Java Actor Library

Description

The goal of this task is to implement a Java Actor System. An Actor is basically a software entity that receives and send messages using Message Queues, and a single thread that reads and process those messages sequentially. The sequential process of messages is extremely useful for concurrency reasons, avoiding race conditions in the access to the state of the actor.

To create an actor we will use an ActorContext entity based on Singleton pattern. The API of the Actor Context is: ActorContext.spawnActor("name",new ActorInstance()), ActorContext.lookup(name), and ActorContext.getNames().

Example:

```
ActorProxy hello = ActorContext.spawnActor("name",new RingActor());
hello.send(new Message(null,"Hello World");
```

ActorProxy only defines a method send(Message) that submits this message to the queue of the Actor.

ActorContext is a registry of Actors indexed using a unique id. Spawning an actor will create the required thread that will listen to messages for this actor.

An Actor has methods to send messages to its queue, and a process method that reacts to messages in that queue. A QuitMessage forces the Actor to stop running. A Message includes a from field which is an Actor reference (used to reply to that actor) and a text message (String).

To demonstrate the Actor system, create a HelloWorldActor that receives a message and writes it in System.out. Now use several Actors that send messages to this Actor, to demonstrate that it can process concurrent messages.

ActorProxy

In order to communicate with an Actor in a OOP way, we will extend ActorProxy to also receive messages from an Actor. The ActorProxy will have its own queue so that Actors can send back a response to the requester Proxy.

Create an InsultActor accepting three messages: GetInsultMessage, AddInsultMessage, GetAllInsultsMessage. The Actor has a list of insults, that can be extended using the AddInsultMessage. GetInsultMessage will return a random insult in a Message to the requesting Actor or Proxy entity.

Demonstrate that you can communicate with Actors using Proxies.

Example:

```
ActorProxy insult = ActorContext.spawnActor(“name”,new InsultActor());
insult.send(new GetInsultMessage());
Message result = insult.receive();
System.out.println(result.getText());
```

ActorDecorator

Using the Decorator pattern, implement an ActorDecorator enabling to modify message processing to an actor. Create a FirewallDecorator and an EncryptionDecorator to demonstrate the pattern.

The FirewallDecorator will only let process messages whose sender is a valid Actor registered in the ActorContext. It will stop all messages coming from a Proxy.

Create a LambdaFirewallDecorator that accepts closures to filter the messages than can be received using an AddClosureMessage.

The EncryptionDecorator will encrypt (send) and decrypt message text (process) between communicating Actors.

Demonstrate a pipeline of Decorators connecting EncryptionDecorator with Firewall Decorator.

Example:

```
ActorProxy hello = ActorContext.spawnActor(“name”,new FireWallDecorator(new RingActor()));
```

Reflection and Dynamic Proxy

Replace the ActorProxy using a DynamicProxy over an intercepted Java class. For example, an InsultService class with methods addInsult, getAllInsults, getInsult.

Example:

```
ActorProxy actor = ActorContext.spawnActor(“name”,new InsultActor());
InsultService insulter = DynamicProxy.intercept(new InsultService(), actor);
insulter.addInsult(“stupid”);
System.out.println(insulter.getInsult());
```

OPTIONAL: Create a ReflectiveActor that given a Java Class, automatically maps the processing of messages (process) to method invocation (i.e getInsultMessage to getInsult). This way, the developer does not need to implement the process method, just provide a valid instance of a Java class.

Example:

```
ActorProxy insult = ActorContext.spawnActor(“name”,new ReflectiveActor(new InsultService()));
```

MonitorService and Observer pattern

The idea is to create a MonitorService that can obtain runtime information about the ActorSystem. Using the Observer pattern, create an ActorListener enabling to receive information about four Actor Events (creation, finalization, incorrect finalization, and received message). Incorrect finalization means that the Actor stopped without receiving a QuitMessage (it halted abruptly due to an error).

The MonitorService can subscribe to some specific Actors (monitorActor(name)) or to all Actors in the system (monitorAllActors()). The service will offer information about Actor message traffic (LOW<5 messages, MEDIUM>5 <15, HIGH >15). A method (getTraffic) can be used to obtain a Map where the key is the traffic (LOW, MEDIUM, HIG) and the value is the list of actor names.
The Monitor Service also offers methods to getNumberofMessages for a givenActor, to log all messages from one or more Actors, and to log all events.

The MonitorService also offers a getSentMessages method working over the message log, which returns a Map where the key is the Actor, and the value is the messages sent by that actor. You can also implement the analogous getReceivedMessages returning a Map where the key is the Actor, and the value is the list of Messages sent by that Actor. You can use Java streams and Lambdas in this task.

The MonitorService also offers a getEvents method working over the event log, which returns a Map where the key is the enum (CREATED, STOPPED, ERROR), and the value contains the aforementioned events.

Validation

To demonstrate that the Actor system is running correctly we propose two simple examples: Ring and PingPong. Ring consists of creating a number of actors connected in a ring: every actor is connected to the next one in the ring. When we send one message to an actor, this will resend the message to the next actor in the ring. PingPong consists of two actors sending message to each other. Create a mechanism to avoid infinite message processing.

Calculate the performance in number of messages that your Actor system can process.
Using System.currentTimeMillis, you can also calculate how much time it took to process 100 entire rounds in a ring of 100 nodes.

Use JavaDocs and Unit tests to validate the system.

OPTIONAL: Create your github project including self-contained documentation.

RemoteActorContext and RemoteActorProxy (OPTIONAL)

Use java.util.RMI to enable the remote communication with Actors. A RemoteActorContext will create a RMI Server waiting for messages. A RemoteActorProxy will create a RMI client that connects to the remote entity (RemoteActorContext) to send or receive messages.

Virtual Actors and performance (OPTIONAL)

Using a Factory Design pattern, enable to use normal Java Threads or Virtual Threads (Java 19) in the Actor system. Compare the performance of both systems using the Ring example.

References

Actor Model
https://en.wikipedia.org/wiki/Actor_model

What is the Actor Model & When Should You Use it?
https://mattferderer.com/what-is-the-actor-model-and-when-should-you-use-it

The actor model in 10 minutes
https://www.brianstorti.com/the-actor-model/

Java Concurrency
https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Virtual Threads
https://blogs.oracle.com/javamagazine/post/java-loom-virtual-threads-platform-threads
https://levelup.gitconnected.com/java-virtual-threads-millions-of-threads-within-grasp-e0a4d26548ba

Java RMI
https://mkyong.com/java/java-rmi-hello-world-example/