Retep Tools

User Guide

Retep Development Group

Peter T Mount

Retep Tools: User Guide

Retep Development Group
by Peter T Mount

Table of Contents

1. annotations 1
Overview 1
2. cluster 3
Overview 3
3. collections 5
Overview 5
4. Concurrency support 7
Overview 7
Locking by annotation 7
ReadWrite locks 8
5. encoding 11
Overview 11
6. graphics 13
Overview 13
7. io 15
Overview 15
8. j2ee 17
Overview 17
9. jaxb 19
Overview 19
10. jmx 21
Overview 21
11. logging 23
Overview 23
12. math 25
Overview 25
13. The Generic Messaging API 27
Overview 27
Introduction 27
Concepts 27
Addressing 27
Component 28
Route 28
Router 28
A Simple example 28
Address 28
Message 28
Components 29
Router and Route 31
Deploy and Run 32
14. nio 35
Overview 35
15. NioSax - a SAX style XML Push Parser for Java NIO 37
Overview 37
Using NIOSax 37
Example 37
16. random 41
Overview 41
17. strings 43
Overview 43
18. swing 45

Overview 45
19. table 47
Overview 47
20. trees 49
Overview 49
21. xml 51
Overview 51
22. test-framework 53
Overview 53
A. BSD License 55
B. The Apache Software License, Version 1.1 57
Index 59

List of Examples

4.1. Standard way to use ReadWriteLock's 7
4.2. ReadWriteLock's with just annotations 9
4.3. ReadWriteLock's with annotations and ReadWriteConcurrencySupport 10
13.1. A simple Message implementation 29
13.2. Simple messaging component 30
13.3. Simple messaging component's 31
13.4. Simple messaging router 32
13.5. Deploying our example 33
15.1. Starting a NioSaxParser 38
15.2. Passing content to a NioSaxParser 38
15.3. Closing a NioSaxParser 39

Chapter 1. annotations

Overview

Holder for annotations

Chapter 2. cluster

Overview

Holder for cluster

Chapter 3. collections

Overview

Holder for collections

Chapter 4. Concurrency support

Overview

The retepTools library provides additional concurrency support to that provided by the java.util.concurrency package of JDK 1.5 or later.

Locking by annotation

The core component of the concurrency support provided by retepTools is the locking. By utilising a set of annotations, it is possible to mark a method so that the entire body of that method is bound to the scope of that Lock.

For example, we have a bean with a property. The properties value can be read by any number of Threads, but it can be set by only one at a time. Example 4.1, “Standard way to use ReadWriteLock's” shows the normal way you would write code to use a ReadWriteLock:

Example 4.1. Standard way to use ReadWriteLock's

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class MyBean {
    private final ReadWriteLock lock = new ReentrantReadWriteLock();

    private int value;

    public int getValue() {
        lock.readLock().lock();
        try {
            return value;
        } finally {
            lock.readLock().unlock();
        }
    }


    public void setValue( int newValue ) {
        lock.writeLock().lock();
        try {
            value = newValue;
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

Now imagine doing that on an object with dozens of properties... a lot of boiler plate code which is prone to typing errors. retepTools removes this boiler plate code by using a set of thee annotations: @Lock, @ReadLock and @WriteLock.

All three annotations follow a simple contract:

- your object must implement a corresponding method for each annotation.
- the associated method is named as it’s annotation, i.e. @ReadLock expects readLock(), etc.
- that method must be declared either private or “protected final”.
- the Lock object returned by those methods must also be final - specifically it must be the same object returned for every invocation of that method on that instance.

ReadWrite locks

For convenience the class uk.org.retep.util.concurrent.ReadWriteConcurrencySupport implements this contract for objects using @ReadLock and @WriteLock.

So Example 4.2, “ReadWriteLock's with just annotations” shows the above code rewritten to use the annotations:

Example 4.2. ReadWriteLock's with just annotations

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import javax.annotation.concurrent.ThreadSafe;
import uk.org.retep.annotations.Contract;
import uk.org.retep.annotations.ReadLock;
import uk.org.retep.annotations.WriteLock;

@ThreadSafe
public class MyBean {

    private final ReadWriteLock lock = new ReentrantReadWriteLock();

    private int value;

    @Contract( ReadLock.class )
    protected final Lock readLock() {
        return lock.readLock();
    }

    @Contract( WriteLock.class )
    protected final Lock writeLock() {
        return lock.writeLock();
    }

