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

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 workFor(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.

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.

Politica Skip

Aggiungo alla classe del thread periodico un parametro intero **skipNumber** che indica quante esecuzioni bisogna saltare. Un apposito deadlineMissedHandler (SkipPolicyHandler) nel suo handle event andrà ad incrementare questo parametro. Il thread, una volta iniziato il ciclo di esecuzione per prima cosa controlla se il parametro skipNumber è >0, in tal caso al posto di fare l'esecuzione normale, non fa altro che calare tale valore di uno, con l'effetto di saltare un job che, con la politica di defalut, verrebbe accodato immediatamente dopo quello precedente.

Aggiungo anche un altro tipo di evento registrabile nel log: quello di job saltato a causa della politica SKIP

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 di iterazioni da saltare, necessario per implementare la politica skip
- il numero dell'iterazione (del job) corrente

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

Per ogni ciclo di esecuzione (job) se il valore di skipNumber è pari a zero (quindi non ci sono release pendenti da trattare con politica skip 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 parametro skipNumber è maggiore di zero, deve essere eseguita una skip:viene scritto sul log che ho fatto la skip di un job e viene decrementato il valore di skipNumber che, si ricorda, indica il numero di esecuzioni da saltare in accordo alla politica skip.

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”.

DeadlineMissedHandler

La classe rappresenta l'handler base 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 verifica un evento di deadlineMissed il sistma provvede a richiamare il metodo handleAsyncEvent. Questo medoto procede a rigistrare l'evento nel log ed a rischedulare il Thread. In sostanza la politica implementata tramite questo handler è una politica di default del sistema, ossia AsSoonAsPossible.

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

SkipPolicyHandler

Questo gestore di deadlineMissed estende la classe deadlineMissedHandler illustrata in precedenza. Quando questo handler è associato ad un PeriodicThread viene realizzata la politica SKIP. Infatti, il metodo HandlerAsyncEvent, oltre ad annotare lo sforamento di deadline nell'handler ed a rischedulare il thread provvede ad aumentare lo skipNumber di quest'ultimo, in modo che questo salti tanti job quanti il valore di skipNumber.

Possibili miglioramenti

Attualmente la politica di gestione in caso di deadlineMissed è condivisa dall'handler e dal thread. Un'idea per migliorare la libreria in quest'area consiste nel separare la gestione della politica scelta in una singola entità. Si potrebbe associare al thread un oggetto deputato ad eseguire azioni sulle sole release “pendenti”, affidando al polimorfismo i diversi comportamenti. In sostanza, quando si verifica un deadlineMissed, l'handler deve incrementare un contatore. Tornato il controllo al thread, questo, quando esegue il job successivo si rende conto, in modo analogo a quanto fatto per la skip, di stare eseguendo un job “di recupero”. Di conseguenza, anziché eseguire la businness logic contenuta nel metodo doJob, esegue il metodo del getore. Questa stategia, tuttavia, non consente di implementare poliche che sospendano o uccidano il thread, cosa che può fare solo l'handler. In linea teorica le due entità potrebbero coincidere e l'handler venire chiamato anche dopo l'evento di deadlineMissed, ma non lo sto sovraccaricando di responsabilita?