在 Java 中使用协程（Coroutine）

在讲到具体内容之前，不能不先讲下 Coroutine 的一些背景知识，来先具体了解下什么是 Coroutine。

1. 背景知识

现在的操作系统都是支持多任务的，多任务可通过多进程或多线程的方式去实现，进程和线程的对比就不在这里说了，在多任务的调度上操作系统采取抢占式和协作式两种方式，抢占式是指操作系统给每个任务一定的执行时间片，在到达这个时间片后如任务仍然未释放对 CPU 的占用，那么操作系统将强制释放，这是目前多数操作系统采取的方式；协作式是指操作系统按照任务的顺序来分配 CPU，每个任务执行过程中除非其主动释放，否则将一直占据 CPU，这种方式非常值得注意的是一旦有任务占据 CPU 不放，会导致其他任务”饿死”的现象，因此操作系统确实不太适合采用这种方式。

说完操作系统多任务的调度方式后，来看看通常程序是如何实现支持高并发的，一种就是典型的基于操作系统提供的多进程或多线程机制，每个任务占据一个进程或一个线程，当任务中有 IO 等待等动作时，则将进程或线程放入待调度队列中，这种方式是目前大多数程序采取的方式，这种方式的坏处在于如想支持高的并发量，就不得不创建很多的进程或线程，而进程和线程都是要消耗不少系统资源的，另外一方面，进程或线程创建太多后，操作系统需要花费很多的时间在进程或线程的切换上，切换动作需要做状态保持和恢复，这也会消耗掉很多的系统资源；另外一种方式则是每个任务不完全占据一个进程或线程，当任务执行过程中需要进行 IO 等待等动作时，任务则将其所占据的进程或线程释放，以便其他任务使用这个进程或线程，这种方式的好处在于可以减少所需要的原生的进程或线程数，并且由于操作系统不需要做进程或线程的切换，而是自行来实现任务的切换，其成本会较操作系统切换低，这种方式也就是本文的重点，Coroutine 方式，又称协程方式，这种方式在目前的大多数语言中都有支持。

各种语言在实现 Coroutine 方式的支持时，多数都采用了 Actor Model 来实现，Actor Model 简单来说就是每个任务就是一个 Actor，Actor 之间通过消息传递的方式来进行交互，而不采用共享的方式，Actor 可以看做是一个轻量级的进程或线程，通常在一台 4G 内存的机器上，创建几十万个 Actor 是毫无问题的，Actor 支持 Continuations，即对于如下代码：

Actor

act 方法

进行一些处理

创建并执行另外一个 Actor

通过消息 box 阻塞获取另一个 Actor 执行的结果

继续基于这个结果进行一些处理

在支持 Continuations 的情况下，可以做到消息 box 阻塞时并不是进程或线程级的阻塞，而只是 Actor 本身的阻塞，并且在阻塞时可将所占据的进程或线程释放给其他 Actor 使用，Actor Model 实现最典型的就是 erLang 了。

对于 Java 应用而言，传统方式下为了支持高并发，由于一个线程只能用于处理一个请求，即使是线程中其实有很多 IO 中断、锁等待也同样如此，因此通常的做法是通过启动很多的线程来支撑高并发，但当线程过多时，就造成了 CPU 需要消耗不少的时间在线程的切换上，从而出现瓶颈，按照上面对 Coroutine 的描述，Coroutine 的方式理论上而言能够大幅度的提升 Java 应用所能支撑的并发量。

2. 在 Java 中使用 Coroutine

Java 尚不能从语言层次上支持 Coroutine，也许 Java 7 能够支持，目前已经有了一个测试性质的版本[1]，在 Sun JDK 7 尚未正式发布的情况下如希望在 Java 中使用 Coroutine，Scala 或 Kilim 是可以做的选择，来分别看下。

Scala 是现在很火的语言之一，Twitter 消息中间件基于 Scala 编写更是让 Scala 名声鹊起，除了在语法方面所做出的改进外，其中一个最突出的特色就是 Scala Actor，Scala Actor 是 Scala 用于实现 Coroutine 的方式，先来具体看看 Scala 在 Coroutine 支持实现的关键概念。

- Actor

Scala Actor 可以看做是一个轻量级的 Java Thread，其使用方式和 Java Thread 基本也一致，继承 Actor，实现 act 方法，启动时也是调用 start 方法，但和 Java Thread 不同的是，Scala Actor 可等待外部发送过来的消息，并进行相应的处理。

