Debugging and Observing Your Scala Code

Will Sargent

@will_sargent

https://tersesystems.com

Context

You're running services in the cloud.

Suddenly a wild exception occurs!

```
akka.pattern.AskTimeoutException: Ask timed out on
[Actor[akka://ask-pattern-timeout-warning/user/dumb-actor#7781
after [2000 ms]
```

What do you do?

Observability Will Detect!

- But the overall health of the system is fine...
- ...so it won't care.

Observe the Unobservable

- Observability has a hard time with non-events
- Show number of emails being sent!
- But an email that doesn't get sent?
- Data never written to database?

Needle In A Haystack

- The Art of Finding a Needle in a Haystack
- Distributed Tracing only covers so much
- Internal State is the Hidden Menace
- Input + ??? = Output
- Especially bad with state over multiple operations

Developers Want Logging

printf debugging may not be the most fun at a party but they're always there when you need them.

@mvsamuel

There is no software issue so profound, complex, or inexplicable that it cannot be resolved via a surfeit of print statements.

@ericasadun

Developers Really Want Logging

The best invention in debugging was still printf debugging.

On the Dichotomy of Debugging Behavior Among Programmers

SWEs are more likely to consult logs earlier in their debugging workflow, where they look for errors that could indicate where a failure occurred.

Debugging Incidents in Google's Distributed Systems

Perception

- Logging is Expensive
- Debug Logging is Super Expensive!

