developerWorks®

Java development 2.0: Introducing Kilim

An actor framework for Java concurrency

Andrew Glover

April 13, 2010

Concurrent programming is central to Java™ development 2.0, but probably not *thread-based* concurrency. Andrew Glover explains why actors trump threads for concurrent programming in multicore systems. He then introduces Kilim, an actor-based message-passing framework that weaves together concurrent and distributed programming.

View more content in this series

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Java concurrency

Debugging nondeterministic defects in multithreaded applications has to be one of the most painful and frustrating activities known to software developers. So, like a lot of people, I've been caught up in the excitement about concurrent programming with functional languages like Erlang and Scala.

Both Scala and Erlang employ the actor model, rather than threads, for concurrent programming. Innovations around the actor model aren't limited to just languages; the actor model is also accessible on Java-based actor frameworks like Kilim.

Kilim's approach to the actor model is intuitive, and as you'll soon see, the library makes building concurrent applications a breeze.

The multicore challenge

In 2005, Herb Sutter wrote a now famous article entitled, "The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software". In it, he tore apart the misplaced belief that Moore's Law would continue to unleash higher and higher CPU clock speeds.

Sutter predicted the end of the "free lunch" that had come of increasing the performance of software applications by piggy-backing on faster and faster chips. Instead, he said that obtaining an appreciable increase in application performance would require taking advantage of multicore chip architectures.

As it turns out, he was right. Chip manufacturers have hit a hard limit, with chip speeds stabilizing at around 3.5 GHz for some years now. Moore's Law is alive and well in the multicore arena with manufacturers increasing the number of cores on chips at an ever-increasing rate.

About this series

The Java development landscape has changed radically since Java technology first emerged. Thanks to mature open source frameworks and reliable for-rent deployment infrastructures, it's now possible to assemble, test, run, and maintain Java applications quickly and inexpensively. In this series, Andrew Glover explores the spectrum of technologies and tools that make this new Java development paradigm possible.

Sutter also noted that concurrent programming would allow developers to take advantage of multicore architectures. But, he added, "we desperately need a higher-level programming model for concurrency than languages offer today."

The basic programming model of languages. like the Java language, is *thread based*. While multithreaded applications aren't terribly hard to write, there are challenges to writing them *correctly*. What's difficult about concurrent programming is thinking in terms of concurrency with threads. Alternate concurrency models have arisen along these lines. One that is particularly interesting, and gaining mindshare in the Java community, is the actor model.

The actor model

The actor model is a different way of modeling concurrent processes. Rather than threads interacting via shared memory with locks, the actor model leverages "actors" that pass asynchronous messages using mailboxes. A *mailbox*, in this case, is just like one in real life — messages can be stored and retrieved for processing by other actors. Rather than sharing variables in memory, the mailbox effectively separates distinct processes from each other.

Actors act as separate and distinct entities that don't share memory for communication. In fact, actors can only communicate via mailboxes. There are no locks and synchronized blocks in the actor model, so the issues that arise from them — like deadlocks and the nefarious lost-update problem — aren't a problem. What's more, actors are intended to work concurrently and not in some sequenced manner. As such, actors are much safer (locks and synchronization aren't necessary) and the actor model itself handles coordination issues. In essence, the actor model makes concurrent programming easier.

The actor model is by no means new; it's been around for quite some time. Some languages, like Erlang and Scala, base their concurrency model on actors as opposed to threads. In fact, Erlang's success in enterprise situations (Erlang was created by Ericsson and has a rich history in the telecom world) has arguably made the actor model more popular and visible, and that has helped it become a viable option for other languages. Erlang is a shining example of the actor model's safer approach to concurrency.

Unfortunately, the actor model isn't built into the Java platform, but it is available to us in a variety of forms. The JVM's openness to alternate languages means that you can leverage actors via a Java-platform language like Scala or Groovy (see Resources to learn about Groovy's actor library, GPars). Also, you can try one of the Java-based libraries that enable the actor model, like Kilim.

Actors with Kilim

Kilim is a library written in Java that embodies the actor model. In Kilim, "actors" are represented by Kilim's `Task` type. `Task`s are lightweight threads; they communicate with other `Task`s via Kilim's `Mailbox` type.

`Mailbox`es can accept "messages" of any type. For example, the `Mailbox` type accepts `java.lang.Object`. `Task`s can send `String` messages or even custom message types — it's entirely up to you.

Everything is tied together in Kilim via method signatures; if you need to do something concurrently, you specify the behavior in a method by augmenting its signature to throw `Pausable`. Thus, creating concurrent classes in Kilim is as easy as implementing `Runnable` or extending `Thread` in Java. It's just that all the baggage that is attached to using `Runnable` or `Thread` (such as the keyword `synchronized`) is reduced.

