BY FUHENG WU

DevOps Best Practice in Real Production Environment

OBSERVABILITY IN ACTION

Master Microservices and Machine Learning System Observability

First Edition

With Zipkin, Jaeger, Prometheus, OpenTelemetry, HiQ

WWW.WU-99.COM

To the peace of the world!

> *Imagine there's no countries. It isn't hard to do. Nothing to kill or die for. And no religion, too. Imagine all the people. Living life in peace.*
>
> *John Lennon*

http://www.wu-99.com

Fremont, California
United States, 94555
Feb 24, 2022

Observability In Action, First Edition

Fuheng Wu

CONTENTS

1 Introduction To Observability **7**

1.1 Observability Overview 8

1.1.1 What is Observability? 8

1.1.2 History Of Observability 9

1.1.3 Profiling Tool 10

1.1.4 Logging System 11

1.1.5 Monitoring System 12

1.1.6 Tracing System 13

1.1.7 Observability 14

1.2 Observability In Microservices 15

1.2.1 System Design of Distributed Monitoring System 15

1.2.2 System Design of Distributed Tracing System 16

1.3 Observability In Machine Learning System 17

2 Zipkin **19**

2.1 Binary and Bits 20

2.1.1 Creating a git repository 21

2.1.2 Further commits 24

2.1.3 git reset 26

3 Jaeger **29**
3.1 Quick Start . . . 30
3.2 Jaeger Components And Setup . . . 31
3.2.1 Jaeger Architecture . . . 31
3.3 Jaeger Persistent Storage . . . 32
3.3.1 Cassandra . . . 32
3.3.2 Elasticsearch . . . 33
3.4 Jaeger Collector . . . 34
3.5 Jaeger Persistent Storage . . . 35
3.5.1 Cassandra . . . 35
3.5.2 Elasticsearch . . . 36
3.6 Jaeger Persistent Storage . . . 37
3.6.1 Cassandra . . . 37
3.6.2 Elasticsearch . . . 38
3.7 Jaeger Persistent Storage . . . 39
3.7.1 Cassandra . . . 39
3.7.2 Elasticsearch . . . 40

4 Prometheus **41**
4.1 Quick Start . . . 43

5 OpenTelemetry **45**
5.1 Quick Start . . . 46

6 HiQ - Non-Intrustive Tracing For Python **47**
6.1 Quick Start . . . 48

CHAPTER

ONE

INTRODUCTION TO OBSERVABILITY

> *The purpose of a programming language is to help express ideas in code.*
>
> *---Bjarne Stroustrup*

Microservices architecture (often shortened to microservices) refers to an architectural style for developing applications. Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility.

1.1 Observability Overview

Observability exists at the same time software was born. Recent years, the adoption of microservices and AI technology makes performance monitoring more important then ever before. Now Observability has become the same important as *scalibility*, *availability*. In CNCF Cloud Native Interactive Landscape, **Observability and Analysis** is one of the largest categories.

1.1.1 What is Observability?

Observability is a system characteristic describing the degree to which a system can be understood from its external outputs. Measured by CPU time, memory, disk space, latency, errors, etc., computer systems can be more or less observable. To ensure there is no service disruption, you' ll need to observe and analyze every aspect of your application so every anomaly gets detected and rectified right away.

1.1.2 History Of Observability

Observability exists when software was born but the word *observability* becomes popular is something after microservices became a de-facto industry standard.

At the early stage of software industry, people need to *catch the exceptions* and related information when software runs so that they can debug and improve the quality of the software. At the very early stage, the software system will generate *dump file*, or some plain-text format *log file*. Those files are the result of the most primitive logging and monitoring system. Even today, we still use core dump file for debugging and root-causing. Later on, as the software system becomes more and more complicated, software performance becomes the main problem. The focus of system logging and monitoring swiched from exception analysis to performance analysis. This makes APM(application performance management) becomes an indisposable tool for large scale software system. APM is mainly used to detect and analyze the performance issue of application system. It can help developers and operation team to pinpoint the root cause and resolve the issues quickly, so that the service level objectives can be met. Application performance management is so important that it has become a business line for many companies such as IBM, Elastic.io and OCI(Oracle Cloud Infrastructure).

