Worker Thread

Also known as Background Thread, Thread Pool

Pattern Properties

Type: Processing

Level: Architectural

Purpose

To improve throughput and minimize average latency.

Introduction

When you introduce threading to an application, your main goal is to use threads to eliminate bottlenecks. However, it requires skill to implement correctly. One way to maximize efficiency in a multithreaded application is to take advantage of the fact that not all threaded tasks have the same priority. For some of the tasks that need to be performed, timing is crucial. For others, they just need to be executed; exactly when isn't important.

To save yourself some long nights, you can separate these tasks from the rest of your application and use the Worker Thread pattern. The worker thread picks up a task from a queue and executes it; when it's finished, it just picks up the next task from the queue.

Threading is easier with Worker Thread because when you want something done, but not specifically now, you put it in the queue. And your code will become easier to read because all the thread object issues are in the worker thread and the queue.

Applicability

Use Worker Thread when:

- You want to improve throughput
- You want to introduce concurrency

Description

One approach to implementing threads is as follows: when you start a new task, create a new `Thread` object and start it. The thread performs its designated task, then dies. That's simple enough. However, creating the thread instance is very expensive in terms of performance, it takes a lot of time, and you only get one task out of it. A more efficient approach is to create a longer-lived "worker thread" that performs many tasks for you, one after the other.

That's the essence of the Worker Thread pattern. The worker thread executes many unrelated tasks, one after the other. Instead of creating a new thread whenever you have a new task, you give the task to the existing worker thread, which handles the task for you.

The Worker Thread might still be handling the first task when you're ready to hand it the next task. Solutions include the following:

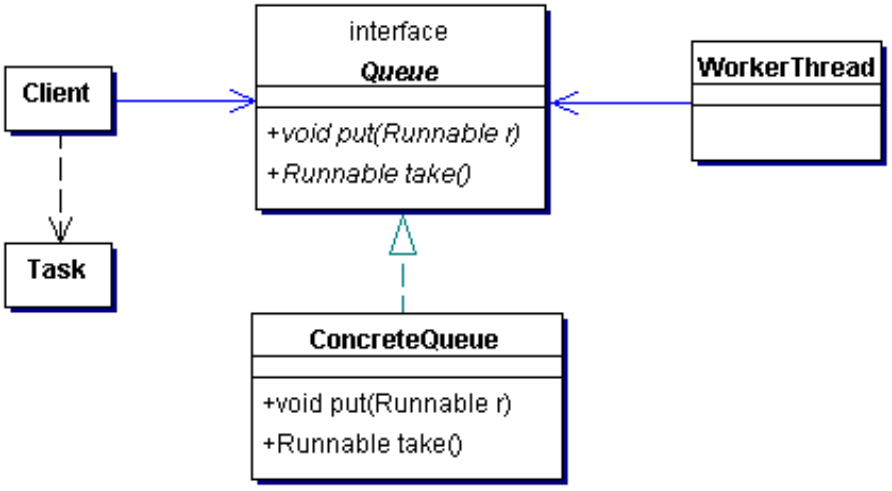
- Your application waits until the Worker Thread becomes available again, but that kills a lot of the benefit you gain from multithreading.
- Your application creates a new instance of the worker thread each time the other worker thread is unavailable, but then you're back at square one—creating a new thread each time you have a new task.

The solution to this problem of the temporarily unavailable thread is to store the tasks until the worker thread is available again. The new task is stored in a queue and when the worker thread has finished with a task, it checks the queue and takes the next task. The task doesn't get performed any sooner, but at least your application isn't standing around waiting to hand off the task. If there are no tasks, it waits for the next task to arrive. Putting a task on the queue is less expensive than creating a new thread.