Lastly, Kilim's magic is enabled by a post process, called a weaver, that alters the bytecode of classes. Methods containing the `Pausablethrows` clause are processed at runtime by a scheduler, which is part of the Kilim library. The scheduler manipulates a limited number of Kernel threads. It is able to leverage this pool for a higher number of lightweight threads, which can context-switch and start up quite fast. Each thread's stack is automatically managed.

Essentially, Kilim makes it easy and simple to create concurrent processes: just extend from Kilim's `Task` type and implement the `execute` method. After you compile your newly minted concurrent-capable class, run Kilim's weaver over it and you're cooking with oil!

Kilim is a little foreign at first, but it has a big payoff. The actor model (and thus Kilim) makes it easier and safer to write asynchronous-acting objects that depend on similar objects. You *can* do the same thing with Java's base thread model (like extending `Thread`), but it's more challenging because it throws you back into the world of locks and synchronization. Simply put: switching your concurrent programming model to actors makes multithreaded applications easier to code.

Kilim in action

In Kilim's actor model, messages are passed between processes via a `Mailbox`. In many ways, you can think of a `Mailbox` as a queue. Processes can put items into a mailbox and also pull items from a mailbox, and they do so in both a blocking and non-blocking manner (what is blocked is the underlying Kilim-implemented lightweight process, not a Kernel thread).

As an example of leveraging mailboxes in Kilim, I wrote two actors (`Calculator` and `DeferredDivision`) that extend from Kilim's `Task` type. These classes will work cooperatively in a concurrent manner. The `DeferredDivision` object will create a dividend and a divisor; however, it won't attempt to divide the two. Imagine that it is expensive to do division, consequently the `DeferredDivision` object will ask the `Calculator` type to handle that task.

The two actors communicate via a shared `Mailbox` instance that accepts a `Calculation` type. This message type is quite simple — a dividend and divisor are provided, consequently the `Calculator` will perform the calculation and then set the corresponding answer. The `Calculator` will then place this `Calculation` instance back into the shared `Mailbox`.

Calculation

Listing 1 shows the simple `Calculation` type. You'll note that this type doesn't require any special Kilim code. In fact, it's just an everyday Java bean.

Listing 1. A Calculation type message

```
import java.math.BigDecimal;

public class Calculation {
 private BigDecimal dividend;
 private BigDecimal divisor;
 private BigDecimal answer;

 public Calculation(BigDecimal dividend, BigDecimal divisor) {
  super();
  this.dividend = dividend;
  this.divisor = divisor;
 }

 public BigDecimal getDividend() {
  return dividend;
 }

 public BigDecimal getDivisor() {
  return divisor;
 }

 public void setAnswer(BigDecimal ans){
  this.answer = ans;
 }

 public BigDecimal getAnswer(){
  return answer;
 }

 public String printAnswer() {
  return "The answer of " + dividend + " divided by " + divisor +
    " is " + answer;
 }
}
```

DeferredDivision

The Kilim-specific classes come into play in the `DeferredDivision` class. This class does a number of things, but at a high level its job is simple: to create instances of `Calculation` with random numbers (of type `BigDecimal`) and send them off to the `Calculator` actor. What's more, this class also checks the shared `MailBox` to see if any `Calculation`s are in it. If a retrieved `Calculation` instance has an answer,`DeferredDivision` will print it out.

Listing 2. DeferredDivision creates random divisors and dividends

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.util.Date;
import java.util.Random;

import kilim.Mailbox;
import kilim.Pausable;
import kilim.Task;

public class DeferredDivision extends Task {

 private Mailbox<Calculation> mailbox;

 public DeferredDivision(Mailbox<Calculation> mailbox) {
  super();
  this.mailbox = mailbox;
 }

 @Override
 public void execute() throws Pausable, Exception {
  Random numberGenerator = new Random(new Date().getTime());
  MathContext context = new MathContext(8);
  while (true) {
   System.out.println("I need to know the answer of something");
   mailbox.putnb(new Calculation(
     new BigDecimal(numberGenerator.nextDouble(), context),
     new BigDecimal(numberGenerator.nextDouble(), context)));
   Task.sleep(1000);
   Calculation answer = mailbox.getnb(); // no block
   if (answer != null && answer.getAnswer() != null) {
    System.out.println("Answer is: " + answer.printAnswer());
   }
  }
 }
}
```

As you can see from Listing 2, the `DeferredDivision` class extends from Kilim's `Task` type, which essentially emulates the actor model. Note that this class also overrides `Task`'s `execute` method, which by default throws `Pausable`. `execute`'s actions will therefore be under the control of Kilim's scheduler. That is, Kilim will make sure that `execute` acts concurrently in a safe manner.

Inside the `execute` method, `DeferredDivision` creates instances of `Calculation` and places them into the `Mailbox`. It uses the `putnb` method to do this in a non-blocking manner.

After populating the `mailbox`, `DeferredDivision` sleeps — note that it isn't the Kernel thread that is sleeping, it's the lightweight Kilim-managed thread. When the actor wakes up, as previously mentioned, it checks the `mailbox` for any `Calculation`s. This call is also non-blocking, which means that `getnb` can return `null`. If `DeferredDivision` finds a `Calculation` instance and its `getAnswer` method has a value (that is, not a `Calculation` instance that has yet to be processed by the `Calculator` type), it prints the value to the console.

Calculator

On the other side of the `Mailbox` is the `Calculator`. Like the `DeferredDivision` actor defined in Listing 2, `Calculator` also extends from Kilim's `Task` and implements the `execute` method. It's important to note that both actors share the *same `Mailbox` instance*. They can't communicate with different `Mailbox`es; they need to share an instance. Accordingly, both actors accept a typed `Mailbox` via their constructor.

Listing 3. At last, the real worker: Calculator

