ AOP-oriented approach ends with processing initial source tree and adding the calls to logging subsystem. So an implementation of portable, simple and efficient log filtering in runtime would be a valuable part of AOP logging implementation for these languages.

Different logging designs have a common structure [15, 16], which is shown in Fig. 1. The structure is logically divided into the following components: a filter, a formatter, and an output handler. The parts cooperate in the following way: first, the filter rejects excessive logging events. Second, the formatter presents output in the form of a text, XML or raw data. Finally, the handler flushes the data to a storage device such as a disk, database or console.

[1]Other brands and names are the property of their respective owners.

If an application is sufficiently big, it is quite important to filter a debugging output. The compile-time filtering is done by developers — they use different logging calls to specify explicitly which part of logging information should be exempted from the final application release. This is well-known approach, but it is not yet sufficient when logging is a required feature of the product release.

One needs an efficient filtering in runtime, so users could specify which logging information they would like to receive. The filtering is usually integrated into the logging system, so an application avoids formatting of logging messages if they are not needed. The simplest way to filter logging messages is based on a level of importance prescribed to each message. When application decides if it should print the logging message it just compares the level of the message with a level given by a user. The main drawback of this approach is that high logging levels are unusable due to the big amount of waste they produce.

That is why it is convenient to use domains for classification of logging messages in addition to levels. The logging domain is a node of a domain tree, that is the tree which describes the logical structure of the application. Each logging message is associated with some logging domain. The filter is a set of marked branches of the tree. The logging message is passed to a formatter if its domain belongs to a marked branch.

The common solution is to match a logging domain of the issued message against a given filter each time when a logging event happens. For example, one could maintain a tree-like structure in memory, and for each debugging message check the flag in the correspondent node. A straightforward implementation of this good idea, when extensively developed, quickly becomes difficult to maintain. For example, if we store the node flag in a global variable, we need to initialize it, to update it when a filter changes and to mention it each time when the logging event is triggered in this domain. This does not include an effort to link these variables into a tree-like structure.

It is quite convenient when a logging domain is represented as a dot or slash separated string. These strings form a domain tree in the same way that path names form a directory structure of a file system. We would think about domain check as matching a domain string against a set of filter strings which could contain wildcards. The logging message is passed to a formatter if its domain matches one of the filter strings.

This approach provides a developer with a very convenient and handy feature — a default logging domain for each logging call. We assume that in a project workspace the directory structure follows the logical structure of the project. So we could create a domain for a logging call in a source file from the full name of the file as is specified in section 4.3 of GNU Coding Standard [17]. For example, if we report an error on line 5 in the file `src/main.cpp` the logging domain for this error becomes `src/main.cpp:5: error:`

Unfortunately matching cost is much higher for strings. If we match a logging domain each time we need to handle a logging event, this becomes a serious performance overhead.

In this article we show that string-based logging domains can be implemented with high performance. The key idea is that the logging filter rarely changes, so the change may be a costly operation. Instead of matching a set of filters against a logging domain for each logging event we suggest checking it only once, when the filter changes or a new logging domain is added.

This approach could be easily implemented in any language which supports local storages with static storage duration. In the next section we will go in detail through our C++ implementation.

Implementation

Logging interface is logically divided into two following parts. The first part consists of calls which produce logging events. We listed these calls and their descriptions in Table.

Another group of logging API allows manipulation with a set of filters. It is just STL-like set with possibility to add or remove a filter string.

Logging macros which should be affected by filtering are based on the macro presented in Fig. 2. For each place where the macro is expanded a compiler reserves memory for a `LogSite` instance with static storage duration. It has a bit field which indicates if this logging site is enabled, i. e. if it should produce any logging output.

When a control flow triggers a `LogSite` constructor, the constructor registers this log site in a list of all log sites, and initial value of the flag is set to true if at least one of filter strings matches the domain of this site. If the filter set changes, we run over the list of log sites and recalculate the flags.

In terms of numbers a simple check of the bit is 15 times quicker than `strncmp` comparison with `org.apache.harmony` domain name. A real program contains some sensible code in addition to logging calls, so the performance benefit for the real program is smaller. But it is nice thing to get for free anyway.

Let's discuss some variations of the implementation. These variations do not have an evident advantage over the proposed design, but for some special case they could fit better.

Logging Interface

Name	Behavior
`DIE(message)`	Outputs an error message and aborts execution
`WARN(message)`	Outputs a warning message
`INFO2(domain, message)`	Issues a message if the domain string matches a runtime filter
`INFO(message)`	Outputs general information
`TRACE2(domain, message)`	Issues a debug message if the domain string matches a runtime filter. Cleaned up from a release version
`TRACE(message)`	Issues a debug message with a default domain for the region of code where the call appears. Cleaned up from a release version

```
/**
 * Issues a log message with a given logging domain. <code>ls</code> class
 * instance is used to cache if a filter matches for this call site.
 */
#define TRACE2(domain, message) { \
    static util::LogSite ls(domain, LOG_FILELINE); \
    if (ls.is_enabled()) ls.get_entitled_log() << message << "\n"; \
}
```

Fig. 2. Log Site in C++

```
/** State of a log site. */
enum log_site_state {
    DISABLED, UNINITIALIZED, ENABLED
};

/** Creates a log site. */ #define TRACE2(domain, message) { \
    static enum log_site_state ls = UNINITIALIZED; \
    if (ls != DISABLED) { \
        if (ls == UNINITIALIZED) log_register(&ls, domain, LOG_FILELINE); \
        if (ls == ENABLED) log_printf message; \
    } \
}
```

Fig. 3. Log Site in ANSI C

Using advanced macro techniques it is possible to generate a list of log sites at compile time. For a program which has plenty of dead, or rarely used code this solution is of questionable benefit, because dynamic registration of a log site in runtime is a very cheap operation. Anyway there is an interesting theoretical question if this could be implemented by means of macro and template support specified in C++ standard.