1.1.3 Profiling Tool

1.1.4 Logging System

1.1.5 Monitoring System

1.1.6 Tracing System

1.1.7 Observability

1.2 Observability In Microservices

1.2.1 System Design of Distributed Monitoring System

Architectural Design

Time Series Data

Time Series Database Design

1.2.2 System Design of Distributed Tracing System

Tracing Model

Sampling Strategy

Propagation

1.3 Observability In Machine Learning System

CHAPTER

TWO

ZIPKIN

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

2.1 Binary and Bits

Have you ever had the feeling that you want to make large scale changes to your code, but are afraid of breaking something? Has the thought crossed your mind that you could copy your source file or files to another directory, so that you have a backup in case something goes wrong?

If so then that's perfectly normal. You might, however, imagine that if you were to work on a project for a longer time, say, several weeks, the number of backup directories would become difficult to have an overview of, and if you actually had to revert some changes from the past, it might be difficult to remember which directory contained which changes.

Version control systems (VCS) have been implemented to help solve these issues, as well as help developers collaborate on the same source code.

Version control systems have a long history, and I won't go through all of that. Suffice to say, there are lots of different version control systems, and many developers have their own personal favourite.

This book will introduce the reader to one of the most useful ones, named git. Git was originally developed in 2005 by Linus Torvalds for the Linux kernel development, and has since become one of the most widely used version control systems worldwide.

Git has several upsides, and a significant one is how it scales: it's very practical to use in hobby projects of individual developers, but is also being used to version control some of the world's largest source code repositories.

There is lots that can be learned about git; this book will cover the basic fundamentals.

2.1.1 Creating a git repository

Let's assume you have a directory which is otherwise empty except a file you've written, hello.py, which contains "print 'hello world'". Let's turn this directory into a git repository:

```
$ git init
Initialized empty Git repository in /home/antti/book/my_
↪project/.git/
```

Git has now initialised an empty repository. What this means in practice is that git has generated a new hidden directory called .git which we can see by running the command "ls -la":

```
$ ls -la
total 16
drwxr-xr-x 3 antti users 4096 Feb  9 00:30 .
drwxr-xr-x 8 antti users 4096 Feb  9 00:30 ..
drwxr-xr-x 7 antti users 4096 Feb  9 00:30 .git
-rw-r--r-- 1 antti users   20 Feb  9 00:30 hello.py
```

What we want to do next is store the file hello.py in the git repository. After that it'll be version controlled. We can do this by running the following:

```
$ git add hello.py
```

This adds "hello.py" to *the staging area* which is an area describing what will be committed next. We can then run "git status" to see the current status:

```
$ git status
On branch master

```

(continues on next page)

(continued from previous page)

```
Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   hello.py
```

What this means is that if we were to commit our changes, i.e. create a new version, the new version would include a new file, namely hello.py. We can furthermore see the contents of the new commit by running "git diff –staged".

Let's create our commit:

```
$ git commit -m "initial commit"
[master (root-commit) b279b4f] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

The "-m" switch to git commit allows the user to enter the commit message, which should include a short description about what the commit is about. Since this is our initial commit, we label it as such.

After running the command, git tells us it's created a new commit with the new file included. What happens behind the scenes is that git stores the necessary information about your file in the .git directory, such that it can be indexed and retrieved again when needed.

If we wanted to see the status of our repository now, we can run "git status" again:

```
$ git status
On branch master
nothing to commit, working tree clean
```

This means that the latest version git has stored matches the contents of our files (in this case, hello.py).

2.1.2 Further commits

Let's now assume we want to make a change to our hello.py, by appending "print 'hello world'" to it:

```
$ echo "print 'hello world'" >> hello.py
```

Now that we've modified our file, we can check status again:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
↪working directory)

    modified:   hello.py

no changes added to commit (use "git add" and/or "git
↪commit -a")
```

The status now tells us that we've modified our file, and that the modifications aren't included in any version controlled by git.

