RebelLabs

Imagine a bacon-wrapped Ferrari. Still not better than our free technical reports.

Attack of the logs! Consider building a smarter log handler

December 17, 2013 Oleg Shelajev No comments

The difference between logging more and logging smarter

When it comes to firefighting almost any software issue, logs are essential (and here's a great story about that). However when everything is running smoothly, logging often just takes lots of space on your drives, decreases performance of your system and annoys your system administrators because it implements rotating incorrectly.

I'm not talking about those logs necessary to audit your system, but those that shed a light on the internal state of the system when something breaks and you're in charge of getting everything working again.

Naturally, your application uses logging levels in a very sane way, where irrelevant information is tagged DEBUG and TRACE might lead you to find something interesting, but when it reaches an erroneous state it uses ERROR to be visible even in the most restricted logging settings.

So here is the challenge: how can you find out what occurred in the system before the actual error was encountered? Something definitely happened, but to understand the problem it's extremely important to find out what started the issue in the first place. However, if you're running the system with INFO level logs only, everything important is swallowed.

There are two natural solutions to this problem: more logging and smarter logging. I've often seen more logging occur by adding more INFO level logs, or by running the system with DEBUG enabled for a period of time after a new release.

But I want to talk about how we could log smarter using what we can call "opportunistic log handling". This is basically an approach that can get us all the logs we need, when we need them and keep them out of the way when we don't. What if we could make our logger smarter, so that it could provide opportunistic information based on time periods or special events and create actual log files when and if a need occurs?

I know what you're thinking: Yes, this will require some memory for each logger instance, but that is a question of scalability. So before we show you how this can be done, let's first look at what advantages you can gain by this approach…

Advantages and applicability of opportunistic log handling

One of the clearest advantages to this approach is that we won't populate log files with unnecessary information. It will make debugging and monitoring easier, as everything in the logs related to specific components will appear only when actual errors occur.

The obvious tradeoff here is the increased memory consumption, as we have to keep messages that are accumulated, but not yet flushed, in memory.

Let's assume that an average log message is 160 characters long. This is quite an optimistic claim, but hey, if 140 characters is enough to convey a scientific discovery, add some padding for message source and 160 should be enough for everyone.
By taking the napkin from underneath my beer, I can quickly see that I can have over 6500 log messages at the expense of just 1 megabyte.

(1 megabyte) / (160 bytes) =

6 553.6

Obviously this is not acceptable to every system, but if you run something smaller than yahoo or reddit, you can probably consider it. I certainly thought about it, and decided to move forward–so let's get our hands dirty and throw together a sample implementation to see how it feels.

Logbook's FingersCrossedLogger

Here is an example of something along the lines of what we're talking about here. FingersCrossedLogger is a Python class from a Logbook library that actually made me think about optionally logging events that happened before the actual error.From its documentation, we get the following.

> *This handler wraps another handler and will log everything in memory until a certain level (action_level, defaults to ERROR) is exceeded. When that happens the fingers crossed handler will activate forever and log all buffered records as well as records yet to come into another handled which was passed to the constructor.*

You can check out the source code for that on GitHub. A quick search reveals that the same concept is available in Monolog for PHP. However, I'm mainly interested in Java, so let's try to port that to the JVM.

Applying the concept to Java

When it comes to logging, SLF4J appears to be the de facto specification for logging at the abstraction level. A previous RebelLabs post that surveyed a small group of international developers found the following results:

Tweets Follow

Attila Szegedi @asz 6h
Wikipedia's 2013 fundraising is nearing end and it's $3 million short of its $20 million goal - chip in if you can!
donate.wikimedia.org
Retweeted by RebelLabs
Expand

Erik Meijer @headinthebox 4h
What I love about Scala is that, unlike most typed languages, I can *always* write what I want, and the type system never runs out of steam.
Retweeted by RebelLabs
Expand

RebelLabs @RebelLabs 14h
Which #Gradle plugins do you like? We have a short list of our favorites in this report on Build Tools:
0t.ee/J4QieO @Gradleware

