链滴

二、Actor 与并发

作者：shiweichn

原文链接：https://ld246.com/article/1533310427213

来源网站：链滴

二、Actor 与并发

---

1 响应式系统设计

Akka 被称为是一个响应式平台，更具体的说，它是Typesafe 响应式平台的一部分。响应式宣言中包了4个准则，也可以说是设计目标：灵敏性、伸缩性、容错性以及事件驱动设计。

1.1 灵敏性

应用程序应该尽可能快的对请求做出响应。为了尽快向用户返回响应，在顺序获取数据和并行获数据之间选择的话，始终应该优先选择并行获取数据。如果可能出现错误，应该立即返回，将问题通用户，不要让用户等待直到超时。

1.2 伸缩性

应用程序应该能够根据不同的工作负载进行伸缩扩展（尤其是通过增加计算资源来进行扩展）。果在虚拟机上运行内存数据库，那么添加另一个虚拟节点就可以将所有的查询请求分布到两台虚拟服器上，将可能的吞吐量增加至原来的两倍。添加额外的节点应该能够几乎线性地提高系统的性能。

增加一个内存数据库的节点后，还可以将数据分为两半，并将其中的一半移至新的节点，这样就能够将内存容量提高至原来的两倍。添加节点应该能够几乎线性地提高内存容量。

1.3 容错性

如果系统的某个组件发生错误，对与该组件无关的请求不应该产生任何影响。错误是难以避免的因此应该将错误造成的影响限制在发生错误的组件内。

1.4 事件驱动/消息驱动

使用消息而不直接进行方法调用提供了一种帮助我们满足另外3个响应式准则的方法。消息驱动系统着重于控制何时、何地以及如何对请求做出响应，允许做出响应的组件进行路由以及负载均衡。

由于异步的消息驱动系统只有在真正需要时才会消耗资源（比如线程），因此它对系统资源的利更高效。消息也可以被发送到远程机器（位置透明）。因为要发送的消息赞成在 Actor 外的消息队列，并从该队列中被发出，所以就能够通过监督机制使得发生错误的系统进行自我恢复。

---

2 剖析 Actor

一个简单的例子，简单的构建一个 Actor ，这个Actor 接收 “Ping” ，返回字符串 “Pong”作为应。

```
package com.shiw.ak;

import akka.actor.AbstractActor;
import akka.actor.Status;
```

```
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.pf.ReceiveBuilder;

/**
 * Created by Sweeney on 2017/9/27.
 */
public class JavaPongActor extends AbstractActor {
    protected final LoggingAdapter log = Logging.getLogger(context().system(), this);

    @Override
    public Receive createReceive() {
        return ReceiveBuilder.create().matchEquals("Ping", message -> {
            sender().tell("Pong", self());
            log.info("message:" + message);
            log.info("sender:" + sender().path());
            log.info("self:" + self());
        }).matchAny(other -> {
            sender().tell(new Status.Failure(new Exception("unknown message")), self());
            log.info("other:" + other);
        }).build();
    }
}
```

- AbstractActor : 这个Java8特有的API，利用了 Lamdba 特性。UntypedActor 也可以作为基类来承，但是这个类比较旧。在 UntypeActor 的API中，会得到一个对象，然后必须用if语句对其进行条判断；但是在Java8的API可以通过模式匹配，表达力更强。
- Receive： AbstractActor类有一个 receive 方法，其子类必须实现这个方法或者在构造方法中调这个方法。receive 方法返回的类型是 PartialFuncation，这个类型来自于 Scala 的API 。在 Java中没有提供任何原生方法来构造 Scala 的 PartialFunction，因此 Akka 为我们提供了一个抽象的构造法类 ReceiveBuilder，用于生产 PartialFunction 作为返回值。但是新版本，有改动
- ReceiveBuilder：连续调用 ReceiveBuilder 的方法，为所有需要匹配处理的消息输入消息类型提响应方法的描述。然后调用 build() 方法生成所需要的返回值 PartialFunction。但是新版本，有改动
- Match : 用于匹配消息类型。match 函数从上至下匹配，所以可以先定义特殊情况，最后定义一般况。
    - match(final Class<? extends P> type, FI.UnitApply<? extends P> apply)
        - 描述了对于任何尚未匹配的该类型的实例，以及响应行为。
    - match(final Class<P> type, final FI.TypedPredicate<P> predicate,final FI.UnitApply<P> apply)
        - 描述了对于 predicate 条件函数为真的某特定类型的消息，应该如何响应。
    - matchAny(final FI.UnitApply<Object> apply)
        - 该函数匹配所有尚未匹配的消息，通常来说，最佳的事件是返回错误消息，或者记录错误信到日志。
- 向 sender() 返回消息：调用了 sender() 方法后，就可以返回所受到消息的响应了。响应的对象既以是Actor，也可以是来源于Actor系统外部的请求。第一种情况相当直接：上面的代码所示，返回的息会直接发送到该Actor的收件信箱中。
- tell()：sender()函数会返回一个 ActorRef。在 sender().tell() 中，tell() 是最基本的单项消息传输

