Fault Tolerance

Haoren Zheng

July 22, 2020

Abstract

Contents

1 Introduction

2 Background

2.1 Server Language

2.1.1 Erlang

Erlang is general-purpose, structured, concurrent, dynamically typed programming language for concurrency. Originally designed by Ericsson for communications applications such as control switches or transformation protocols, it is ideal for building distributed, real-time soft parallel computing systems. An application run time written in Erlang usually consists of thousands of lightweight processes that communicate with each other through messaging.

Different from traditional object-oriented language, it has its own advantages: first, it is based on the concurrent process, this process is lightweight and collaborate organized handle affairs, users don't have to worry about specific PV operation, the process is transparent for the operating system, only one process for the operating system is running. Second, each process has its own

independent memory and is completely dependent on messaging in inter-process communication. Each process has its own independent mailbox and finds messages to process through pattern matching, and then asynchronously processes them separately. This reduces coupling between processes and increases independence. Erlang also have reliable fault-tolerant mechanism, due to the relatively independent between processes, and can be used in the Erlang some process to link or monitor other process, when the monitored process because of the errors and produce abnormal exit, responsible for the monitoring program will receive the news of the process exits and corresponding processing on these processes. In OTP, Erlang can perform one-to-one or one-to-many monitoring using a monitoring tree. Most importantly, the support for multicore CPUs in Erlang does not require developers to manage the operation of multicore, it is completely transparent to developers, we just need to write programs as usual. Finally, Erlang supports hot code upgrades. In Erlang, code upgrades can be carried out without downtime to achieve hot upgrades in software running. In Erlang versioning, you can keep two different versions of a module, supporting rollback of the version.

Certain aspects of the Erlang language focus on demand, is not completely suitable for all situations, while it has many attractive features also has some disadvantages: the language of the abstract ability is not strong, Erlang is a weakly typed language, can adjust the message more convenient when matching the requirements of the content or patterns, but when an error occurs, the error concealment is stronger.

2.1.2 Scala with Akka

Akka is a toolkit and runtime for building highly concurrent, distributed, and fault-tolerant applications on the JAVA virtual Machine (JVM) platform. Akka is written in Scala and provides Scala and JAVA development interfaces. Akka's approach to concurrency is based on the Actor model, where the only mechanism for communication between actors is messaging.

Akka features a higher abstraction of the concurrency model as an asynchronous, non-blocking, high-performance event-driven programming model. Event handling is lightweight (1GB of memory holds millions of actors). It provides a concurrency model called Actor, which is less granular than threads and can enable a large number of actors in the system. It provides a fault-tolerant mechanism that allows some recovery or reset in the event of an exception from the Actor. Akka can build highly concurrent programs on a single machine or distributed programs on a network and provide Actor location services with location transparency.

Threads are the basic unit of execution for concurrent programs, but in Akka the unit of execution is Actor. The traditional concurrent program is based on the object-oriented method, which transmits information through the method call of the object. If the method of the object modifies the state of the object itself, the inconsistency of the state of the object may occur under multithreading. At this time, the method call must be synchronized, and the synchronous operation will sacrifice performance. Instead of telling the Actor what to do through some method of an Actor object in the Actor model, you send a message to the Actor. When an Actor receives a message, it is possible for it to do something based on the content of the message, such as change its own state, which is then changed by the Actor itself, not by outside intervention.

2.2 Fault Tolerance

2.2.1 Erlang

An exit signal is generated when an Erlang process exits unexpectedly. All processes linked to the dying process receive this signal. By default, the recipient exits altogether and transmits signals to other processes to which it is linked until all processes directly or indirectly linked together exit. This cascading behavior allows a group of processes to exit as a single application, so you don't have to worry about whether there are any remaining processes that failed to shut down completely when the system restarts as a whole. Each process encapsulates its entire state so that the rest of the system is not damaged when the process exits. As with individual processes, this also applies to groups of processes that are linked to each other. One process crashes and the other processes that work with it exits, cleanly wiping out all the previously established complexities, saving the programmer time and errors.

One of the main ways to achieve fault tolerance in is to override the default propagation behavior of exit signals. By setting the trap_exit process flag, you can stop the process obeying the exit signal from outside, and instead capture it. In this case, when the process receives the signal, it will first convert it to a message in the format 'EXIT', Pid, Reason, which describes which process failed for what Reason, and then the message will be dropped into the mailbox just like a normal message. The process that captures the signal can check and process the message. These signal-catching processes are sometimes called system processes, and the code they execute is often different from normal worker processes (that is, processes that do not usually capture signals).As a bulwark against the further spread of exit signals, system processes block the connection between other processes linked to them and the outside world, so they can be

used to report failures and even restart subsystems. We call such processes supervisors.

The purpose of stopping and restarting the entire subsystem is to bring the system back to a known working state. It's a bit like rebooting your computer: by rebooting, you can quickly get it back to work. But the problem with rebooting an entire computer is that it's too granular. Ideally, you should be able to restart only a portion of the system, with as little granularity as possible. Together with supervisors, Erlang's process linking provides a fine-grained "restart" mechanism.

2.2.2 Scala with Akka

Akka's fault tolerance mechanism is based on a hierarchy: Akka adds a monitoring policy to the Actor to monitor its children. The following code adds a monitoring policy to the Actor, which monitors the content of the policy: if the child Actor throws an Exception while running, the child Actor is stopped (that is, stopped).

2.3 Server Test Tools

2.3.1 Chaos Monkey

Chaos Monkey is a tool invented by Netflix in 2011 to test the resilience of its IT infrastructure.[2] It works by deliberately disabling computers on Netflix's production network to test how the rest of the system responds to outages.Chaos Monkey is now part of a larger set of tools called the Ape Army, which is designed to simulate and test responses to various system failures and edge conditions.

References

@articlearmstrong1993concurrent, title=Concurrent programming in ERLANG, author=Armstrong, Joe and Virding, Robert and Wikström, Claes and Williams, Mike, year=1993, publisher=Citeseer