Actor编程

回顾：

今天任务：

了解并发编程；

了解Scala Actor原理；

学习使用Scala并发编程。

教学目标：

了解并发编程；

了解Scala Actor原理；

能运用Scala Actor完成并发编程的任务。

1.Java并发编程

1.1、new Thread的弊端

执行一个异步任务你还只是如下new Thread吗？

```
new Thread(new Runnable() {
@Override
public void run() {
    // TODO Auto-generated method stub
}
}).start();
```

那你就out太多了，new Thread的弊端如下： a. 每次new Thread新建对象性能差。 b. 线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom。 c. 缺乏更多功能，如定时执行、定期执行、线程中断。 相比new Thread，Java提供的四种线程池的好处在于： a. 重用存在的线程，减少对象创建、消亡的开销，性能佳。 b. 可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。 c. 提供定时执行、定期执行、单线程、并发数控制等功能。

1.2、Java 线程池

Java通过Executors提供四种线程池，分别为： newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。 newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。 newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。 newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。 (1). newCachedThreadPool 创建一个可缓存线程池，

如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

(2). newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadTest {
    public static void main(String[] args) throws Exception {
        // 创建一个固定线程数量的线程池，固定线程数量指的是线程数量上限
        final ExecutorService threadPool = Executors.newFixedThreadPool(10);
        // 创建一个可复用线程的线程池
//        final ExecutorService threadPool = Executors.newCachedThreadPool();


        for (int i = 0; i < 10; i++) {
            threadPool.execute(new Runnable() {

                public void run() {
                    System.out.println("Thread name:" + Thread.currentThread().getName()
                            + "======Thread id:" + Thread.currentThread().getId());
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }

        System.out.println(Thread.currentThread().getName());
        Thread.sleep(5000);

        for (int i = 0; i < 15; i++) {
            threadPool.execute(new Runnable() {

                public void run() {
                    System.out.println("Thread name:" + Thread.currentThread().getName()
                            + "*********Thread id:" + Thread.currentThread().getId());
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }

    }
}
```

(3) newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ThreadTest {
    public static void main(String[] args) throws Exception {

        final ScheduledExecutorService threadPool = Executors.newScheduledThreadPool(10);

        for (int i = 0; i < 10; i++) {
            threadPool.schedule(new Runnable() {

                public void run() {
                    System.out.println("Thread name:" + Thread.currentThread().getName()
                            + "======Thread id:" + Thread.currentThread().getId());
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //延迟5秒执行
            },5, TimeUnit.SECONDS);
        }

        System.out.println(Thread.currentThread().getName());
        Thread.sleep(5000);

        for (int i = 0; i < 15; i++) {
            threadPool.scheduleAtFixedRate(new Runnable() {

                public void run() {
                    System.out.println("Thread name:" + Thread.currentThread().getName()
                            + "*********Thread id:" + Thread.currentThread().getId());
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //延迟1秒后每3秒执行一次
            },1,3,TimeUnit.SECONDS);
        }

    }
}
```

(4)、newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ThreadTest {
    public static void main(String[] args) throws Exception {
        // 创建一个固定线程数量的线程池，固定线程数量指的是线程数量上限
        final ScheduledExecutorService threadPool =
Executors.newSingleThreadScheduledExecutor();

        for (int i = 0; i < 10; i++) {
            threadPool.execute(new Runnable() {

                public void run() {
                    System.out.println("Thread name:" + Thread.currentThread().getName()
                            + "======Thread id:" + Thread.currentThread().getId());
                    try {
                        Thread.sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
        }

}}
```

结果依次输出，相当于顺序执行各个任务。

1.3、submit和Future

execute() 方法没有返回值，无法获取任务执行的结果或者检查任务执行的状态。

```
executorService.execute(runnableTask)
```

submit() 方法提交一个Callable 或者Runnable 任务给 ExecutorService，返回 Future类型的结果.

```
Future<String> future =  executorService.submit(callableTask);
```

invokeAny() 提交一个任务集给ExecutorService,每一个任务都会被执行，如果其中一个成功执行，就返回其中该执行成功任务的结果 。

```
String result = executorService.invokeAny(callableTasks);
```

invokeAll() 提交一个任务集给ExecutorService,每一个任务都会被执行，返回所有任务成功执行后的Future类型的对象集合 .