式。第一个参数是要发送至对方邮箱的消息，第二个参数是希望对方Actor 看到的发送者。 ActorRef.oSender()则表示没有发送者，也就没有返回地址。

但是，在当前最新的2.5.4版本中，AbstractActor 中的一些方法被调整。比如上面书中所说要重写 receive方法，但是新版中改为必须要重写 createReceive 方法，且返回值由之前的 PartialFunction 为 Receive 。与之对应的 生产 PartialFunction 的 ReceiveBuilder 也做了调整，之前 ReceiveBuilder 中的 match 之类的方法由 static 全部变为非 static。build 方法也被重写。

```
// 旧版本的build方法 位于akka.japi.pf.AbstractPFBuilder
public PartialFunction<F, T> build() {
    PartialFunction<F, T> empty = CaseStatement.empty();
    PartialFunction<F, T> statements = this.statements;

    this.statements = null;
    if (statements == null)
      return empty;
    else
      return statements.orElse(empty);
  }

// 2.5.4 版本build方法 位于akka.japi.pf.ReceiveBuilder
public Receive build() {
    PartialFunction<Object, BoxedUnit> empty = CaseStatement.empty();

    if (statements == null)
      return new Receive(empty);
    else
      return new Receive(statements.orElse(empty)); // FIXME why no new Receive(statements)?
  }
```

---

3 创建 Actor

访问 Actor 的方式和访问普通对象的方式有所不同，我们从来不会得到 Actor的实例，也不调用 Acto的方法，也不直接改变 Actor 的状态，反之，只会向 Actor 发送消息。通过使用基于消息的机制，可相当完整的将 Actor 给封装起来，如果只通过消息通讯，那就永远不会需要获取 Actor 的实例，只需一种机制来支持向 Actor 发送消息并接受响应。—— ActorRef

在 Akka 中，这个指向 Actor 实例的引用叫做 ActorRef 。ActorRef 是一个无类型的引用，将其指向 Actor 封装起来，提供了更高层的抽象，并且给用户提供了一种与Actor进行通信的机制。

```
ActorRef pingref = system.actorOf(Props.create(JavaPongActor.class), "pingActor");
```

actorOf 方法会生成一个新的 Actor 并返回指向这个 Actor 的引用。

3.1 Props

```
def create(clazz: Class[_], args: AnyRef*): Props = new Props(deploy = Props.defaultDeploy, clazz = clazz, args = args.toList)
```

为了能够将 Actor 的实例封装起来，不让其被外部直接访问。我们将所有构造函数的参数传给一个 Pr ps 实例，Props 允许我们传入 Actor 类型以及一个可变参数列表。

actorOf 创建一个 Actor，并返回该 Actor 的引用 ActorRef，除此之外，还可以使用 actorSelection 来获取Actor 的 ActorRef。

每个 Actor 在创建的时候都会有一个路径。可以通过 ActorRef.path查看路径，如：

```
ActorRef pingref = system.actorOf(Props.create(JavaPongActor.class), "pingActor");
System.out.println(pingref.path());
```

输出：akka://PongPing/user/pingActor。该路径是一个URL，它甚至可以指向使用 akka.tcp 协议远程 Actor。例如书上：akka.tcp://my-sys@remotehost:5678/user/CharlieChaplin。

如果知道了 Actor 的路径，就可以使用 actorSelection 来获取指向该 Actor 引用的 ActorSelection 无论该 Actor 在本地还是在远程。

```
ActorRef pingref = system.actorOf(Props.create(JavaPongActor.class), "pingActor");
ActorSelection selection = system.actorSelection(pingref.path());
```

ActorSelection 也是一个指向 Actor 的引用。作用和 ActorRef 一样，同样可以使用 ActorSeletion Actor 之间互相通信。这也是对 Akka 位置透明性的最好诠释。

---

4 Promise、Futrue 和事件驱动的编程模型

4.1 阻塞

项目中最常见的就是阻塞式代码。一般的 web 服务器都会维护一个线程池，每来一个请求，就在池取出一个线程去处理，处理完后再将这个线程放回池中。这样避免了频繁的创建关闭线程，在一定程上提高了服务器性能。但是这种方式仍然是阻塞的，只不过是多线程阻塞式。如果在服务器负载比较的情况下，线程池中所有线程都被使用且都处于阻塞状态时，新的请求将会排队等待。这个时候 即使务器还有可用计算的资源，也没有任何线程来使用这些资源，因此 服务器无法充分利用资源。有人可会说 把线程池中的线程数量初始化多一点不就解决了？ 但是，这样就会导致系统把时间都花费在了要 CPU 的线程之间的上下文切换上，而没有更多的时间利用CPU进行实际的工作。时间片相关信息