    @ReadLock
    public int getValue() {
        return value;
    }

    @WriteLock
    public void setValue( int newValue ) {
        value = newValue;
    }
}
```

Now thats alot cleaner, less error prone and yet easier to see what the business logic is rather than having it obscured by the locking mechanism.

retepTools also provides several concrete base classes to make this even easier. The main one is uk.org.retep.util.concurrent.ReadWriteConcurrencySupport which provides concrete implementations of readLock() and writeLock().

Example 4.1, “Standard way to use ReadWriteLock's” makes it even simpler by extending ReadWriteConcurrencySupport:

Example 4.3. ReadWriteLock's with annotations and ReadWriteConcurrencySupport

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import javax.annotation.concurrent.ThreadSafe;
import uk.org.retep.annotations.ReadLock;
import uk.org.retep.annotations.WriteLock;
import uk.org.retep.util.concurrent.ReadWriteConcurrencySupport;

@ThreadSafe
public class MyBean
    extends ReadWriteConcurrencySupport
{
    private int value;

    @ReadLock
    public int getValue() {
        return value;
    }

    @WriteLock
    public void setValue( int newValue ) {
        value = newValue;
    }
}
```

Now thats alot cleaner, less error prone and yet easier to see what the business logic is rather than having it obscured by the locking mechanism.

Chapter 5. encoding

Overview

Holder for encoding

Chapter 6. graphics

Overview

Holder for graphics

Chapter 7. io

Overview

Holder for io

Chapter 8. j2ee

Overview

Holder for j2ee

Chapter 9. jaxb

Overview

Holder for jaxb

Chapter 10. jmx

Overview

Holder for jmx

Chapter 11. logging

Overview

Holder for logging

Chapter 12. math

Overview

Holder for math

Chapter 13. The Generic Messaging API

Overview

A common use case is the routing of messages from one component within a system to another. The standard API for this type of messaging within Java is the Java Messaging Service or JMS - but that can be a bit overkill for small embedded systems or if you are implementing another existing messaging protocol like Jabber. Originating from the retepXMPP project, retepTools 9.2 introduces a core messaging API which can form the basis of many simple messaging systems.

Introduction

The Messaging API is relatively simple comprising of four main classes in the `uk.org.retep.util.messaging` package:

- MessagingService - a class that forms the entry point into a messaging system for messages
- Message - an interface defining the message that can be handled by the service
- Component - an interface defining a component that can accept messages
- Router - an interface defining a component that can route messages to another Router or component.

The API is totally generic so the above classes have no concept of the inplementation of an address (where messages are sent to) or the route required to send them between Routers - this is left entirely to the implementing code. The only requirement is that the implementing classes obey the contract between hashCode and equals.

The API also provides some concrete implementations of the interfaces. These implementations are themselves generic, but they are also fully concurrent allowing the implemented messaging system to operate in a fully multi-threaded environment.

The rest of this article will show how to implement a simple messaging system where a group of interacting components can pass messages comprising of simple POJO's between themselves.

Concepts

Addressing

The first concept within the Messaging API is addressing. An address is a POJO that defines the Component that the message should be sent to. It also defines where the message was sent from if a response is required. This POJO can be any Java object. For this example we will be using java.lang.String but for more complex systems you would implement a class specifically for that purpose.

For example, in Jabber an address is known as a JID (Jabber ID) and has the form node@domain/resource (where node and resource are optional). In retepXMPP the address is implemented by a JID class which holds those three components individually allowing the messages to be routed by domain first and then by node and finally by resource.

Component

The second concept is a Component, represented by the Component interface. It represents a POJO that can accept a Message for some form of processing.

Route

The third concept is a Route. A route is a POJO used by a Router to determine where to send messages. Usually a Route implementation would hold the part of an address relevant to the specific Router using it. You could think of the global Domain Name System (DNS). For DNS you could have an address of "retep.org". Then sending a messsage to that address would invove the core router using the Route "org" to send the message to the router handling "org". That router wouild then send the message to the end Component using the Route "retep".

Router

The final concept is a Router. A Router, represented by the Router interface is a specialised Component that can contain other Components. It deals with forwarding Message's to one or more Components by using Route's.

A Simple example

To show how simple it is to setup a MessagingService, we'll create a simple system comprising of three Components that will interact with each other when an event occurs. We'll call these components A, B and C (yes imaginative naming here).

Address

For this example our address is a `java.lang.String`.

Message

The first step is to create the message. This will hold an int describing the event.

Example 13.1. A simple Message implementation

```
package example;

import uk.org.retep.util.messaging.message.AbstractMessage;

public class Event
    extends AbstractMessage<String>
{
    private int event;

    public int getEvent()
    {
        return event;
    }

    public void setEvent( int event )
    {
        this.event = event;
    }
}
```

AbstractMessage is a concrete implementation of the Message interface implementing all of it's methods. The `<String>` generic defines the type of the address, in this case `java.lang.String`.

Components

Now for our three components. Here component A will simply send a message to B when it's setValue() method is called. B will log that event and forward the message to C. C will then log the event itself and reply back to the originator (in this case A) with the negative value. A will then log the returned value.