```
List<Future<String>> futures = executorService.invokeAll(callableTasks);
```

```
import java.io.FileOutputStream;
import java.io.InputStream;
import java.net.URL;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureDemo {
    public static void main(String[] args) throws Exception {
        // 创建一个可复用线程的线程池
        ExecutorService threadPool = Executors.newCachedThreadPool();

        // Future相当于一个容器，可以封装返回值
        // 当计算过程没有完成的时候，此时会线程阻塞（线程等待），Future在线程阻塞是为空值
        // submit方法是有返回值的，返回类型就是Future
        final Future<String> future = threadPool.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                System.out.println("Thread name:" + Thread.currentThread().getName()
                        + "======Thread id:" + Thread.currentThread().getId());
                System.out.println("开始下载图片......");
               // Thread.sleep(2000);
                String url
="https://ss2.baidu.com/6ONYsjip0QIZ8tyhnq/it/u=1804135712,2125831907&fm=173&app=25&f=JPEG?
w=640&h=408&s=0E01B1440BE8A77E12DF118B0300E088";
                //Exception in thread "main" java.util.concurrent.ExecutionException:
java.io.FileNotFoundException: C:\temp (拒绝访问。)
               // String dest =  "C:\\temp\\" ;
                String dest =  "C:\\temp\\1.jpeg" ;
                FutureDemo.download(url,dest);
                System.out.println("图片下载完毕......");
                return "success";
            }
        });

        Thread.sleep(3000);

        System.out.println("**********************");
        if (future.isDone()){
            System.out.println(future.get());
        } else {
            System.out.println("Nothing ...");
        }
```

```
    }

    /**
     * Download image to local drive
     * @throws Exception
     */
    public static void download(String imageUrl,String destinationPath) throws Exception {
        // Open connetction to the image
        URL url = new URL(imageUrl);
        InputStream is = url.openStream();
        // Stream to the destionation file
        FileOutputStream fos = new FileOutputStream(destinationPath);

        // Read bytes from URL to the local file
        byte[] buffer = new byte[4096];
        int bytesRead = 0;
        while ((bytesRead = is.read(buffer)) != -1)
            fos.write(buffer,0,bytesRead);

        // Close destination stream
        fos.close();
        // Close URL stream
        is.close();
    }
}
```

1.4、关闭ExecutorService

一般情况下，ExecutorService 即使没有任务需要处理，也不会自动销毁，它会一直等待新任务的到来。在某些情况下，这非常有用，例如，如果一个app需要处理的任务不是规律出现的，或者在编译阶段任务的数量是无法预料的。另一方面APP总会有结束的时候，但是该APP不会停止，应为ExecutorService 导致JVM一直运行。这个时候我们可以关闭ExecutorService，使用shutdown() 或者shutdownNow().

shutdown()方法不会立即停止 ExecutorService.它会使得ExecutorService停止接受新任务，在所有正在执行的线程完成当前任务后关闭。

```
executorService.shutdown();
```

shutdownNow() 方法会立即销毁ExecutorService ,但是不保证所有正在运行的线程会同时停止，该方法会返回一个等待处理的任务列表，由开发者决定如何处理这个任务。

```
List<Runnable> notExecutedTasks = executorService.shutDownNow();
```

推荐的较好的关闭ExecutorService 是使用这些方法的同时，配合使用awaitTermination()方法. ExecutorService 方法首先停止接受新任务，同时会等待指定的时间让所有任务完成，如果超时，执行立即停止。

```
executorService.shutdown();
try {
    if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {
        executorService.shutdownNow();
    }
} catch (InterruptedException e) {
    executorService.shutdownNow();
}
```

1.5、Future 接口

submit() 和 invokeAll()方法返回一个对象或者一个 Future类型对象的集合，我们可以获取任务执行的结果或者检查任务的状态（是正在执行还是已经执行完毕）

Future接口提供了一个阻塞的get()方法，该方法返回Callable任务执行的结果或者Runnable 任务返回null。当任务在执行，调用该方法会导致阻塞直至相应任务已经执行完毕或者结果可获取。

```
Future<String> future = executorService.submit(callableTask);
String result = null;
try {
    result = future.get();
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

