Note sviluppo libreria realtime

Logger

Ogni processo eseguibile ed ogni risorsa ha a disposizione un proprio log (da valutare se usare uno StringWriter) sul quale in maniera esclusiva scrive messaggi in merito alla sua esecuzione

I messaggi sono del tipo: codice messaggio ; tempo in cui il codice messaggio si presenta

Finita l'esecuzione si possono reperire i log ed elaborarli (magari riunificandoli) per ottenere informazioni di più alto livello.

I codici correnti sono:

00: messaggio generico

10: creazione di un oggetto schedulabile (thread o handler)

20: inizio job

30: termine job

40: occupazione risorsa

50: rilascio risorsa

60: deadline non rispettata

70: job saltato per polica skip

80: job interrottto per politica stop

Si può pensare che ad ogni schedulabile ed ogni risorsa viene associata una vera e propria lasse Log che mostra funzionalità di più alto livello: ad esempio scriviInizioJob() ecc.Si può pensare ad inserire una gerarchia di Log: uno legato ad un processo periodico, uno legato ad un handler, uno legato ad una risorsa e uno legato ad un processo aperiodico.

La classe base contiene una hashtable che per ogni codice riporta una stringa che ne descrive il significato.

Dal lato interrogazione deve essere possibile, oltre che ottenere lo StringWriter sottostante anche ottenere una collezione di eventi del tipo logLine.

Questi oggetti, sono formati dal codice dell'evento, dalla sua descrizione sottoforma di stringa dal tempo assoluto (in forma di AbsoluteTime) in cui l'evento è avvenuto e dal nome del thread che ha scatentato l'evento. Il toString di uno di questi oggetti è dato dalla concatenazione di questi tre campi. L'unica eccezione è data dal messaggio generico che riporta il testo indicato dall'utente al posto del campo realtivo al nome del thread.

Il toString del log è la stampa del contenuto dello String Writer

In seguito valuta se usare uno strinwriter solito o sfrtuttare uno delle zone peciali di memoria messe a disposizione dalla libreira di java realtime.

Devi fornire una classe util, in costante sviluppo che offre tutta una serie di metodi. Il primo deve permettere di riunire vari log restituendo una stampa di tutti gli eventi in ordine cronologico

Una seconda funzionalità deve fare lo stesso, ma i tempi devono essere relativi ad un tempo zero specificato.

Busy Wait

La classe principale, busy wait deve essere un singleton. Per poter essere utilizzata deve essere inizializzata. L'inizializzazione avviene tramite il metodo initialize().

Ha un attributo privato, float, che indica il numro di iterazioni compiute mediamente in un millisecondo

Ha come attributo due long, uno che indica di quanti millisecondi deve essere lunga la fase di inizializzazione di default, se non settato diversamente, è pari a 1000 (1 secondo). E' consigliabile, per non perdere precisione che tale valore non scenda sotto il decimo di secondo.

Un altro attributo float indica quanto deve essere lunga la fase di calibrazione di default.Se non settato diversamente è pari a due secondi, per non perdere è consigliabile sia comunque almeno di mezzo secondo

Esibisce il metodo doJobFor(long millis) che non fa altro che eseguire lo stesso ciclo usato per la inizializzazione un numero di volte pari al prodotto del numero medio di cicli in un milliscondo moltiplicato per il numero di millisecondi richiesto. Lancia un'eccezione se la classe non è inizializzata.

Esibisce anche il metodo doInterumpibleJobFor(log millis). Questo metodo è identico a doJobFor, con la differenza che, quando viene eseguito può essere interrotto asincronicamente dal momento che può lanciare una AsynchronouslyInterruptedException. Ovviamente non è possibile richiamare al suo interno il metodo doJobfor dal momento che se il metodo interrupt viene chiamato si sta eseguendodel codice ATC-deferred (ossia un metodo syncronized o che non dichiara di poter lanciare una AsynchronouslyInterruptedException) l'eccezione viene lanciata solo quando si esce dal metodo e si torna in un metodo interrompibile.

Il metodo Initialize crea un'istanza della classe InitializerMasterThread e la avvia. La classe initializerMasterThread è un thread realtime che esegue con priorità massima.

Ha, come attributo due long: il primo indica di di quanti millisecondi deve essere lunga la fase di inizializzazione, il secondo di quanto deve essere la fase di calibrazione.

Un'altro attributo è un long che indica quanti cicli sono stati eseguiti nel tempo di inizializzazione.