REPORTS POSTS

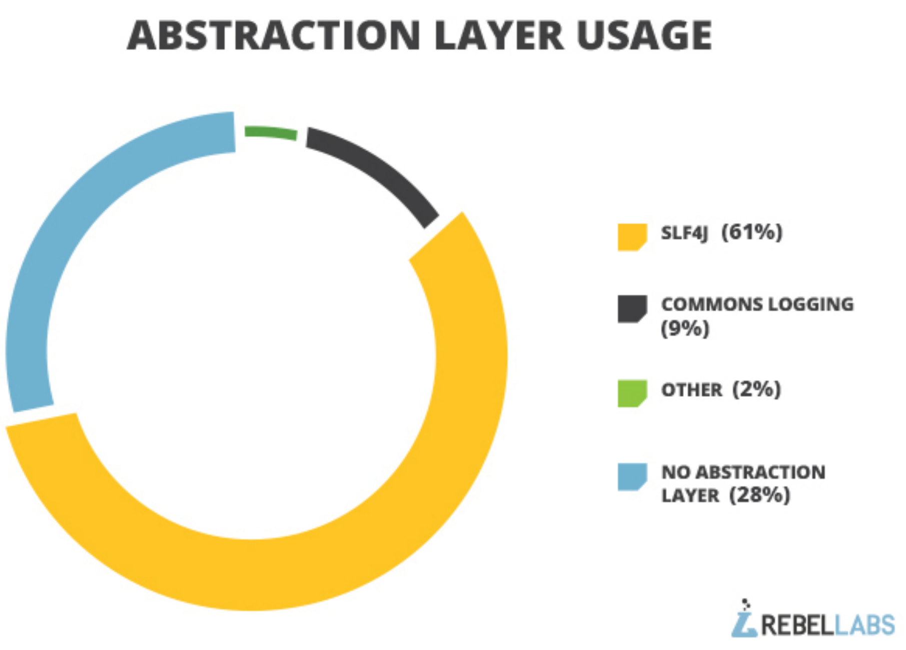
Developer Productivity Report 2013 – How Engineering Tools & Practices Impact Software Quality & Delivery
18 September 2013

The Curious Coder's Java Web Frameworks Comparison: Spring MVC, Grails, Vaadin, GWT, Wicket, Play, Struts and JSF
30 July 2013

The Great Java Application Server Debate with Tomcat, JBoss, GlassFish, Jetty and Liberty Profile
21 May 2013

Tags

continuous delivery	21
development	13
DevOps	23
eclipse	16
GeekOut	14
java	68
JBoss	15
jenkins	15
Jevgeni Kabanov	14
JRebel	74
JVM	15
LiveRebel	41
Maven	13
Press Release	23
productivity	26
rebelcast	16



Naturally, I want my implementation to be compliant with the slf4j-api artifact, so the first thing is to satisfy that.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.7.5</version>
</dependency>
```

And then we just need to define a subclass of org.slf4j.Logger, which is the main user entry point of slf4j-api.

A straightforward implementation doesn't look complicated at all, however one thing is bothering me a little. The logger doesn't declare levels for messages; instead, there are specific methods for: info(), debug(), trace(), which works great for the API because it is more portable across frameworks. However, an opportunistic logger needs a specific marker to know when to stop collecting messages to memory and start flushing. I thought about using *LocationAwareLogger.XXX_INT* constants, which are available in the same API jar, but it seems too unrelated, so there is a dedicated enum for that.

All in all, the definition looks something like this.

```
public class OpportunisticLogger extends MarkerIgnoringBase {
  private final Logger delegate;
  public final Level triggerLevel;
  private LinkedList messages = new LinkedList();
  private volatile boolean triggered;

  public OpportunisticLogger(Logger delegate, Level triggerLevel) {
    if (delegate == null) {
      throw new IllegalArgumentException("Delegate logger shouldn't be null
    }
    this.delegate = delegate;
    this.triggerLevel = triggerLevel;
  }

  private void tryTriggering(Level lvl) {
    if (!triggered && lvl.level <= triggerLevel.level)
      trigger();
  }

  private void trigger() {
    for (String msg : messages) {
      delegate.info(msg);
    }
    triggered = true;
  }