We can view the changes we've made by running "git diff":

```
$ git diff
diff --git a/hello.py b/hello.py
index a968078..01283b8 100644
--- a/hello.py
+++ b/hello.py
@@ -1 +1,2 @@
 print 'hello world'
```

(continues on next page)

(continued from previous page)

```
+print 'hello world'
```

The output suggests that there's been a new line added to the end of hello.py.

We can now commit this change:

```
$ git add hello.py
$ git commit -m "add printing hello world again"
[master 43130e1] add printing hello world again
 1 file changed, 1 insertion(+)
```

Now we already have two commits. We can see the commit log by running "git log":

```
$ git log
commit 43130e10f89232f5ce542c4d864ff78e0a171796
Author: Fuheng Wu <wufuheng@gmail.com>
Date:   Fri Feb 9 00:42:07 2018 +0100

    add printing hello world again

commit b279b4fb109844ab0337bc906897f6e48a3c18cf
Author: Fuheng Wu <wufuheng@gmail.com>
Date:   Fri Feb 9 00:35:05 2018 +0100

    initial commit
```

The log will show the summary of each commit as well as the *commit hash*, which uniquely identifies each commit.

2.1.3 git reset

Now comes the interesting part: let's say we want to go back to the previous version, before we added the second "print 'hello world'" to the end of hello.py. One way to do this is the following:

```
$ git reset --hard b279b4f
HEAD is now at b279b4f initial commit
```

The command asks git to reset the state of the current working tree to the commit b279b4f (the first few characters of the commit hash we're interested in). Git does this, and as part of that, replaces our hello.py with the old version:

```
$ cat hello.py
print 'hello world'
```

If we look at the log, we see the previous commit is no longer there:

```
$ git log
commit b279b4fb109844ab0337bc906897f6e48a3c18cf
Author: Fuheng Wu <wufuheng@gmail.com>
Date:   Fri Feb 9 00:35:05 2018 +0100

    initial commit
```

If we wanted to get our changes back, we can, because we noted the commit hash:

```
$ git reset --hard 43130e10
HEAD is now at 43130e1 add printing hello world again
$ cat hello.py
print 'hello world'
print 'hello world'
```

This was a very short introduction to git. There will be more covered in this book later; you can also check the built in documentation for git by running "git –help" or the help for specific commands, for example "git status –help".

Exercise: Create a git repository and repeat the above commands yourself.

Exercise: Use git to version control all your previous and future software development projects.

CHAPTER

THREE

JAEGER

As a CNCF project, Jaeger was created by Uber and was written in Go. It's similar to, but also different from, its older sibling Zipkin. In addition to Zipkin's feature set, Jaeger also provides dynamic sampling, a REST API, a ReactJS-based UI, and support for Cassandra and Elasticsearch in-memory datastores. To implement these features, Jaeger takes a different, more distributed approach than Zipkin.

Jaeger's architecture includes a client that emits traces to an agent, which listens for inbound spans and routes them to the collector. The collector then validates, transforms and persists spans. Jaeger's distributed architecture makes it highly scalable. Jaeger also has a unique way of collecting data: unlike other systems that try to collect every trace and span generated, Jaeger takes a dynamic representative sample of the monitored data. This approach not only handles sudden surges in traffic, but increases Jaeger's overall performance.

3.1 Quick Start

From user point of view, Jaeger has no much differnce from Zipkin. Although Jaeger has more components than Zipkin, it has server-side and client-side components.

3.2 Jaeger Components And Setup

Jager server-side components includes collector, query, and ingester.

3.2.1 Jaeger Architecture

In contrast to Zipkin, Jaeger has two more components: agent and ingestor.

3.3 Jaeger Persistent Storage

Jaeger uses external services for ingesting and persisting the span data, such as Elasticsearch, Cassandra and Kafka. This is due to the fact that the Jaeger Collector is a stateless service and you need to point it to some sort of storage to which it will forward the span data.

The standard persistent storage for Jaeger is Cassandra. Later on, Elasticsearch was added in. These two storage types are the most popular one so far. In addition, Jaeger also supports other types.