Il metodo run di InitializerMasterThread non fa altro che chiamare il metodo run della classe InitializerServerThread, sospendersi per il tmpo della fase di calibrazione e, al risveglio, interrompere InitializerServerThread e controllare quanti cicli di esecuzione ha svolto.

La classe InitializerServerThread ha come proprietà il numero di cicli eseguiti finora

Il suo metodo run ha un ciclo in cui si incrementa e si decrementa una variabile intera e si incrementa il numero di iterazioni.

Facendo le prove si è visto che i tempi di esecuzione con la busy wait semplice sono in realtà circa la metà di quelli che ci si aspetta. E' quindi necessario fare una calibrazione della variabile cicli per millisecondo: si fa una busywait di durata pari al parametro di calibrationTime e se verifica quanto questa è durata in realtà. Quindi si ottiene il vero numero di iterazioni del ciclo di busy wait dividendo il vecchio numero di cicli al millesimo per la durata effettiva e moltiolicando per quella nominale. In formula: **n iterazioni al millesimo = n° iterazioni al millesimo * durata nominale ciclo di busyWait / durata effettiva ciclo di busyWait.**

Possibile miglioramento: introdurre nel ciclio perditempo delle operazioni più pesanti, che rendano trascurabile l'aumento del contatore delle iterazioni, in questo modo il metodo workFor non avrebbe bisogno di tenere traccia del numero di esecuzioni in una variabile interna al metodo, con possibili asimmetrie rispetto all'aggiornamento del campo di una classe.

Gestione politica in caso di DeadlineMiss

Quando un thread sfora la deadline si può introdurre della logica di controllo in due momenti distinti: nel momento stesso in cui il thread sfora la deadline e nelle release successive del thread. Il primo compito è svolto dall'handler di deadlineMiss associato al thread che viene appositamente risvegliato dal sistema qualora il thread sfora la deadline. Si è deciso che la seconda fase sia responsabilità di un oggetto che implementa un'interfaccia (IpendingJobManager), che contiene la logica da eseguire in caso di job di recupero.Questo oggetto può anche essere l'handler stesso che, in questo modo, contiene tutta la politica di gestione nel caso di missedDeadline.

Il thread deve quindi porter distinguere tra quelli che sono i job normali e quelli di recupero diversificando la priopria esecuzione nei due casi differenti.A tale scopo si è inserito il flag booleano pendingMode. Se false, significa che la release corrente è una release ordinaria, in cui si deve eseguire la logica oridinaria, se true deve eseguire il metodo specifico del manager di deadlineMiss. Tale flag viene inizializzato a false in modo che il thread, in assenza di interventi dall'esterno compia sempre la sua logica ordinaria.

Periodic Thread

Classe che modella un generico thread periodico. Le sue proprietà sono:

- il log sul quale scrivere gli eventi che la riguardano
- il tempo di esecuzione di ciascun job
- il numero di iterazioni che deve fare (il numero di job)
- il numero dell'iterazione (del job) corrente
- il flag che indica se eseguire un job normale o di recupero (inizializzato a false per indicare che, se non modificato da fonti esterne si esegue sempre un job normale)
- il riferimento al gestore di release di recupero

La classe estende javax.realtime.RealTimeThread. La sua parte più significativa è il metodo run.

Per ogni ciclo di esecuzione (job) se il valore del flag della release di recupero è pari a false (quindi si tratta di un'esecuzione “normale”)viene scritto l'inizio e la fine del job sul log e viene eseguito il metodo doJob deputato a contenere contiene la businness logic dell'esecuzione del thread. Questo

metodo non fa altro che eseguire una busyWait di durata pari al tempo di esecuzione indicato nella relativa proprietà.

Se, al contrario il flag è false, si richiama il metodo opportuno del gestore (pendingReleaseJob), se questo è presente. Altrimenti si esegue il metodo doJob.

Bad Thread

Classe che estende PeriodicThread e che modella il thread che si comporta in modo anomalo. In particolare il metodo doJob (che contiene la businness logic dell'esecuzione di ciascun job) è studiato in modo che un particolare job abbia un tempo di esecuzione diverso da tutti gli altri.

Le proprietà distintive di questa classe sono quindi l'iterazione in cui eseguire una esecuzione anomala e la durata di tale esecuzione anomala.

Il metodo doJob controlla quindi quale iterazione si sta eseguendo: se è quella in cui eseguire l'esecuzione anomala fa una BusyWait pari al valore indicato; in caso contrario il tempo della busyWait è quello dell'esecuzione “normale”.