```
import java.math.RoundingMode;

import kilim.Mailbox;
import kilim.Pausable;
import kilim.Task;

public class Calculator extends Task{

 private Mailbox<Calculation> mailbox;

 public Calculator(Mailbox<Calculation> mailbox) {
  super();
  this.mailbox = mailbox;
 }

 @Override
 public void execute() throws Pausable, Exception {
  while (true) {
   Calculation calc = mailbox.get(); // blocks
   if (calc.getAnswer() == null) {
    calc.setAnswer(calc.getDividend().divide(calc.getDivisor(), 8,
      RoundingMode.HALF_UP));
    System.out.println("Calculator determined answer");
    mailbox.putnb(calc);
   }
   Task.sleep(1000);
  }
 }
}
```

`Calculator`'s `execute` method, like `DeferredDivision`'s, loops continuously looking for items on the shared `Mailbox`. The difference is that `Calculator` invokes the `get` method, which is a blocking call. Accordingly, when a `Calculation` "message" does show up, it performs the required division calculation. Finally, the `Calculator` places the modified `Calculation` back into the `Mailbox` (in a non-blocking manner) and then takes a break. The sleep calls in both actors are there only to make reading the console easier.

Kilim's weaver

Earlier, I mentioned that Kilim works by byte-code manipulation via its *weaver*. This is simply a post-process that you run on your classes *after* you've compiled them. The weaver then adds some special code into the various classes and methods where the `Pausable` marker is found.

Invoking the weaver is simple. For instance, in Listing 4, I'm using Ant to call the Weaver. All I have to do is tell the Weaver where my desired classes are and where to put the resulting byte-code. In this case, I am instructing the Weaver to alter classes found in the `target/classes` directory and write the resulting byte code back into the same directory.

Listing 4. Ant invokes Kilim's weaver

```
<target name="weave" depends="compile" description="handles Kilim byte code weaving">
 <java classname="kilim.tools.Weaver" fork="yes">
  <classpath refid="classpath" />
  <arg value="-d" />
  <arg value="./target/classes" />
  <arg line="./target/classes" />
 </java>
</target>
```

After the code has been altered, I'm free to start leveraging Kilim at runtime provided I've included its .jar file in my classpath.

Kilim at runtime

Putting the two actors into motion is just like firing off two normal `Thread`s in Java code. You use the same shared `sharedMailbox` instance to create and seed the two actor instances, then invoke the `start` method to set the actors in motion.

Listing 5. A simple runner

```
import kilim.Mailbox;
import kilim.Task;

public class CalculationCooperation {
 public static void main(String[] args) {
  Mailbox<Calculation> sharedMailbox = new Mailbox<Calculation>();

  Task deferred = new DeferredDivision(sharedMailbox);
  Task calculator = new Calculator(sharedMailbox);

  deffered.start();
  calculator.start();

 }
}
```

Running these two actors yields the output shown in Listing 6. If you run this code, your output will probably look a little different, but the logic sequence of activities will line up. In Listing 6, `DeferredDivision` requests calculations and `Calculator` responds with an answer.

Listing 6. Your output will vary — actors are nondeterministic

```
[java] I need to know the answer of something
[java] Calculator determined answer
[java] Answer is: The answer of 0.36477377 divided by 0.96829189 is 0.37671881
[java] I need to know the answer of something
[java] Calculator determined answer
[java] Answer is: The answer of 0.40326269 divided by 0.38055487 is 1.05967029
[java] I need to know the answer of something
[java] Calculator determined answer
[java] Answer is: The answer of 0.16258913 divided by 0.91854403 is 0.17700744
[java] I need to know the answer of something
[java] Calculator determined answer
[java] Answer is: The answer of 0.77380722 divided by 0.49075363 is 1.57677330
```

In conclusion

The actor model facilitates concurrent programming by allowing a safer mechanism for message-passing between processes (or actors). Implementations of this model vary between languages and frameworks. I suggest checking out Erlang's actors, followed by Scala's. Both implementations are quite neat, given their respective syntax.

If you want to leverage "plain Jane" Java actors, then your best bet might be Kilim or a similar framework (see Resources). No free lunch is involved, but an actor-based framework does make concurrency programming, and leveraging multicore processes, much easier.