  private void log(String msg) {
    if (triggered) {
      delegate.info(msg);
    }
    else {
      messages.add(msg);
    }
  }
  …
}
```

The messages store is just a linked list; when messages are retranslated to an actual logger, they all go as info messages. Alternatively, we can figure out which level is enabled in the delegate instance and use that. All in all the implementation is very straightforward.

Check out the whole project on GitHub.

Ok, let's try to improve performance

The one shortcoming of such an approach to logging is of course heavy memory use to store all the messages. One way to improve the implementation in that area is to use a smarter data structure for them.

A trie data structure comes to mind immediately. It is a tree where a common ancestor of nodes describes a common prefix of said nodes. Why is it good? Storing many strings can take less memory when using a trie, because the space is reused between several instances.

However, as far as I know there is no trie implementation that is considered a default one. Java standard library doesn't have this at all, and although there was something in Google Collections, it didn't end up making it to Guava.

Of course there are several easily "Googleable" implementations, but it's always tricky with randomly-found data structure implementations. It's so easy to skew the performance, that I personally tend to believe in battle-tested libraries only.

On another note, you can always switch to a more performant logging implementation in general, which would be more relevant to the performance of logging in general than to opportunistic loggers.

For example I've consider trying out AsyncLoggers from log4j2 for some time now and as I'm dealing with logs anyway, it feels like a right time to do so. For lazy readers who don't want to click through links, AsyncLoggers uses LMAX Disruptor to enable higher throughput in a multi-threaded setting.

It probably won't make your opportunistic logs better or more usable, if you find that the performance of the latter is not acceptable for your needs, but it's always good to know there are options.

Conclusion and open questions

When starting this post, I just wanted to share the idea of FingersCrossedLogger because of its cool name. But it turned out that it led me to other interesting questions:

- Have you thought about something like an "Opportunistic Log Handler" in your projects?
- Do you log information per process if something goes wrong or do you prefer to log everything?
- How would you implement a trie in Java?
- What is your general go-to source for all kinds of algorithmic snippets?

In general, it pays to take some time to figure out your logging framework–even if it doesn't make sense to deal with logging performance in a small application, but when the need arises it's easier to switch logging off in production than to figure out if it can be made usable when it's already too late.

Please leave your comments below, or tweet me @shelajev or @RebelLabs.

OLEG SHELAJEV
Product Engineer
The Software Engineer in ZeroTurnaround. Oleg likes to solve complex problems despite of their origin: algorithmics, chess, cryptography or social relationship. Believes in miracles. Likes pizza, beaches and new things that can be taken apart to see how they're done.

development, java, logging, open_problem

8 comments ★ 1

Join the discussion…

Best Community Share Login

Rex Sheridan • a day ago
Oleg, I don't understand the rationale behind logging forever after the event is triggered. It seems like there should be a mechanism to turn it off after a while, though I am not sure what that mechanism would be since basing it on elapsed time would be rather crude.

I also wonder about the tradeoffs of using a trie in this case. While they definitely perform better than many datastructures you may find that their algorithmic complexity could hinder performance. In the case that your production system is already suffering a from a performance problem the could not only exacerbate the problem but also confound diagnosis of the problem.

In addition, it could be useful to bound the LinkedList used for storing the messages and throw away old messages once the list is full.

All in all, I think this is a great idea. Are you guys considering putting this in one of your products. Your company isn't really known for building application libraries but I think it would be good for you to expand into other market niches as well. Good luck and thanks for building JRebel. It saves me a ton of time everyday.
2 • Reply • Share ›

gbevin → Rex Sheridan • a day ago
Rex, great comments, just chiming in about the application libraries. We at ZeroTurnaround have been putting out some parts of our underlying features in two distinct libraries: zt-zip (easily handle file archives) and and zt-exec (easily run external processes). Both are available on GitHub: https://github.com/zeroturnaro... and https://github.com/zeroturnaro...