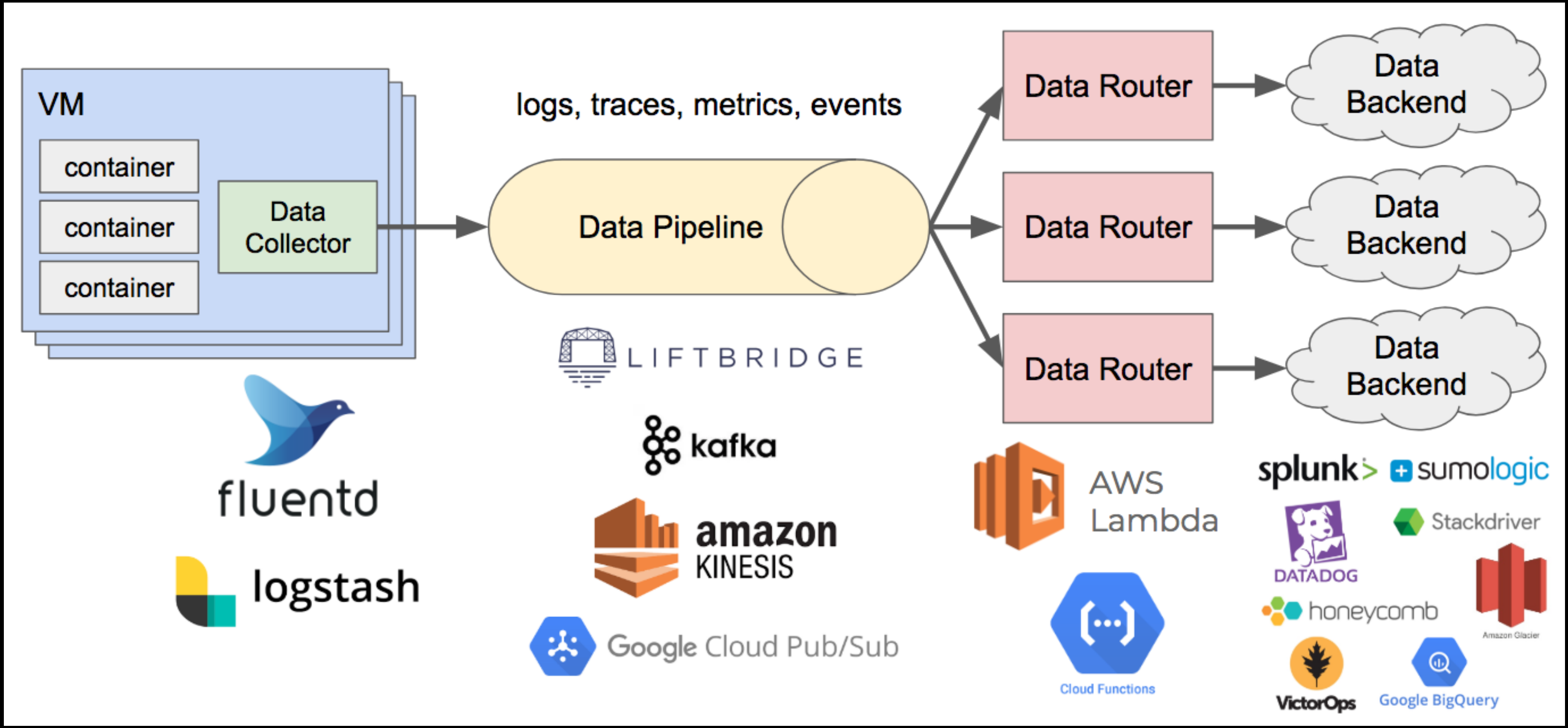
Reality

- Logging is Cheap!
- Observation is Expensive!

Logging is Cheap

- CPU overhead? Trivial.
- Disk storage? You're not keeping this long term.
- Blocking threads? Use async appender.
- IO Bandwidth? Memory-mapping: 803 inserts per ms.
- Still worried about writing to disk? Use tmpfs.
- Allocation Pressure? Not in the newer JVMs.

Observation Is Expensive



The Observability Pipeline

Observation On Demand

- Write Structured Logging in JSON
- Write to mem-mapped SQLite using Blacklite
- Turn on litestream to replicate to S3 (as needed)
- Pull from S3 and attach using Jupyter Lab.
- Do Science To It.

Instrumentation

- 3rd party library that doesn't have logging?
- No problem.
- We can inject logging with instrumentation.

Backtracing

- Application can query SQLite DB using JSON API
- On exception, query by correlation id...
- And attach logs as breadcrumbs to error reporting.

Terse-Logback

- Instrumentation
- Relative Nanos
- Unique ID
- Correlation ID

Demo Time!

https://terse-logback-showcase.herokuapp.com

SLF4J Predates Structured Logging

- 1999 log4j
- 2005 SLF4J
- 2006 Logback
- 2006 Logging in JSON
- 2011 Logs for Machines in JSON
- 2012 Log4J 2
- 2013 logstash-logback-encoder

How Can We Make This Easier?

```
 // creation is verbose
val marker = LogstashMarkers.append("correlationId", cid)

// thread local map of String, async problem
MDC.set("correlationId", cid)

// explicit conditional with marker
if (logger.isLoggingDebug(marker)) {
  // explicit marker, person calls toString implicitly
  logger.debug(marker, "Hi there {}", person)
}

// Eats exception because it's an argument
logger.error("Oh noes {}", ex)
```

Blindsight

- DSL and Type Classes For Structured Logging
- Contextual Logging
- Conditional Logging
- Flow Based Logging and Tracing
- Script Conditions
- "printf debugging" intention macros

Demo Time

Turning Code from SLF4J to Blindsight!

https://github.com/tersesystems/blindsight-starter

DSL and Type Classes

```scala
case class Person(name: String, age: Int)
implicit val personToArgument = ToArgument[Person] { person =>
  import DSL._
  Argument(
    ("name" -> person.name)
    ~ ("age_year" -> person.age)
  )
}
logger.info("person is {}", Person("steve", 12))
```

Building up Context

```scala
import com.tersesystems.blindsight._
import DSL._

val correlationId: String = "123"
val clogger = logger.withMarker(
  bobj("correlationId" -> correlationId)
)
clogger.info("Logs to JSON w/ a correlationId field")
```

Conditional Logging

```
def condition: Boolean =
  java.lang.Boolean.getBoolean("debug.enabled")

logger
  .withCondition(condition)
  .debug("Only logs when condition is true")

logger.debug.when(condition) { log => log("when true") }
```

Conditions are Powerful!

What’s really for us is to be able to real-time turn on and off of what log stuff you’re collecting at a pretty granular level.