The log site concept could be implemented in a language like ANSI C which lacks constructors. The sketch is shown in Fig. 3. In this case we use `UNINITIALIZED` value of `ls` variable to specify that the logging site has not been registered yet. In the following conditional operator we check the flag for this value, and perform registration if needed. From our point of view an object-oriented approach is more vivid, but from performance perspective there is no actual difference if a good compiler is used.

In a multi-threaded environment operations with the global set of filters, and the list of log sites should be made thread safe, for example, by means of read/write locks. For such case the proposed solution gives an additional performance benefit because no synchronization is needed for all disabled log sites.

Now let's modify this example to meet an additional requirement that a logging site state depends on a current thread. For example, this requirement arises from Java invocation API [18] which requires coexistence of multiple Java virtual machines in the same address space with different runtime logging settings. In this case we cannot just check a state flag in a local storage for each call because we need to take thread information into account. The solution is to keep in a local storage a unique logging site number `site_id`, and move state flags in a dedicated array which is specific for logging a context. The typical logging call evaluates the array first, and then uses a logging site number as the index `getLog(pthread_self())->enabled[site_id]`.

Studying how a developer uses debug output we could see that intelligent filters are open to further improvements. For example, a sophisticated filter should take an advantage of garbage collection techniques [19], i. e. it could be configured to forget excessive information in safe points. Here is an example from practice — if one investigates a fatal error, he is interested in a tail of a log. So the logging design theme is not yet finished.

Conclusion

We have shown how the concept of self-registering logging sites is used to implement a high performance logger. Each place which emits a log message stores a boolean flag. The flag indicates if logging output is produced by this logging site. This flag is calculated when the logging site registers in a logging site list, and recalculated when the filter changes. This allows creating of a logging system with no or little performance impact.

Acknowledgments

The author is grateful to Salikh Zakirov, Ivan Volosyuk, Valery Olshansky, Gregory Shimansky, Ilya Berezhniuk, Ricardo Morin and Pavel Pervov for useful technical discussions.

With special thanks to Brynell Somerville and John Kinsky for editorial input.

References

[1] Goerzen J. Finding stubborn bugs with meaningful debug info // Linux J. 2005. Vol. 129. Also available as `http://www.linuxjournal.com/article/7747`

[2] Good M. The use of logging data in the design of a new text editor // Proceedings of the SIGCHI conference on Human factors in computing systems. ACM Press, 1985. P. 93–97.

[3] Stockinger H. et al. On-Demand Grid Application Tuning and Debugging with the NetLogger Activation Service // IEEE Computer Society. 2003.

[4] Yang G. Build a distributed logging framework using java RMI // DevX.com. `http://www.devx.com/java/Article/7999` 2002.

[5] Finlayson R., Cheriton D. Log files: an extended file service exploiting write-once storage // Proceedings of the eleventh ACM Symposium on Operating systems principles. ACM Press, 1987. P. 139–148.

[6] Hagmann R. Reimplementing the Cedar file system using logging and group commit // Proceedings of the eleventh ACM Symposium on Operating systems principles. ACM Press, 1987. P. 155–162.

[7] Keen J.S., Dally W.J. Extended ephemeral logging: Log storage management for applications with long lived transactions // ACM Trans. Database Syst. 1997. Vol. 22(1). P. 1–42.

[8] Hamilton G. et al. JSR 47: Logging API specification // `http://jcp.org/en/jsr/detail?id=47`

[9] Field R. et al. JSR 163: JavaTM platform profiling architecture // `http://jcp.org/en/jsr/detail?id=163`

[10] Tayal A. Logging with Apache Commons logging // `http://carbon.sourceforge.net/modules/core/docs/logging/Design.html`

[11] Ossher H. et al. Aspect-oriented software development conference // `http://aosd.net/`

[12] HANNEMANN J., KICZALES G. Design pattern implementation in Java and AspectJ // Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM Press, 2002. P. 161–173.

[13] LADDAD R. Simplify your logging with AspectJ // `http://www.developer.com/java/other/article.php/3109831` 2003.

[14] MILES R. AspectJ Cookbook // O'Reilly & Associates, Inc. 2004.

[15] GILSTRAP B.R. An introduction to the java logging API // `http://www.onjava.com/lpt/a/2406`

[16] SAJIP V. A logging system for python // `http://www.red-dove.com/python_logging.html`

[17] STALLMAN R. ET AL. GNU coding standards // `http://www.gnu.org/prep/standards/`

[18] LIANG S. Java native interface // `http://java.sun.com/j2se/1.4.2/docs/guide/jni/`

[19] AHN J., MIN S.-G., HWANG C.-S., YU H. Efficient garbage collection schemes for causal message logging with independent checkpointing // J. Supercomput. 2002. Vol. 22(2). P. 175–196.

Received for publication 2 May 2007