Example 13.2. Simple messaging component

```
package example;

import javax.annotation.concurrent.ThreadSafe;
import uk.org.retep.util.messaging.MessageException;
import uk.org.retep.util.messaging.component.AbstractComponent;

@ThreadSafe
public class A
    extends AbstractComponent<String, Event>
{
    public void setValue( int value )
    {
        System.out.printf( "A.setValue( %d )\n", value );

        Event event = new Event();
        event.setFrom( "A" );
        event.setTo( "B" );
        event.setEvent( value );

        try
        {
            send( event );
        }
        catch( MessageException ex )
        {
            ex.printStackTrace();
        }
    }

    @Override
    public void consume( Event message )
        throws MessageException
    {
        System.out.printf( "A received event from %s: %d\n",
        message.getFrom(),
        message.getEvent() );
    }
}
```

You'll notice that the setValue method creates a new Event, sets the from and to addresses, the value of the event and then sends the message via the send() method. Once the Component is attached to the MessagingSystem then that message will be sent on to its destination.

Example 13.3. Simple messaging component's

```
@ThreadSafe
public class B
    extends AbstractComponent<String, Event>
{
    @Override
    public void consume( Event event )
        throws MessageException
    {
        System.out.printf( "B received event from %s: %s\n",
        event.getFrom(),
        event.getEvent() );

        Event newEvent = new Event();
        newEvent.setFrom( event.getFrom() );
        newEvent.setTo( "C" );
        newEvent.setEvent( event.getEvent() );
        send( newEvent );
    }
}

@ThreadSafe
public class C
    extends AbstractComponent<String, Event>
{
    @Override
    public void consume( Event event )
        throws MessageException
    {
        System.out.printf( "C received event from %s: %s\n",
        event.getFrom(),
        event.getEvent() );

        Event newEvent = new Event();
        newEvent.setFrom( "C" );
        newEvent.setTo( event.getFrom() );
        newEvent.setEvent( -event.getEvent() );

        send( newEvent );
    }
}
```

Router and Route

Now every messaging service has a core router which accepts every message as it first enters the system. For this example we'll use the address as the route as it's a single keyword:

Example 13.4. Simple messaging router

```
package example;

import javax.annotation.concurrent.ThreadSafe;
import uk.org.retep.util.messaging.router.AbstractRouter;

@ThreadSafe
public class SimpleRouter
    extends AbstractRouter<String, String>
{
    @Override
    protected String getRoute( String to )
    {
        return to;
    }
}
```

The Generics here define the types for the Address and the Route respectively. Now as we are using String's then both are set to String.

Our router implements the getRoute() method which should form the Route for this instance from the to address. In this case it's just the address, but usually you would have some other logic here.

Deploy and Run

Now all there's left is to deploy the system and send a message:

Example 13.5. Deploying our example

```
package example;

import uk.org.retep.util.messaging.MessagingService;
import uk.org.retep.util.messaging.Router;
public class Main
{
    private Main()
    {
    }

    public static void main( String... args )
        throws Exception
    {
        Router<String, String> router = new SimpleRouter();

        MessagingService<String, String> messagingService =
            new MessagingService<String, String>( router );

        A a = new A();
        messagingService.addRoute( "A", a );
        messagingService.addRoute( "B", new B() );
        messagingService.addRoute( "C", new C() );
        messagingService.startService();
        a.setValue( 42 );
    }
}
```

Here we create the core Router and then the MessagingService using that router. We then create the A component and add it to the service. We then do the same for B and C. Finally we start the service, and then set the value of A to 42. When run we should then see the following as the event propagates through the system:

```
A.setValue( 42 )
B received event from A: 42
C received event from A: 42
A received event from C: -42
```

Note: C says it's received the event from A and not B because the event's from address is A and not B - this is from the statement `newEvent.setTo( event.getFrom() );` in B.

Now with this example it's pretty simple. We could have started the service before adding the components if we wanted to - components can be added or removed at will.

Also this example is bad in the sense that we are adding all components directly to the MessagingService.

This would be correct for components routed by the core route, but say we had a router called D containing a component called D/A - the component D/A would be attached to router D and D would be connected to the messaging service.

Chapter 14. nio

Overview

Holder for nio

Chapter 15. NioSax - a SAX style XML Push Parser for Java NIO

Overview

NioSax (pronounced “'Neo-Sax'”) provides a Java NIO friendly XML push parser similar in operation to SAX. Unlike SAX, with NioSax it is possible for the xml source to contain partial content (i.e. only part of the XML stream has been received over the network). When this occurs, instead of failing with an error, NioSax simply stops. As soon as your application receives more data you simply call the same instance of the parser again and it will resume parsing where it left off.