- Actor 的消息发送机制

[1] http://weblogs.java.net/blog/forax/archive/2009/11/19/holy-crap-jvm-has-coroutinecontinuationfiber-etc

发送消息到 Actor 的方式有异步、Future 两种方式，异步即指发送后立即返回，继续后续流程，使用异步发送的方法为：actor ! MessageObject，其中消息对象可以为任何类型，并且 Scala 还支持一种称为 case Object 的对象，便于在收到消息时做 pattern matching。

Future 方式是指阻塞线程等待消息处理的结果，使用 Future 方式发送的方法为：actor !! MessageObject，在等待结果方面，Scala 支持不限时等待，限时等待以及等待多个 Future 或个别 Future 完成，使用方法如下：

```
val ft=actor !! MessageObject // Future 方式发送消息
val result=ft() //  不限时等待
val results=awaitAll(500,ft1,ft2,ft3)    //  限时等待多个 Future 返回值
val results=awaitEither(ft1,ft2) //  等待个别 future 完成
```

接收消息方通过 reply 方法返回 Future 方式所等待的结果。

- Actor 的消息接收机制

当代码处于 Actor 的 act 方法或 Actor 环境（例如为 Actor 的 act 方法调用过来的代码）中时，可通过以下两种方式来接收外部发送给 Actor 的消息：一为 receive 方式，二为 react 方式，代码例子如下：

```
receive{
    case MessageObject(args) => doHandle(args)
}
react{
    case MessageObject(args) => doHandle(args)
}
```

receive 和 react 的差别在于 receive 需要阻塞当前 Java 线程，react 则仅为阻塞当前 Actor，但并不会阻塞 Java 线程，因此 react 模式更适合于充分发挥 coroutine 带来的原生线程数减少的好处，但 react 模式有个缺点是 react 不支持返回。

receive 和 react 都有限时接收的方式，方法为：receiveWithin(timeout)、reactWithin(timeout)，超时的消息通过 case TIMEOUT 的方式来接收。

下面来看基于 Scala Actor 实现并发处理请求的一个简单例子。

```
class Processor extends Actor{
    def act(){
        loop{
```

```
                react{
                    case command:String => doHandle(command)
                }
            }
        }

        def doHandle(command:String){
            // 业务逻辑处理
        }
    }
```

当需要并发执行此 Processor 时，在处理时需要的仅为调用以下代码：

```
val processor=new Processor()
processor.start
processor ! “Hello”
```

从以上说明来看，要在旧的应用中使用 Scala 还是会有一些成本，部署运行则非常简单，在 Scala IDE Plugin 编写了上面的 scala 代码后，即生成了 java class 文件，可直接在 jvm 中运行。

Kilim 是由剑桥的两位博士开发的一个用于在 Java 中使用 Coroutine 的框架，Kilim 基于 Java 语法，先来看看 Kilim 中的关键概念。

- Task

可以认为 Task 就是 Actor，使用方式和 Java Thread 基本相同，只是继承的为 Task，覆盖的为 execute 方法，启动也是调用 task 的 start 方法。

- Task 的消息发送机制

Kilim 中通过 Mailbox 对象来发送消息，Mailbox 的基本原则为可以有多个消息发送者，但只能有一个消息接收者，发送的方式有同步发送、异步发送和阻塞线程方式的同步发送三种，同步发送是指保证一定能将消息放入发送队列中，如当前发送队列已满，则等待到可用为止，阻塞的为当前 Task；异步发送则是尝试将消息放入发送队列一次，如失败，则返回 false，成功则返回 true，不会阻塞 Task；阻塞线程方式的同步发送是指阻塞当前线程，并保证将消息发送给接收者，三种方式的使用方法如下：

```
mailbox.put(messageObject); // 同步发送
```

```
mailbox.putnb(messageObject); // 异步发送
mailbox.putb(messageObject); // 阻塞线程方式发送
```