Conditions

- By Feature Flag
- By Tracer Bullet
- By Circuit Breaker
- By Time
- By Script

Source Aware Scripts

```scala
package exampleapp
class MyClass {
  val sm     = new ScriptManager(scriptHandle)
  val logger = LoggerFactory.getLogger

  val location = new ScriptBasedLocation(sm, default = false)
  def logDebugSpecial(): Unit = {
    // location.here invokes implicit sourcecode macros
    logger.debug.when(location.here) { log =>
      log("This will log because of method name!")
    }
  }
}
```

Tweakflow script

```
library blindsight {
  function evaluate: (long level, string enc,
                      long line, string file) ->
    if (enc == "exampleapp.MyClass.logDebugSpecial") then
      true
    else
      (level >= 20); # info_int = 20
}
```

Flow Based Logging

```
implicit def flowB[B: ToArgument]: FlowBehavior[B] = ...

def flowMethod(arg1: Int, arg2: Int) = flowLogger.trace {
  arg1 + arg2
}
```

Useful for tracing entry / exit methods

Flow with Duration

```
class DurationFlowBehavior[B: ToArgument]
  (implicit spanInfo: SpanInfo) extends FlowBehavior[B] {
  override def exitStatement(resultValue: B,
                             source: Source) =
    Some {
      val span = popCurrentSpan
      Statement()
        .withMarkers(Markers(markerFactory(span)))
        .withMessage(s"${source.enclosing.value} exit,"
                    + s"duration ${span.duration()}")
        .withArguments(Arguments(resultValue))
    }
}
```

Inspecting Code

What do people actually do in printf debugging?

Finding 5: Event logging serves three major purposes in the reference domain, i.e., state dump, execution tracing and event reporting.

Industry Practices and Event Logging: Assessment of a Critical Software Development Process

Inspecting Vals

```scala
val fn = { dval: ValDefInspection =>
  logger.debug(s"${dval.name} = ${dval.value}")
}
decorateVals(fn) {
  val a = 5
  val b = 15
  a + b
}
```

Inspecting Ifs

```
decorateIfs(dif => l.debug(s"${dif.code} = ${dif.result}")) {
  if (System.currentTimeMillis() % 17 == 0) {
    println("branch 1")
  } else if (System.getProperty("derp") == null) {
    println("branch 2")
  } else {
    println("else branch")
  }
}
```

Inspecting Fields

```
class ExampleClass(val someInt: Int) {
  protected val protectedInt = 22
}
val exObj        = new ExampleClass(42)
val publicFields = dumpPublicFields(exObj)

logger.debug(s"exObj public fields = $publicFields")
```

Where Do You Start?

When to Log

- For libraries, at least INFO or WARN.
- Libraries can be instrumented, but may lack context.
- Change Logback levels with `ChangeLogLevel`.
- Use conditional logging for your application.
- Log application code at TRACE to disable filtering.
- Ideally, use conditions you can change at runtime.

