UNIVERSIDAD CENTRAL DEL ECUADOR

FACULTAS DE CIENCIAS APLICADAS

SISTEMAS DE INFORMACIÓN

PROGRAMACIÓN DISTRIBUIDA

FAULT TOLERANCE

Nombre

Vanessa Morales

2022-2023

Universidad Central del Ecuador
Facultad de Ingeniería y Ciencias Aplicadas
Programación Distribuida

Nombre: Vanessa Lizbeth Morales Yugsán

Fault Tolerance

Cada vez es más importante crear microservicios tolerantes a fallas. La tolerancia a fallas se trata de aprovechar diferentes estrategias para guiar la ejecución y el resultado de alguna lógica. Las políticas de reintento determinan si deben llevarse a cabo ejecuciones y cuándo, y los fallbacks ofrecen un resultado alternativo cuando una ejecución no se completa con éxito.

La especificación de Fault tolerance debe centrarse en los siguientes aspectos:

Timeout: Define una duración para el tiempo de espera.

Retry: Define un criterio sobre cuándo reintentar.

Fallback: Proporciona una solución alternativa para una ejecución fallida.

CircuitBreaker: Ofrece una forma de falla rápida al fallar automáticamente la ejecución para evitar que el sistema sobrecarga y espera indefinida o tiempo de espera por parte de los clientes.

Bulkhead: Aisla las fallas en una parte del sistema mientras que el resto del sistema aún puede funcionar.

El diseño principal es separar la lógica de ejecución de la ejecución. la ejecución se puede configurar con políticas de tolerancia a fallas, como Retry, Fallback, Bulkhead y CircuitBreaker.

Relación con otras especificaciones

Esta especificación define un conjunto de anotaciones para ser utilizadas

ESTILO

por clases o métodos. Las anotaciones son enlaces de interceptor. Por lo tanto, esta especificación depende de las especificaciones Jakarta Interceptors, Contexts, Dependency Injection definidas en la plataforma Jakarta EE.

Relación con contextos e inyección de dependencia

La especificación Contexts y Dependency Injection (CDI) define un poderoso modelo de componentes para permitir el diseño de arquitectura débilmente acoplada. Esta especificación explora el SPI proporcionado por CDI para registrar un interceptor para que las políticas de Fault Tolerance se puedan aplicar a la invocación del método.

Relación con los interceptores de Jakarta

La especificación Jakarta Interceptors define un modelo de programación básico y la semántica de los interceptores. Esta especificación utiliza los enlaces de interceptor con seguridad de tipos. Las anotaciones @Asynchronous, @Bulkhead, @CircuitBreaker, @Fallback, @Retry, @Timeout son enlaces de interceptor.

Estas anotaciones pueden vincularse a nivel de clase o de método. Las anotaciones se adhieren a las reglas vinculantes del interceptor definidas por la especificación de Jakarta Interceptors.

Dado que esta especificación depende de las especificaciones de CDI e interceptores, las operaciones de tolerancia a fallas tienen las siguientes restricciones:

- Los enlaces de los interceptores de tolerancia a fallas se deben aplicar en una clase de bean o en un método de clase de bean, de lo contrario, se ignorará.

- La invocación debe ser una invocación de método comercial.
- Si un método y la clase que lo contiene no tienen ningún enlace de interceptor de tolerancia a fallas, no se considerará como una operación de tolerancia a fallas.

Relación con la configuración del microperfil

La especificación de configuración de MicroProfile define un modelo de configuración flexible para permitir la configuración de microservicios y lograr la separación estricta de la configuración del código. Todos los parámetros en las anotaciones/enlaces del interceptor son propiedades de configuración. Se pueden configurar externamente a través de otras fuentes de configuración predefinidas (por ejemplo, variables de entorno, propiedades del sistema u otras fuentes). Por ejemplo, el parámetro maxRetries en la anotación @Retry es una propiedad de configuración. Se puede configurar externamente.

Ejecución

Usar el interceptor y la anotación para especificar la ejecución y la configuración de la política. La anotación Asynchronous debe especificarse para cualquier llamada asíncrona. De lo contrario, se asume la ejecución síncrona.

Asynchronous

Significa que la ejecución de la solicitud del cliente se realizará en un subproceso separado.

Uso

Un método o una clase se pueden anotar con @Asynchronous, lo que significa que el método o los métodos de la clase serán invocados por un hilo separado. El contexto de RequestScoped debe estar

ESTILO

activo durante la invocación del método asíncrono. El método anotado con @Asynchronous debe devolver un Future o un CompletionStage del paquete java.util.concurrent. De lo contrario, se produce una FaultToleranceDefinitionException.

La ejecución de los interceptores restantes y el cuerpo del método se llevará a cabo en un subproceso separado.