Implementation

A Worker Thread class diagram is shown in Figure 4.5

Figure 4.5. Worker Thread class diagram



For the Worker Thread pattern, implement the following:

- `Client` – The client is responsible for creating the `Task` instances and putting the `Tasks` on the `Queue`.
- `Task` – The `Task` is the class that contains the work that needs to be executed. It implements the `java.lang.Runnable` interface.
- `Queue` – The `Queue` interface defines the methods by which the `Client` is able to hand off the `Tasks` and the `WorkerThread` to retrieve the `Tasks`.
- `ConcreteQueue` – The `ConcreteQueue` class implements the `Queue` interface and is responsible for storing and retrieving the `Tasks`. It determines the order in which `Tasks` are provided to the `WorkerThread`.

- `WorkerThread` – The `WorkerThread` takes `Tasks` from the `Queue` and executes them. If there are no `Tasks` on the `Queue` it waits. Because the `Queue` and the `WorkerThread` are tightly coupled, often the `WorkerThread` class is an inner class of the `ConcreteQueue` class.

Benefits and Drawbacks

The `WorkerThread` influences performance in several ways.

1. The client no longer needs to create thread objects in order to run several tasks. It only needs to put the task on the queue, which in performance is less expensive than creating a thread object.
2. A Thread that is not running is taking up performance because the scheduler still schedules the thread to be run, if the thread is in a runnable state. Creating and starting a thread per task means that the scheduler has to schedule each of these threads individually, which takes more time than when the scheduler has to schedule only one worker thread. More threads means more scheduling. A task that is sitting in the queue and isn't running takes up no time whatsoever.
3. The drawback of this design can occur when tasks are dependent on each other. Because the queue can be sequential, the system can get into a deadlock. That's disastrous from a threading and from a performance point of view.
   There are a couple of possible solutions for this dilemma:

   - Make sure that there are as many worker threads as there are tasks that need to be run concurrently. That means that you need to implement an expandable thread pool. The thread pool is discussed in the " Pattern Variants " section.
   - Only allow tasks on the queue that do not depend on other tasks. Sometimes such behavior cannot be guaranteed. In that case, the client cannot put the task on the queue, but has to instantiate its own thread or start another queue with worker threads.
   - Create a smart queue that understands how the tasks work together and knows when to give which task to a worker thread. This should be considered a last resort as this smart queue will be tightly bound to the application and may become a maintenance nightmare.

Pattern Variants

Thread pool is a variant in which there is not just one instance of the worker thread, but several instances in a pool. (Hence the name thread pool.) This pool manages the `WorkerThread` class instances. The pool creates the worker threads, determines when they are not needed for a while, and might ultimately destroy some worker thread instances.

The pool decides how many workers to create at startup and what the maximum will be. The pool can either choose to create some threads when it starts, so that it always has some threads available, or it can wait until the first request is made (lazy instantiation).

When there are too many tasks for the current number of threads, however, the system (like a drain) gets clogged. Several solutions exist:

- Increase the number of workers – This works for a limited time only; as this fixes the symptom, not the problem. Generally, you should choose a better solution.

- Don’t limit the number of tasks in the queue – Just let the queue grow until the system runs out of memory. This solution is better than increasing the number of workers, but still will fail due to a shortage of resources.
- Limit the size of the queue – When the backlog gets too big, clients no longer make calls to add tasks to the queue. The queue can then focus on processing the backlog of tasks.
- Ask clients to stop sending tasks – Clients can then choose to send either no requests, or fewer requests.
- Drop requests that are stored in the queue – If the pool can be certain that the client will retry, it's safe to drop new requests. Dropping old requests is the right choice when it's likely that the clients posting the request have gone away.
- Let the client run the task itself – The client becomes single-threaded while running the task, and can’t create new tasks until the first task is completed.

Related Patterns

None.

Example

Note:

For a full working example of this code example, with additional supporting classes and/or a `RunPattern` class, see “ Worker Thread ” on page 517 of the “ Full Code Examples ” appendix.

In a typical application, certain jobs have to be done. It's not always important that they happen now, just that they do happen. You can compare this to cleaning a house. It's not important that it happen at a particular time, as long as somebody does it sometime this week—or month, or year, depending on your standards.