1 • Reply • Share ›

Oleg Šelajev → Rex Sheridan • a day ago
Rex, those are great comments about trie performance and bounding the linkedlist in this implementation. Obviously this is not a piece of code that you would put into a real system, but just an illustration of a concept. Although it works.

To my mind, logging forever after triggering is a good thing. I see using these kinds of loggers for specific events in the application. For example, serving a single request can make use of 1 instance of such logger, or processing some background task or any isolated event that you would like to observe as a whole. Then when event is finished, you just dispose the logger. At least that was the idea.

Trie operations are linear in length of the input, so perfomance hit can be acceptable. However that is entirely application specific. Most probably you have some underused parallelism in the system and that could be put in use here, async handlers can be very performant.

But, to be fair, all of this is just theoretical, I haven't profile this and I don't even have a thorough implementaion of such logger. But I would definitely put that in use in some projects.

1 ^ | ˅ • Reply • Share ›

David Leppik • 2 days ago

I've been doing this for years in Java; I call it a TransientLog. It's incredibly useful. Unfortunately, I had to write it entirely from scratch, since it's so much different from how Log4J approaches logs. The main value is the ability to tell where the program is in its execution: not the stack, but the call progress.

Because of that, if I were to write it again, I'd make it hierarchical, similar to timer traces, on the theory that once you've dived down a call stack and come back up, you no longer need to know what happened during the dive.

2 ^ | ˅ • Reply • Share ›

Oleg Šelajev → David Leppik • a day ago

Hierarchy is a great idea, however there can be drawbacks, if I understood the intent correctly. Imagine a code that does something like:

Object a = doStuff(params); // might return null occasionally
doSomethingElse(a); // will throw NPE

Then hierarchical logs will omit everything useful that happened during doStuff() and you will just get notified that there is an NPE.

However, it's definitely worth thinking about a hierarchical approach more.

^ | ˅ • Reply • Share ›

David Leppik → Oleg Šelajev • 10 hours ago

That's true. The few times that I've done hierarchical logging, it's been done using try/finally blocks to push/pop. (Wrapped in a closure, if I'm using Scala.) This makes it fairly coarse-grained, which is appropriate for this sort of thing. In your example, doStuff(params) would only be omitted if it explicitly pushes/pops a log-- which would be a signal that it does long, complicated processing. So yes, it's a trade-off.

My original inspiration was a debugger that someone developed in the 1990s as computer science research. The debugger logged every command as it executed, so you could step through the code forwards or backwards in time. Presumably this never caught on because it is incredibly slow and memory-consuming.

^ | ˅ • Reply • Share ›

Oleg Šelajev → David Leppik • 10 hours ago

Yeah, it is a trade-off. Probably in a more monad-appropriate language than java it is easier to do this right.

About the debugger, the idea somewhat sticked still, Chronon does flight recording and allows back-in-time execution too: http://chrononsystems.com/

^ | ˅ • Reply • Share ›

Guest • 21 hours ago

Different log-files for each log level, with the ones for more verbose log levels on a ramdisk, rotated frequently & discarded early.

1 ^ | ˅ • Reply • Share ›

ALSO ON ZEROTURNAROUND WHAT'S THIS?

First vJUG online "meetup" with Trisha Gee freakin' rocked
1 comment • 24 days ago
Oleg Šelajev — It was an awesome session. Execution topped my expectations, I'm certaily looking for …

10 Reasons Why Java Now Rocks More Than Ever: Part 1 – The Java Compiler
40 comments • 3 months ago
nbevans — The Java Compiler: no structs/value-types, no lambdas, to start naming its flaws.The Core API: …

JRebel 5.4 Released – Updates for WebLogic, WebSphere, RESTEasy …
4 comments • 3 months ago
arhan — You can use "-y" argument with the script to answer "Yes" to all the confirmations automatically: ./setup.sh …

The 2014 Decision Maker's Guide to Java Web Frameworks
21 comments • a month ago
salk31 — GWT is insecure because the client is JavaScript!? It is the client technology. Are the other frameworks …

Subscribe Add Disqus to your site

Company

Our story
Meet the Team
Careers
Contact Us

Resources

JRebel White Paper (PDF)
LiveRebel Fact Sheet (PDF)
2012 Devs Productivity Report
2013 IT Ops & DevOps Productivity Report