- Hasta que finalice la ejecución, el Future o CompletionStage que se devolvió será incompleto.
- Si la ejecución arroja una excepción, Future o CompletionStage se completarán con esa excepción.
- Si la ejecución finaliza normalmente y devuelve un valor, Future o CompletionStage tendrán un comportamiento equivalente al valor devuelto (que, en sí mismo, es Future o CompletionStage).

Timeout

El timeout evita que la ejecución espere para siempre. Se recomienda que una invocación de microservicio tenga un tiempo de espera asociada

Uso

Un método o una clase se pueden anotar con @Timeout, lo que significa que el método o los métodos de la clase tendrán aplicada la política de tiempo de espera.

Política de Retry

Para recuperarse de una breve falla en la red, se puede usar @Retry para invocar la misma operación nuevamente. La política de reintentos permite configurar:

- maxRetries: El máximo de reintentos.
- delay: Retrasos entre cada reintento.

- delayUnit: La unidad de retardo.
- maxDuration: Duración máxima para realizar el reintento.
- durationUnit: Unidad de duración.
- jitter: La variación aleatoria de los retrasos de reintento.
- jitterDelayUnit: La unidad de fluctuación
- retryOn: Especifica los fallos para reintentar.
- abortOn: Especificar los fallos para abortar.

Uso

@Retry se puede aplicar al nivel de clase o método. Si se aplica a una clase, significa que se aplicará la política @Retry a todos los métodos de la clase. Si se aplica a un método, significa que ese método tendrá aplicada la política @Retry. Si la política @Retry se aplicó en un nivel de clase y en un nivel de método dentro de esa clase, el nivel de método @Retry anulará la política @Retry de nivel de clase para ese método en particular.

Cuando un método regresa y la política de reintento está presente, se aplican las siguientes reglas:

- Si el método devuelve normalmente (no arroja), simplemente se devuelve el resultado.
- De lo contrario, si el objeto (asignado) arrojado es asignable a cualquier valor en el parámetro abortOn, el objeto se vuelve a lanzar.
- De lo contrario, si el objeto arrojado se puede asignar a cualquier valor en el parámetro RetryOn, el método se vuelve a intentar.

La anotación @Retry se puede usar junto con @Fallback, @CircuitBreaker, @Asychronous, @Bulkhead y @Timeout. Se puede especificar un @Fallback y se invocará si el método aún falla después de que hayan ejecutado

ESTILO

las retiradas. Si @Retry se usa con @Asynchronous y se requiere un reintento, el nuevo reintento se puede ejecutar en el mismo subproceso que el intento anterior o en un subproceso diferente.

Fallback

Se invoca un método Fallback si un método anotado con @Fallback se completa excepcionalmente.

La anotación Fallback se puede usar sola o junto con otras anotaciones de Fault Tolerance.

El respaldo se invoca si se lanzaría una excepción después de que se hayan realizado todos los demás procesos de Fault Tolerance.

Para un Retry, el Fallback se maneja cada vez que el Retry excede su número máximo de intentos.

Para un CircuitBreaker, se invoca cada vez que falla la invocación del método. Cuando el circuito está abierto, siempre se invoca el Fallback.

Uso

Un método se puede anotar con @Fallback, lo que significa que el método tendrá aplicada la política de respaldo. Hay dos formas de especificar el respaldo.

- Especificar una clase FallbackHandler.
- Especificar el fallbackMethod.

Especificar una clase FallbackHandler

Si se registra un FallbackHandler para un método que devuelve un tipo diferente al que devolvería el FallbackHandler, el contenedor debe tratarse como un error y la implementación falla.

Los FallbackHandlers están destinados a ser administrados por CDI y deben seguir el ciclo de vida del alcance del bean.

Especificar el fallbackMethod

Esto se usa para especificar que se debe llamar a un método con nombre si se requiere un respaldo.

Cuando se usa fallbackMethod, se lanzará un FaultToleranceDefinitionException si no se cumple alguna de las siguientes restricciones:

- El método alternativo nombrado debe estar en la misma clase, una superclase o un método implementado.
- El método alternativo con nombre debe tener los mismos tipos de parámetros que el método anotado.
- El método alternativo con nombre debe tener el mismo tipo de retorno que el método anotado.
- El método alternativo con nombre debe ser accesible desde la clase que declara el método anotado.

Especifique los criterios para activar Fallback.

El respaldo puede activarse cuando se produce una excepción, incluidas las definidas en esta especificación (ej: BulkheadException, CircuitBreakerOpenException, TimeoutException, etc.). Cuando un método regresa y la política de respaldo está presente, se aplican las siguientes reglas:

- Si el método devuelve normalmente (no arroja una excepción), el resultado simplemente se devolverá.
- De lo contrario, si el objeto lanzado se puede asignar a cualquier valor en el parámetro skipOn, el objeto lanzado se volverá a lanzar.