- Task 的消息接收机制

Kilim 中通过 Mailbox 来接收消息，接收消息的方式有同步接收、异步接收以及阻塞线程方式的同步接收三种，同步接收是指阻塞当前 Task，直到接收到消息才返回；异步接收是指立刻返回 Mailbox 中的消息，有就返回，没有则返回 null；阻塞线程方式的同步接收是指阻塞当前线程，直到接收到消息才返回，使用方法如下：

```
mailbox.get(); // 同步接收，传入 long 参数表示等待的超时时间，单位为毫秒
mailbox.getnb(); // 异步接收，立刻返回
mailbox.getb(); // 阻塞线程方式接收
```

下面来看基于 Kilim 实现并发处理请求的一个简单例子。

```
public class Processor extends Task{
    private String command;
    public Processor(String command){
        this.command=command;
    }
    public void execute() throws Pausable,Exception{
        // 业务逻辑处理
    }
}
```

在处理时，仅需调用以下代码：

```
Task processor=new Processor(command);
processor.start();
```

从以上代码来看，Kilim 对于 Java 人员而言学习门槛更低，但对于需要采用 coroutine 方式执行的代码在编译完毕后，还需要采用 Kilim 的 kilim.tools.Weaver 类来对这些已编译出来的 class 文件做织入，运行时需要用织入后生成的 class 文件才行，织入的方法为：java kilim.tools.Weaver –d [织入后生成的 class 文件存放的目录] [需要织入的类文件所在的目录]，目前尚没有 Kilim IDE Plugin 可用，因此 weaver 这个过程还是比较的麻烦。

上面对 Scala 和 Kilim 做了一个简单的介绍，在实际 Java 应用中使用 Coroutine 时，通常会出现以下几种典型的更复杂的使用场景，由于 Actor 模式本身就是异步的，因此其天然对

异步场景支持的就非常好，更多的问题会出现在以下几个同步场景上，分别来看看基于 Scala、Kilim 如何来实现。

- Actor 同步调用

  Actor 同步调用是经常会出现的使用场景，主要为 Actor 发送消息给其他的 Actor 处理，并等待结果才能继续。

  - Scala

  对于这种情况，在 Scala 2.7.7 中，目前可采取的为以下两种方法：

  一种为通过 Future 方式发送消息来实现：

```
class Processor(command:String) extends Actor{
    def act(){
        val actor=new NetSenderActor()
        val ft=actor !! command
        println(ft())
    }
}
class NetSenderActor extends Actor{
    def act(){
        case command:String => {
            reply("received command:"+command)
        }
    }
}
```

  第二种为通过 receive 的方式来实现：

```
class Processor(command:String) extends Actor{
    def act(){
        val actor=new NetSenderActor()
        actor ! command
        var senderResult=""
        receive{
            case result:String => {
```

```
                    senderResult=result
                }
            }
            println(senderResult)
        }
    }
    class NetSenderActor extends Actor{
        def act(){
            case command:String => {
                sender ! (“received command:”+command)
            }
        }
    }
```

但这两种方式其实都不好，因为这两种方式都会造成当前 Actor 的线程阻塞，这也是因为目前 Scala 版本对 continuations 尚不支持的原因，Scala 2.8 版本将提供 continuations 的支持，希望到时能有不需要阻塞 Actor 线程实现上述需求的方法。

还有一种常见的场景是 Actor 调一段普通的 Scala 类，然后那个类中进行了一些处理，并调用了其他 Actor，此时在该类中如需要等待 Actor 的返回结果，也可使用上面两种方法。

- Kilim

在 Kilim 中要实现 Task 之间的同步调用非常简单，代码如下：

```
public class TaskA extends Task{
    public void execute() throws Pausable,Exception{
        Mailbox<Object> result=new Mailbox<Object>();
        Task task=new TaskB(result);
        task.start();
        Object resultObject=result.get();
        System.out.println(resultObject);
    }
}
```

```
public class TaskB extends Task{
    private Mailbox<Object> result;
    public TaskB(Mailbox<Object> result){
        this.result=result;
    }
    public void execute() throws Pausable,Exception{
        result.put("result from TaskB");
    }
}
```

Kilim 的 Mailbox.get 并不会阻塞线程，因此这种方式是完全满足需求的。

- 普通 Java 代码同步调用 Actor

由于已有的应用是普通的 Java 代码，经常会出现这样的场景，就是希望实现在这些 Java 代码中同步的调用 Actor，并等待 Actor 的返回结果，但由于 Scala 和 Kilim 都强调首先必须是在 Actor 或 Task 的环境下才行，因此此场景更佳的方式应为 Scala Actor(Kilim Task) → Java Code → Scala Actor(Kilim Task)，这种场景在对已有的应用中会是最常出现的，来看看在 Scala 和 Kilim 中如何应对这样的需求。