4.2 异步非阻塞代码

使用Java8提供的CompletableFuture 可以写出 异步非阻塞 的代码。如下：

```
package com.shiw.ak;

import org.junit.Test;

import java.util.concurrent.CompletableFuture;

/**
 * Created by Sweeney on 2017/9/30.
 */
public class CompletableFutureTest {
    @Test
    public void test1() throws InterruptedException {
        String uid = "101001010101";
```

```
        CompletableFuture<String> async = CompletableFuture.supplyAsync(() -> getUsername
yUid(uid));
        async.thenAccept(s -> getPrintln(s));

        for (int i = 0; i < 5; i++) {
            System.out.println("i=" + i + "  , thread:" + Thread.currentThread().getName());
            Thread.sleep(1000);
        }
    }

    private void getPrintln(String x) {
        System.out.println("username:" + x + "  , thread:" + Thread.currentThread().getName());
    }

    public String getUsernameByUid(String str) {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if ("101001010101".equals(str)) {
            return "admin";
        }
        return "";
    }
}
```

输出：

```
i=0  , thread:main
i=1  , thread:main
i=2  , thread:main
i=3  , thread:main
username:admin  , thread:ForkJoinPool.commonPool-worker-1
i=4  , thread:main
```

可以看到 17 行的 for 并没有等待 15 行执行完就开始循环了。然后从输出结果中看出，当for循环输出 = 3之后，15行的代码开始执行。

从线程的角度看，代码首先会调用 getUsernameByUid 方法，然后进入该方法内部,接着几乎立即返一个 Future/CompletableFuture。返回的这个结果只是一个占位符，真正的值在未来某个时刻会最返回到这个占位符内，然后执行 getPirntln 方法。

对于 supplyAsync 方法，可以使用指定的线程池 例如：

```
ExecutorService threadPool = Executors.newFixedThreadPool(10);
CompletableFuture.supplyAsync(() -> getUsernameByUid(uid),threadPool);
```

4.3 理解 Futute

```
@Test
public void test2() throws InterruptedException {
    askPong("Ping").thenAccept(x -> System.out.println("replied with : " + x));
```

```
    Thread.sleep(100);
}

public CompletionStage<Object> askPong(String message) {
    CompletionStage<Object> ping = ask(pingref, message, 1000);
    return ping;
}
```

这是一段异步的代码（配合前面的代码）。Future 和 CompletableFuture 成功时会返回一个类型为 bject 的值，失败返回 Throwable 。

● 对返回结果执行代码

一旦结果返回就执行一个事件，可以用 thenAccept 来操作返回的结果，如下：

```
askPong("Ping").thenAccept(x -> System.out.println("replied with : " + x));
```

● 对返回结果进行转换

最常见的一种用例就是在处理响应之前先异步的对其进行转换,thenApply 操作会返回个新的 Future 包含 Char类型。

```
askPong("Ping").thenApply(x -> x.toString().charAt(0)).thenAccept(System.out::println);
```

● 对返回结果进行异步转换

有时候进行异步调用，在得到结果后，进行另一个异步调用，可以使用 thenCompose。

```
askPong("Ping").thenCompose(x -> askPong(x.toString()));
```

● 在失败的情况下执行代码

```
askPong("cause error").handle((x, t) -> {
          if (t != null) {
              System.out.println("Error: " + t);
          }
          return null;
      });
```

handle 接受一个 BiFunction 作为参数，该函数会对成功或失败情况进行转换。handle 中的函数在功的情况下会提供结果，在失败的情况下会提供 Throwable。因此只用检查 Throwable 是否存在即 。而失败的情况下又不需要对返回值做任何操作，因此直接返回 null 即可。

● 从失败中恢复

有时候即使发生错误，我们也想使用一个默认值来继续操作。在 Java中可以使用 exceptionally 将 Th owable 转换为一个可用的值。

```
askPong("Ping").exceptionally(t -> {
          return "default";
      });
```

● 异步的从失败中恢复

在发生错误的时候使用一个异步来恢复，例如：

- 重试某个失败的操作

- 没有命中缓存时，需要调用另一个服务操作。

```
askPong("cause error")
```

```
        .handle((pong, ex) -> ex == null
                ? CompletableFuture.completedFuture(pong)
                : askPong("Ping"))
        .thenCompose(x -> x);
```

首先检查exception是否为null ，如果为 null，就返回包含结果的 Future，否则返回重试的 Future 最后调用 thenCompose 将嵌套的 CompletionStage 扁平化。

4.4 链式操作

上面的每个方法都会返回一个新的 Future ，可以应用函数式的风格把多个操作组合起来，在组合的程中无需处理异常。我们可以把注意力放在成功的情况上，在链式操作的结尾再收集错误。

```
 askPong("Ping").thenCompose(x -> askPong("Ping" + x))
        .handle((x, t) -> {
            if (t != null) {
                return "default";
            } else {
                return x;
            }
        });
```

执行操作链中的任一操作时发生的错误都可以作为链的末端发生的错误来处理。这样就形成了一个很效的操作管道，无论是哪个操作导致了错误，都可以在最后来处理异常。我们可以集中注意力描述成的情况，无需在链的中间做额外的错误检查。可以在最后单独处理错误。