This example uses a `Queue` to hold tasks. The `Queue` interface defines two basic methods, `put` and `take`. These methods are used to add and remove tasks, represented by the `RunnableTask` interface, on the `Queue`.

Example 4.10. `Queue.java`

```
1.    public interface Queue{
2.        void put(RunnableTask r);
3.        RunnableTask take();
4.    }
```

Example 4.11. `RunnableTask.java`

```
1.    public interface RunnableTask{
2.        public void execute();
3.    }
```

The `ConcreteQueue` class implements the `Queue` and provides a worker thread to operate on the `RunnableTask` objects. The inner class defined for `ConcreteQueue`, `Worker`, has a `run` method that continually searches the queue for new tasks to perform. When a task becomes available, the worker thread pops the `RunnableTask` off the queue and runs its `execute` method.

Example 4.12. `ConcreteQueue.java`

```
1.    import java.util.Vector;
2.    public class ConcreteQueue implements Queue{
3.        private Vector tasks = new Vector();
4.        private boolean waiting;
5.        private boolean shutdown;
6.
7.        public void setShutdown(boolean isShutdown){ shutdown = isShutdown; }
8.
9.        public ConcreteQueue(){
10.           tasks = new Vector();
11.           waiting = false;
12.           new Thread(new Worker()).start();
13.       }
14.
15.       public void put(RunnableTask r){
16.           tasks.add(r);
17.           if (waiting){
18.               synchronized (this){
19.                   notifyAll();
20.               }
21.           }
22.       }
23.
24.       public RunnableTask take(){
25.           if (tasks.isEmpty()){
26.               synchronized (this){
27.                   waiting = true;
28.                   try{
29.                       wait();
30.                   } catch (InterruptedException ie){
31.                       waiting = false;
32.                   }
33.               }
34.           }
35.           return (RunnableTask)tasks.remove(0);
36.       }
37.
38.       private class Worker implements Runnable{
39.           public void run(){
40.               while (!shutdown){
41.                   RunnableTask r = take();
42.                   r.execute();
43.               }
44.           }
45.       }
46.   }
```

Two classes, `AddressRetriever` and `ContactRetriever`, implement the `RunnableTask` interface in this example. The classes are very similar; both use RMI to request that a business object be retrieved from a server. As their names suggest, each class retrieves a specific kind of business object, making `Address` and `Contact` objects from the server available to clients.

Example 4.13. `AddressRetriever.java`

```
1.    import java.rmi.Naming;
2.    import java.rmi.RemoteException;
3.    public class AddressRetriever implements RunnableTask{
4.        private Address address;
5.        private long addressID;
6.        private String url;
7.
8.        public AddressRetriever(long newAddressID, String newUrl){
9.            addressID = newAddressID;
10.           url = newUrl;
11.       }
12.
13.       public void execute(){
14.           try{
15.               ServerDataStore dataStore = (ServerDataStore)Naming.lookup(url);
16.               address = dataStore.retrieveAddress(addressID);
17.           }

18.           catch (Exception exc){
19.           }
20.       }
21.
22.       public Address getAddress(){ return address; }
23.       public boolean isAddressAvailable(){ return (address == null) ? false : true; }
24.   }
```

Example 4.14. `ContractRetriever.java`

```
1.    import java.rmi.Naming;
2.    import java.rmi.RemoteException;
3.    public class ContactRetriever implements RunnableTask{
4.        private Contact contact;
5.        private long contactID;
6.        private String url;
7.
8.        public ContactRetriever(long newContactID, String newUrl){
9.            contactID = newContactID;
10.           url = newUrl;
11.       }
12.
13.       public void execute(){
14.           try{
```

```
15.              ServerDataStore dataStore = (ServerDataStore)Naming.lookup(url);
16.              contact = dataStore.retrieveContact(contactID);
17.          }
18.          catch (Exception exc){
19.          }
20.       }
21.
22.       public Contact getContact(){ return contact; }
23.      public boolean isContactAvailable(){ return (contact == null) ? false : true; }
24.   }
```