Jactor2 Revisited by Example

by Bill la Forge, 2014

JActor2 is a robust and high-performance alternative to threads and locks. Jactor2 Revisited focuses on a subset of the API that is easy to learn but reasonably comprehensive.

The HelloWorld Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class HelloWorld extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new HelloWorld();
        System.out.println("initialized");
    }

    public HelloWorld() throws Exception {
        new AIO("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                        final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                    throws Exception {
                System.out.println("Hello world!");
                Plant.close();
                System.out.println("finished");
            }
        }.signal();
    }
}
```

Output:

```
initialized
Hello world!
finished
```

The *HelloWorld* class is a *Blade*. It has a *Reactor* that is created when the default constructor of *IsolationBladeBase* is called.

```
    public static void main(final String[] args) throws Exception {
        new Plant();
        new HelloWorld();
        System.out.println("initialized");
    }
```

The *main* method does three things:

1. An instance of *Plant* is created. This provides the operating environment and configuration for the reactors.
2. An instance of *HelloWorld* is created. And
3. The line *initialized* is printed, as this completes the program initialization.

```
    public HelloWorld() throws Exception {
        new AIO("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                        final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                    throws Exception {
                System.out.println("Hello world!");
                Plant.close();
                System.out.println("finished");
            }
```

```
        }.signal();
    }
```

The constructor creates a *run* signal which is passed to the *HelloWorld Blade* via its *Reactor*. On receipt of this signal, the *Blade* prints the line *Hello world!*, closes the operating environment and then prints the line *finished*.

Notes:

1. The *AIO.signal* method can be called from any thread and within any context. In this case the method was called from the main thread.
2. *AIO* is a nested class, defined in one of the super classes of *HelloWorld*. This is how the *signal* method accesses the *Reactor* of *HelloWorld*.

The Worker Blade

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Worker extends IsolationBladeBase {
    public final String id;
    private int count;

    public Worker(final int _id) throws Exception {
        id = "Worker" + _id;
    }

    public int getCount() {
        return count;
    }

    public AO<Void> run(final long _iterations, final int _timeoutMillis) {
        return new AO<Void>("run" + id) {

            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                        final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                    throws Exception {
                _asyncRequestImpl.setMessageTimeoutMillis(_timeoutMillis);
                System.out.println(id + ": started  " + count++);
                for (long i = 0L; i < _iterations; i++);
                System.out.println(id + ": finished " + count);
                _asyncResponseProcessor.processAsyncResponse(null);
            }
        };
    }
}
```

The *Worker* blade is useful for simulating a CPU load and we will use it in a number of examples. It has one operation, *run*, which returns an Asynchronous Operation, *AO*, that can be used to pass the *run* request to *Worker*.

Like *AIO*, *AO* is defined as a nested *class* in a super class of *Worker*, which again is how it can access the *Reactor* of *Worker*. But unlike *AIO*, *AO* can not be used to send a signal. (This is the only difference—*AO* is the super class of *AIO*.)

Note that *count*, which is the number of times a run request has been received, is *private* and is only updated when processing a request, *run*. This means that there will be no race conditions for *count*, as requests are processed strictly one at a time.

We have not yet covered the *AsyncRequestImpl.setMessageTimeoutMillis* method, but we will do that when covering some examples that use *Worker*.

The Simple Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Simple extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new Simple();
        System.out.println("initialized");
    }

    public Simple() throws Exception {
        new AIO("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                        final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                    throws Exception {
                AsyncResponseProcessor<Void> runResponseProcessor =
                        new AsyncResponseProcessor<Void>() {
                    @Override
                    public void processAsyncResponse(Void _response) throws Exception {
                        Plant.close();
                        System.out.println("finished");
                    }
                };
                _asyncRequestImpl.send(new Worker(0).run(100000000L, -1), runResponseProcessor);
            }
        }.signal();
    }
}
```

Output:

```
initialized
Worker0: started  0
Worker0: finished 1
finished
```

The *Simple Blade* sends a *run* request to a *Worker* and then processes the response message. But note that it is while processing the *run* signal sent to *Simple* that the *AsyncRequestImpl.send* method is called.The *send* method can not be called except while processing a message. The *send* method takes two arguments: the request to be sent and an *AsyncResponseProcessor* object used to process the response message.

The Timeout Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class Timeout extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new Timeout();
        System.out.println("initialized");
    }

    public Timeout() throws Exception {
        new AIO("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                        final AsyncResponseProcessor<Void> _asyncResponseProcessor)
```

```
                    throws Exception {
                AsyncResponseProcessor<Void> runResponseProcessor =
                        new AsyncResponseProcessor<Void>() {
                    @Override
                    public void processAsyncResponse(Void _response) throws Exception {
                        Plant.close();
                        System.out.println("finished");
                    }
                };
                _asyncRequestImpl.send(new Worker(0).run(10000000000L, -1), runResponseProcessor);
            }
        }.signal();
    }
}
```