3.3.1 Cassandra

...

3.3.2 Elasticsearch

...

3.4 Jaeger Collector

The Jaeger Collector is the component responsible for receiving the spans that were captured by the tracer and writing them to a persistent storage.

3.5 Jaeger Persistent Storage

Jaeger uses external services for ingesting and persisting the span data, such as Elasticsearch, Cassandra and Kafka. This is due to the fact that the Jaeger Collector is a stateless service and you need to point it to some sort of storage to which it will forward the span data.

The standard persistent storage for Jaeger is Cassandra. Later on, Elasticsearch was added in. These two storage types are the most popular one so far. In addition, Jaeger also supports other types.

3.5.1 Cassandra

...

3.5.2 Elasticsearch

...

3.6 Jaeger Persistent Storage

Jaeger uses external services for ingesting and persisting the span data, such as Elasticsearch, Cassandra and Kafka. This is due to the fact that the Jaeger Collector is a stateless service and you need to point it to some sort of storage to which it will forward the span data.

The standard persistent storage for Jaeger is Cassandra. Later on, Elasticsearch was added in. These two storage types are the most popular one so far. In addition, Jaeger also supports other types.

3.6.1 Cassandra

...

3.6.2 Elasticsearch

...

3.7 Jaeger Persistent Storage

Jaeger uses external services for ingesting and persisting the span data, such as Elasticsearch, Cassandra and Kafka. This is due to the fact that the Jaeger Collector is a stateless service and you need to point it to some sort of storage to which it will forward the span data.

The standard persistent storage for Jaeger is Cassandra. Later on, Elasticsearch was added in. These two storage types are the most popular one so far. In addition, Jaeger also supports other types.

3.7.1 Cassandra

...

3.7.2 Elasticsearch

...

CHAPTER

FOUR

PROMETHEUS

Prometheus, a Cloud Native Computing Foundation project, is a systems and service monitoring system. It collects metrics from configured targets at given intervals, evaluates rule expressions, displays the results, and can trigger alerts when specified conditions are observed. Prometheus by design implements a pull-based approach for collecting metrics.

The features that distinguish Prometheus from other metrics and monitoring systems are:

- A multi-dimensional data model (time series defined by metric name and set of key/value dimensions)
- PromQL, a powerful and flexible query language to leverage this dimensionality
- No dependency on distributed storage; single server nodes are autonomous
- An HTTP pull model for time series collection
- Pushing time series is supported via an intermediary gateway for batch jobs
- Targets are discovered via service discovery or static configuration
- Multiple modes of graphing and dashboarding support

- Support for hierarchical and horizontal federation

4.1 Quick Start

From user point of view, Jaeger has no much differnce from Zipkin. Although Jaeger has more components than Zipkin, it has server-side and client-side components.

CHAPTER

FIVE

OPENTELEMETRY

OpenTelemetry is an open source project and unified standard for service instrumentation, or a way of measuring performance. Sponsored by the Cloud Native Computing Foundation (CNCF), it replaces OpenTracing and OpenCensus. The goal is to standardize how you collect and send telemetry data to a backend platform. The OpenTelemetry project consists of specifications, application programming interfaces (APIs), libraries, and integrations, including software development kits (SDKs) for various languages like Java, Go, and Python. It also defines a centralized collector service that you can use for collecting telemetry data from your applications and services. It includes exporters to send that data to the observability platform that you choose.

5.1 Quick Start

From user point of view, Jaeger has no much differnce from Zipkin. Although Jaeger has more components than Zipkin, it has server-side and client-side components.

CHAPTER

SIX

HIQ - NON-INTRUSTIVE TRACING FOR PYTHON

HiQ is a declarative, non-intrusive, dynamic and transparent tracking and optimization system for both monolithic application and distributed system. It brings the runtime information tracking and optimization to a new level without compromising on performance or losing insights.

6.1 Quick Start

From user point of view, Jaeger has no much differnce from Zipkin. Although Jaeger has more components than Zipkin, it has server-side and client-side components.

Contact author: wufuheng@gmail.com