The public API consists of the classes within this package, although the bare minimum required for use are the NioSaxParser, NioSaxParserHandler and NioSaxSource classes.

Using NIOSax

To use NioSax you simply use NioSaxParserFactory to create a NioSaxParser, implement a SAX ContentHandler and finally create a NioSaxSource which references the content.

Then you can parse one or more ByteBuffer's by updating the NioSaxSource with each buffer and pass it to the NioSaxParser.parse(NioSaxSource) method.

The only other two things you must to do with the parser is to ensure that you call NioSaxParser.startDocument() prior to any parsing, and call NioSaxParser.endDocument() once you are done with the parser so any resources used can be cleaned up.

Example

Example 15.1, “Starting a NioSaxParser” creates an NioSaxParser, sets the ContentHandler and stores it somewhere so when you receive data from java nio:

Example 15.1. Starting a NioSaxParser

```
import java.nio.ByteBuffer;
import uk.org.retep.niosax.NioSaxParser;
import uk.org.retep.niosax.NioSaxParserFactory;
import uk.org.retep.niosax.NioSaxParserHandler;
import uk.org.retep.niosax.NioSaxSource;

public class MyParser
{
    private NioSaxParser parser;
    private NioSaxParserHandler handler;
    private NioSaxSource source;

    public void start()
    {
        NioSaxParserFactory factory = NioSaxParserFactory.getInstance();

        parser = factory.newInstance();
        parser.setHandler( handler );
        source = new NioSaxSource();

        parser.startDocument();
    }
}
```

Next, when you receive data from some nio source and have the data in a ByteBuffer you need to pass it to the parser, as shown in Example 15.2, “Passing content to a NioSaxParser”:

Example 15.2. Passing content to a NioSaxParser

```
    public void parse( ByteBuffer buffer )
    {
        // flip the buffer so the parser starts at the beginning
        buffer.flip();

        // update the source (presuming the buffer has changed)
        source.setByteBuffer( buffer );

        // Parse the available content then compact
        parser.parse( source );
        source.compact();
    }
```

Finally when you are done (i.e. the connection has been closed) you then call NioSaxParser.endDocument():

Example 15.3. Closing a NioSaxParser

```
public void close()
{
    // releases any resources and notifies the handler the docment has comp
    parser.endDocument();
}
```

Note: The above example presumes that the buffer you have received has its position set to the end of the available content, so the flip() sets the limit to the end of the available data and the position to the start.

One the parser has run, we compact the buffer. This moves any remaining bytes to the beginning of the buffer and sets the position to the end. This then allows you to add more data to the same buffer when it arrives.

The reason for this is that, depending on the encoding of the document, you may have only part of a character in the buffer. In this case the parser will stop at that point so when it resumes it carries on with what should now be a complete character.

One last thing, if you know that you have the same ByteBuffer each time, then you can set it when you create the NioSaxSource and not worry about updating it with every call to parse.

 This page intentionally left blank

Chapter 16. random

Overview

Holder for random

Chapter 17. strings

Overview

Holder for strings

Chapter 18. swing

Overview

Holder for swing

Chapter 19. table

Overview

Holder for table

Chapter 20. trees

Overview

Holder for trees

Chapter 21. xml

Overview

Holder for xml

Chapter 22. test-framework

Overview

Holder for test-framework

Appendix A. BSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the retep.org.uk nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Appendix B. The Apache Software License, Version 1.1

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowlegement:

   > "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)."

   Alternately, this acknowlegement may appear in the software itself, if and wherever such third-party acknowlegements normally appear.

4. The names "Ant" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED “AS IS” AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index

Symbols

@Lock, 7
@ReadLock, 7
@WriteLock, 7

A

AbstractMessage, 29
annotations, 1-1

B

ByteBuffer, 37

C

cluster, 3-3
collections, 5-5
Component, 27
Concurrency, 7-10
ContentHandler, 37

E

encoding, 11-11

G

graphics, 13-13

I

io, 15-15

J

j2ee, 17-17
java.util.concurrency, 7
jaxb, 19-19
jmx, 21-21

L

logging, 23-23

M

math, 25-25
Message, 27
Messaging API, 27-33
MessagingService, 27

N

nio, 35-35
NioSax NIO XML Parser, 37-39
NioSaxParser, 37, 37
NioSaxParserFactory, 37
NioSaxSource, 37

R

random, 41-41
ReadWriteConcurrencySupport, 8
Router, 27

S

SAX Parser
    NioSax NIO XML Parser, 37-39
strings, 43-43
swing, 45-45

T

table, 47-47
test-framework, 53-53
trees, 49-49

X

xml, 51-51
XML Parser
    NioSax NIO XML Parser, 37-39