  - Scala

目前 Scala 中如希望在 Java Code 中调用 Scala Actor，并等待其返回结果，暂时还没办法，做法只能改为从 Java Code 中去调一个 Scala 的 Object，然后在这个 Object 中调用 Actor，并借助上面提到的 receive 或 future 的方法来获取返回值，最后将这个返回值返回 Java Code。

  - Kilim

目前 Kilim 中如希望实现上面的需求，其实非常简单，只需要在 Java Code 的方法上加上 Throw Pausable，然后通过 mailbox.get 来等待 Kilim Task 返回的结果即可，在 Kilim 中只要调用栈上的每个方法都有 Throw Pausable，就可在这些方法上做等待返回这类的同步操作。

从上面这两个最常见的需求来看，无疑 Kilim 更符合需求，但要注意的是对于 Kilim 而言，如果出现 Task → nonpausable method → pausable method 这样的状况时，pausable method 中如果想执行阻塞当前 Task 的操作，是无法做到的，只能改造成 Task (在 mailbox 上做等待，并传递 mailbox 给后续步骤) → nonpausable method (传递 mailbox) →

pausable method (将逻辑转为放入一个 Task 中，并将返回值放入传递过来的 mailbox)，这种状况在面对 spring aop、反射调用等现象时就会出现了，目前 kilim 0.6 的版本尚未提供更透明的使用方法，不过据 kilim 作者提供的一个试用版本，其中已经有了对于反射调用的透明化的支持，暂时在目前只能采用上述方法，迁移成本相对较大，也许以后的 kilim 版本会考虑这样的场景，提供相应的方法来降低迁移的成本。

3. 性能、所能支撑的并发量对比

在对 Scala、Kilim 有了这些了解后，来具体看看采用 Scala、Kilim 后与传统 Java 方式在性能、所能支撑的并发量上的对比。

- 测试模型

采用一个比较简单的模型进行测试，具体为有 4 个线程，这 4 个线程分别接收到了一定数量的请求，每个请求需要交给另外一个线程去执行，这个线程所做的动作为循环 10 次获取另外一个线程的执行结果，此执行线程所做的动作为循环 1000 次拼接一个字符串，然后返回。