Output:

```
initialized
Worker0: started  0
[pool-1-thread-2] ERROR org.agilewiki.jactor2.core.reactors.Reactor - message timeout -> reactor close
[Thread-1] WARN org.agilewiki.jactor2.core.reactors.Reactor - Uncaught throwable
[pool-1-thread-2] ERROR org.agilewiki.jactor2.core.reactors.Reactor - hung thread
message=runWorker0, isComplete=true, isOneWay=false, source=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@2, this=class
org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#41de9139
message=run, isComplete=true, isOneWay=true, source=null, target=org.agilewiki.jactor2.core.impl.mtReactors.IsolationReactorMtImpl@1,
this=class org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl#6bb97f10
[pool-1-thread-2] ERROR org.agilewiki.jactor2.core.reactors.Reactor - hung thread -> plant exit
org.agilewiki.jactor2.core.reactors.ReactorClosedException
         at org.agilewiki.jactor2.core.impl.mtRequests.RequestMtImpl.close(RequestMtImpl.java:366)
         at org.agilewiki.jactor2.core.impl.mtRequests.AsyncRequestMtImpl.close(AsyncRequestMtImpl.java:331)
         at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.fail(ReactorMtImpl.java:298)
         at org.agilewiki.jactor2.core.impl.mtPlant.Recovery.onMessageTimeout(Recovery.java:46)
         at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.reactorPoll(ReactorMtImpl.java:624)
         at org.agilewiki.jactor2.core.impl.mtReactors.ReactorMtImpl.reactorPoll(ReactorMtImpl.java:634)
         at org.agilewiki.jactor2.core.impl.mtPlant.PlantMtImpl$1.run(PlantMtImpl.java:283)
         at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
         at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:308)
         at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$301(ScheduledThreadPoolExecutor.java:180)
         at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:294)
         at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
         at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
         at java.lang.Thread.run(Thread.java:745)
```

In the *Timeout* example, the number of iterations that *Worker* is told to perform has been increased to 10 billion. The result is that the message times out and the *Worker Reactor* is closed because the thread is hung. This unanticipated exception causes the *Timeout Reactor* to *close* as well and the program exits.

Most messages do not take long to process, so the default timeout is only a few seconds.

The VerySlow Example

```
package org.agilewiki.jactor2.core.revisited;

import org.agilewiki.jactor2.core.blades.IsolationBladeBase;
import org.agilewiki.jactor2.core.impl.Plant;
import org.agilewiki.jactor2.core.requests.AsyncResponseProcessor;
import org.agilewiki.jactor2.core.requests.impl.AsyncRequestImpl;

public class VerySlow extends IsolationBladeBase {

    public static void main(final String[] args) throws Exception {
        new Plant();
        new VerySlow();
        System.out.println("initialized");
    }

    public VerySlow() throws Exception {
        new AIO("run") {
            @Override
            protected void processAsyncOperation(final AsyncRequestImpl _asyncRequestImpl,
                    final AsyncResponseProcessor<Void> _asyncResponseProcessor)
                    throws Exception {
                AsyncResponseProcessor<Void> runResponseProcessor =
                        new AsyncResponseProcessor<Void>() {
                    @Override
                    public void processAsyncResponse(Void _response) throws Exception {
                        Plant.close();
                        System.out.println("finished");
```

```
                    }
                };
                _asyncRequestImpl.send(new Worker(0).run(10000000000L, 10000),
                        runResponseProcessor);
            }
        }.signal();
    }
}
```

Output:

```
initialized
Worker0: started  0
Worker0: finished 1
finished
```

Remember the *AsyncRequestImpl.setMessageTimeoutMillis* method used in the *Worker run* request? Until now it has been passed a value of -1, which indicates that the default timeout should be used. In the VerySlow example, a timeout value of 10,000 is used. (10 seconds.) A large timeout value should always be used for messages might take some time to process, to avoid closing reactors needlessly when the system becomes loaded.