ESTILO

- De lo contrario, si el objeto arrojado se puede asignar a cualquier valor en el parámetro applyOn, el valor especificado se activará la reserva.
- De lo contrario, el objeto lanzado se volverá a lanzar

CircuitBreaker

Un interruptor de circuito evita fallas repetidas, por lo que los servicios disfuncionales o las APIs fallan rápidamente. Si un servicio falla con frecuencia, el disyuntor se abre y no se intentan más llamadas a ese servicio hasta que haya transcurrido un periodo de tiempo.

Hay tres estados de circuito:

- Cerrado: En funcionamiento normal, el interruptor automático está cerrado. El disyuntor registra si cada llamada es un éxito o un fracaso y realiza un seguimiento de los resultados más recientes en una ventana móvil.

Una vez que la ventana móvil está llena, si la proporción de fallas en la ventana móvil aumenta por encima de la tasa de fallas, se abrirá el interruptor automático.

- Abierto: Cuando el disyuntor está abierto, las llamadas al servicio que opera debajo del disyuntor fallarán inmediatamente con una CircuitBreakerOpenException. Después de un retraso configurable el interruptor automático pasa al estado semiabierto.
- Semiabierto: En estado semiabierto, se permite un número configurable de ejecuciones de prueba del servicio. Si alguno de ellos falla, el disyuntor vuelve al estado abierto. Si todas las ejecuciones de prueba tienen éxito, el interruptor automático pasa al estado cerrado.

Uso

Un método o una clase se pueden anotar con @ CircuitBreaker, lo que significa que el método o los métodos de la clase tendrán aplicada la política CircuitBreaker.

Configurar cuando se abre y se cierra el circuito.

Los siguientes parámetros controlan cuándo se abre y se cierra el interruptor automático.

- requestVolumeThreshold: Controla el tamaño de la ventana movil utilizada cuando el interruptor automático está cerrado.
- failureRatio: Controla la proporción de fallas dentro de la ventana móvil que provocarán la apertura del interruptor automático.
- SuccessThreshold: Controla el número de llamadas de prueba que se permiten cuando el disyuntor está medio abierto.
- delay y delayUnit: Controlan cuánto tiempo permanece abierto el interruptor automático.

Configurar qué excepciones se consideran un error

Los parámetros failOn y skipOn se utilizan para definir qué excepciones se consideran fallas con el fin de decidir si se debe abrir el interruptor automático.

Cuando un método devuelve un resultado, se aplican las siguientes reglas para determinar si el resultado es un éxito o un fracaso:

- Si el método no lanza un Throwable, se considera un éxito.
- De lo contrario, si el objeto (lanzado) arrojado es asignable a cualquier valor en el parámetro skipOn, es considerado un éxito.
- De lo contrario, si el objeto arrojado es asignable a cualquier valor en el parámetro failOn, es considerado un fracaso.

• De lo contrario se considera un éxito.

Bulkhead

El patrón Bulkhead es para evitar que las fallas en una parte del sistema se transmitan en cascada a todo el sistema, lo que podría provocar la caída de todo el sistema. La implementación consiste en limitar el número de solicitudes simultáneas que acceden a una instancia. Por lo tanto, el patrón Bulkhead solo es efectivo cuando se aplica @Bulkhead a un componente al que se puede acceder desde múltiples contextos.

Uso

Un método o clase se puede anotar con @Bulkhead, lo que significa que el método o los métodos de la clase tendrán la política de Bulkhead aplicada correspondientemente. Hay dos enfoques diferentes para el Bulkhead: Aislamiento de grupos de subprocesos y aislamiento de semáforos. Cuando @Bulkhead se usa con @Asynchronous, se usará el enfoque de aislamiento de semáforo. El enfoque del grupo de subprocesos permite configurar el máximo de solicitudes simultáneas junto con el tamaño de la cola de espera. El enfoque de semáforo solo permite la configuración del número simultáneo de solicitudes

Interacciones con otras anotaciones

La anotación @Bulkhead se puede usar junto con @FallBack, @CircuitBreaker, @Asynchronous, @Timeout y @Retry.

Si se especifica un @Fallback, se invocará si se lanza la BulkheadException. Si @Retry se usa con @Bulkhead, cuando una invocación falla debido a una BulkheadException, se vuelve a intentar después de esperar el retraso configurado en @Retry.

Si el Bulkhead permite que una invocación se ejecute pero luego lanza otra excepción que es manejada por @Retry, primero abandona el mamparo, lo que reduce el recuento de solicitudes simultáneas en ejecución en 1, espera el retraso configurado en @Retry y luego intenta para entrar en el mamparo de nuevo. En este punto, puede aceptarse, ponerse en cola (si el método también está anotado con @Asynchronous) o fallar con una BulkheadException (lo que puede resultar en más reintentos).