InterrumpiblePeriodicThread

Classe che estendePeriodicThread e che modella un thread periodico la cui esecuzione può essere interrotta. Ciò è fatto eseguendo, nel metodo do job, una busy wait interrompibile al posto di una busy wait “normale”. L'azione da esegure nel caso di interruzione, inserita nella catch della AsynchronouslyInterruptedException, si limita a fare la clear dell'eccezione (necessaria per evitare che questa venga ulteriormente propagata) e a scrivere sul log che il job è stato interrotto.

IpendingReleaseManager

Le classi che implementano questa interfaccia devono implementare i metodi che contengono la logica da eseguire in caso di job “di recupero”. L'interfaccia è fatta da un metodo solo doPendingJob. Attualmente sono gli stessi handler che implementano questa interfaccia.

DeadlineMissedHandler

La classe base che rappresenta l'handler per la gestione del deadline missed. Questa classe estende direttamente la classe AsyncEventHandler, classe di sistema deputata a gestire gli eventi asincroni, quali gli sforamenti di deadline da parte dei thread real-time.Le sue proprietà sono il nome, il log ed il riferimento al thread che deve gestire. Quando si richiama il metodo setThread, questa provvede ad indicare al thread stesso che l'handler è anche il gestore per i job di recupero.

Sonostati inseriti due metodi, GetPriority e setPriority, per rendere più comodo la gestione della priorità dell'handler. Questi due metodi lavorano con la priorità collegata ai priorityParamenters associati all'handler

ASAPPolicyHandler

Handler che permette di implementare la politica di default del sistema, ossia AsSoonAsPossible.

Quando si verifica un evento di deadlineMissed il sistema provvede a richiamare il metodo handleAsyncEvent. Questo medoto procede a registrare l'evento nel log ed a rischedulare il Thread. Il metodo doPendingJob non viene mai utilizzato in quanto si eseguono sempre job normali e non si

entra mai in pending mode.

SkipPolicyHandler

Questo gestore di deadlineMissed estende la classe deadlineMissedHandler illustrata in precedenza.

Quando viene richiamato il metodo HandleAsyncEvent a fronte di uno sforamento di deadline, oltre a scrivere l'evento sul log ed al rischedulare il thread, l'handler imposta il flag pendigMode del thread a true e incrementa il suo contatore skipCount, che tiene traccia di quanti job in futuro si dovranno saltare.

Dualmente, il metodo doPendingJob, oltre a scrivere sul log che si è saltato il job, decrementa il contatore dei job da saltare e, se questo è arrivato a zero imposta il flag pendingMode del thread controllato a false.

StopPolicyHandler

La classe stopPolicyHandler modella la politica di gestione dei deadline miss secondo la quale, quando un job viola la sua deadline, questo debba essere interrotto. Perchè ciò sia possibile occorre che il thread controllato possa ricevere un'interruzione dall'handler quando quest'ultimo viene invocato dal sistema in occasione del deadline miss. Occorre, quindi che il thread controllato da questo tipo di handler sia un InterrumpiblePeriodicThread. A tale scopo si sono modificati sia il costruttore, sia la prioprietà controlledThread al fine di supportare solo un InterrumpiblePeriodicThread e non il generico PeriodicThread.Il metodo handleAsyncEvent, oltre a registrare sul log la violazione della deadline, intrrompe il lavoro del thread invocando metodo interrupt del thread controllato, che sta eseguendo una busyWait Interrompibile. Questo causa l'interruzione del job. Occorre, infine rischedulare il thread affinchè il nuovo job possa partire.

Possibili miglioramenti

- Si potrebbe inserire un altro momento in cui controllare l'esecuzione del thread: appena prima di terminare il job. Attualemente non ce ne è bisogno, ma in teoria si potrebbe utilizzare un flag apposito, o lo stesso pendingMode, per decidere se eseguire o meno un metodo tipo doPendingJob deputato a contenere la logica da eseguire al termine del job che ha sforato la deadline.
- Attualmente SKIPPolicyHandler non sfrutta mai il pending mode e di fatto non utilizza il metodo doPending job, si potrebbe fare si che lo sfrutti, usando un contatore delle release come fa SKIPPolicyHandler.L'esecuzione del job pendente non fa altro che richiamare il metodo dojob del thread.
- Potrei prevedere di iserire nel modulo di logging dei messaggi per diversificare l'inizio e la fine di un job normale rispetto all'inizio ed alla fine di un job di recupero.