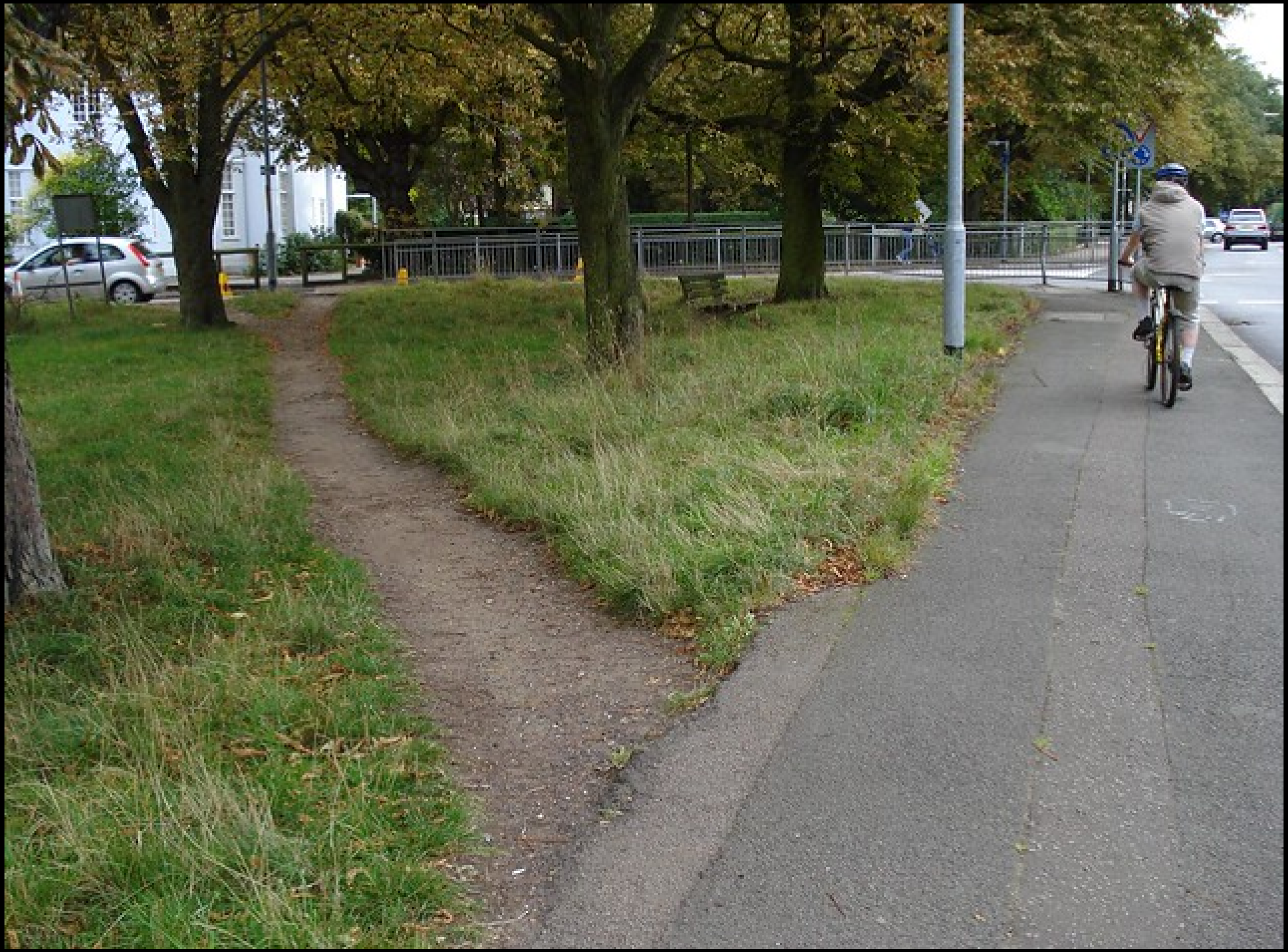
Special Condition

- If you want to run expensive queries...
- If you want to dump lots of info...
- If you want to sample at TRACE...
- If you want to suppress useless errors...

...use a special condition.

What To Log?

Desire Paths



by kake pugh

Logging as a desire path

When you (think) you're done with a logging statement, resist the temptation to delete it or put it in a comment. If you delete, you lose the work you put to create it. If you comment it out, it will no longer be maintained, and as the code changes it will decay and become useless. Instead, place the logging statement in a conditional. (p111)

Effective Debugging

Logging as Guide

Logs should contain sufficient information to help with the reconstruction of critical state transitions.

Neal Hu, Logging Like a Pro

Log as if your program was writing a journal of its execution: major branching points, processes starting, etc., errors and other unusual events.

Logging Wisdom: How To Log

Diagnostic Logging

- Debug Logging With An Audience
- Domain level concepts and flow.
- Use diagnostic logging in place of comments.
- "If there were a bug, where would it be?"
- Explicitly pass context through async boundaries.

Your Voice In The Machine

- Diagnostic logging is better than comments.
- Intuition, guidance, live walkthrough of code.
- What objects and what states are significant?
- What does this mean?

Good Books

- Why Programs Fail
- Effective Debugging

Useful Services

- Honeycomb
- Rookout
- Lightrun

Questions?