- 实现代码

由于目前 Scala 版本对 Continuation 支持不够好，但上面的场景中又有此类需求，所以导致 Scala 版本的代码写的比较麻烦一些。

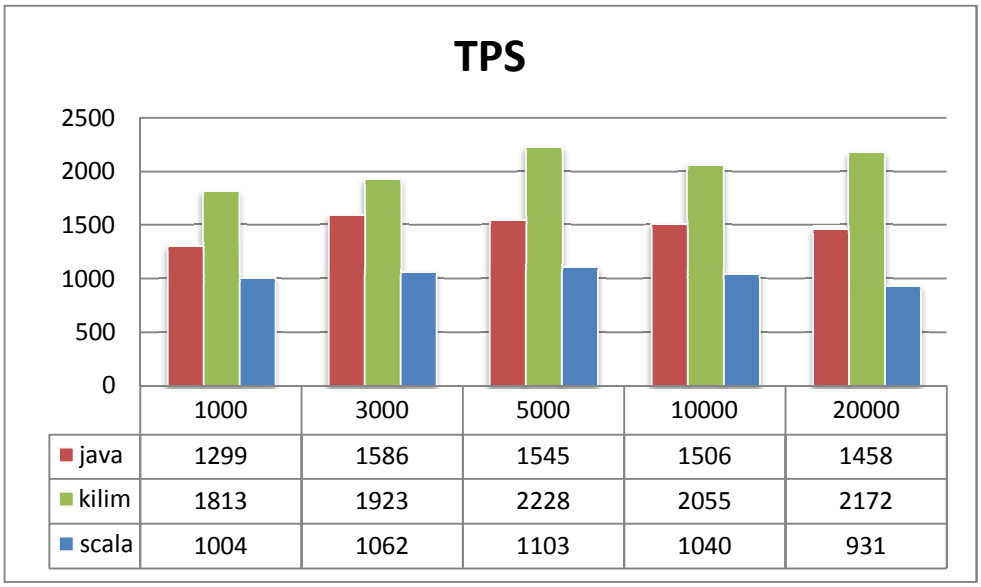
实现代码以及可运行的环境请从此处下载：

http://www.bluedavy.com/open/benchmark.zip

- 结果对比

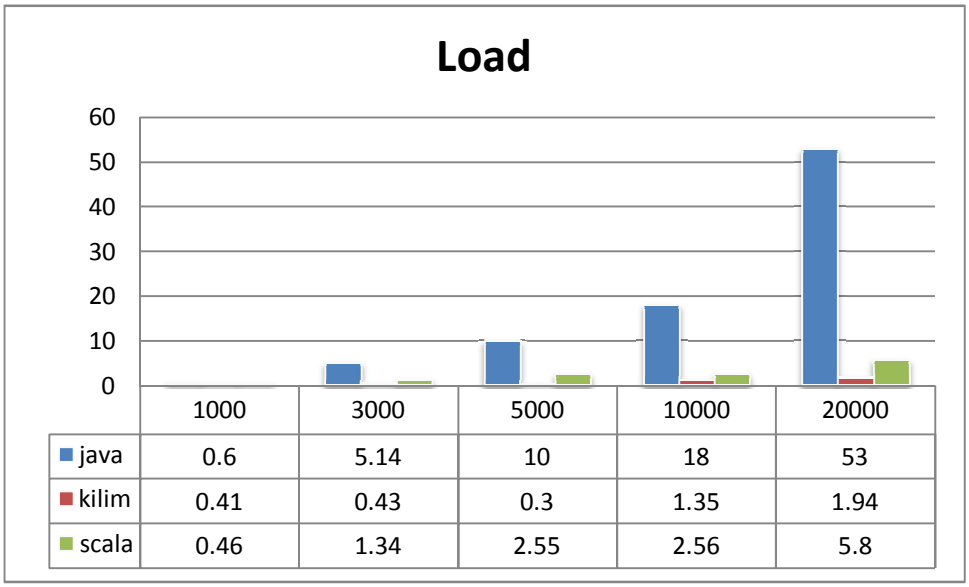
测试机器为一台 4 核的 linux 机器。

TPS 的对比结果如下：



	1000	3000	5000	10000	20000
java	1299	1586	1545	1506	1458
kilim	1813	1923	2228	2055	2172
scala	1004	1062	1103	1040	931

Load 的对比结果如下：



	1000	3000	5000	10000	20000
java	0.6	5.14	10	18	53
kilim	0.41	0.43	0.3	1.35	1.94
scala	0.46	1.34	2.55	2.56	5.8

从上面的测试结果来看，在这个 benchmark 的场景中，基于 Kilim 和 Scala 实现的 Coroutine 版本在随着请求数增长的情况下 load 的增长幅度都比纯粹的 Java 版本低很多，Kilim 版本表现尤其突出，在 TPS 方面，由于目前 Scala 版本对 Continuation 支持的不好，因此在这个测试场景中有点吃亏，表现反而最差，经过上面的测试可以看到，基于 Coroutine 版本可以以同样的 load 或更低的 load 来支撑更高的 TPS。

到此为止，基本上对 Java 中使用 Coroutine 的相关知识做了一个介绍，总结而言，采用 Coroutine 方式可以很好的绕开需要启动太多线程来支撑高并发出现的瓶颈，提高 Java 应用所能支撑的并发量，但在开发模式上也会带来变化，并且需要特别注意不能造成线程被阻塞的现象，从开发易用和透明迁移现有 Java 应用两个角度而言目前 Coroutine 方式还有很多不足，但相信随着越来越多的人在 Java 中使用 Coroutine，其易用性必然是能够得到提升的。

4. 参考资料

1. http://en.wikipedia.org/wiki/Computer_multitasking
2. http://en.wikipedia.org/wiki/Coroutine
3. http://en.wikipedia.org/wiki/Actor_model
4. http://en.wikipedia.org/wiki/Continuation
5. http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf
6. http://www.scala-lang.org/sites/default/files/odersky/jmlc06.pdf
7. http://www.malhar.net/sriram/kilim/kilim_ecoop08.pdf
8. http://lamp.epfl.ch/~phaller/doc/ScalaActors.pdf