调用get()方法长时间的阻塞会导致应用性能的下降，如果结果不是很重要，可以设置超时避免该问题：

```
String result = future.get(200, TimeUnit.MILLISECONDS);
```

如果执行时间比指定的时长长，将会抛出超时异常。

isDone() 方法可以用来检查任务是否处理完毕。

cancel() 方法可以用来取消任务的执行，检查是否执行取消调用isCancelled() 方法。

```
boolean canceled = future.cancel(true);
boolean isCancelled = future.isCancelled();
```

2. Scala future

2.1 在Future中运行任务

future提供了一种高效非阻塞方式并行运行多个任务的方式，future是一个可以在未来的某个时间点给你一个结果的对象，是一个可以在未来执行的代码块。

```
Future{
    Thread.sleep(10000)
    println(s"This is the future at ${LocalTime.now}")
}

println(s"This is the present at ${LocalTime.now}")
```

当创建future时，它的代码会在某个线程上执行；

每个future在构造时，必须有一个指向ExecutionContext的引用，ExecutionContext类似于Java中的Executor;

最简单的引入方式是；

```
import  ExecutionContext.Implicits.global
```

这样任务会在一个全局的线程池中执行。

2.2 结果获取

当执行一个Future任务时，可以使用阻塞的调用来等待结果：

```
import scala.concurrent.duration._
val f =Future{Thread.sleep(10000);88}
val result = Await.result(f,10.seconds)
```

如果在分配的时间内任务没有就绪，Await.ready方法抛出TimeoutException。如果任务抛出了异常，该异常会在Await.result中调用中再次抛出。

使用回调的方式获取执行结果：

```
import ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.util.{Failure, Random, Success}

val f = Future{Thread.sleep(10000)
  if (new Random().nextFloat() < 0.5) throw new Exception
  42
}

f.onComplete{
  case Success(v) => println(s"The answer is $v")
  case Failure(ex) => println(ex.getMessage)
}
```

2.3 组合Future任务

如果我们需要获取两个任务的执行结果，并将执行结果合并起来，每个任务都是长时间运行的，应该放在Future中执行，组合方式如下：

```
val future1 = Future{getData1()}
val future2 = Future{getData2()}
val combined = for(n1 <- future1;n2 <- future2) yield n1+n2
```

2.4 Promise

Future的结果是任务结束或者失败时隐式的被设置的，Promise于Future类似，只是他的结果可以被显式的指定。

```
def computeAnswer(arg:String) = Future{
    val n = workHard(arg)
    n
}

def computeAnswer(arg:String) = {
val p = Promise[Int]()
Future{
      val n = workHard(arg)
      p.success(n)
      workOnSomethingElse()
}
    p.future
}
```

3. 了解Actor模型

注：我们现在学的Scala Actor是scala 2.10.x版本及以前版本的Actor。

Scala在2.11.x版本中将Akka加入其中，作为其默认的Actor，老版本的Actor已经废弃。

3.0 Actor模型

Actor模型是一个概念模型，用于处理并发计算。

3.1 什么是Actor

一个Actor指的是一个最基本的计算单元。它能接收一个消息并且基于其执行计算。

这个理念很像面向对象语言，一个对象接收一条消息（方法调用），然后根据接收的消息做事（调用了哪个方法）。

Actors一大重要特征在于actors之间相互隔离，它们并不互相共享内存。这点区别于上述的对象。也就是说，一个actor能维持一个私有的状态，并且这个状态不可能被另一个actor所改变。

聚沙成塔

One ant is no ant, one actor is no actor. 光有一个actor是不够的，多个actors才能组成系统。在actor模型里每个actor都有地址，所以它们才能够相互发送消息。

Scala中的Actor能够实现并行编程的强大功能，它是基于事件模型的并发机制，Scala是运用消息（message）的发送、接收来实现多线程的。使用Scala能够更容易地实现多线程应用的开发。

Actor是计算机科学领域中的一个并行计算模型，它把actors当做通用的并行计算原语：一个actor对接收到的消息做出响应，进行本地决策，可以创建更多的actor，或者发送更多的消息；同时准备接收下一条消息。

在Actor理论中，一切都被认为是actor，这和面向对象语言里一切都被看成对象很类似。但包括面向对象语言在内的软件通常是顺序执行的，而Actor模型本质上则是并发的。、

每个Actor都有一个(恰好一个)Mailbox。Mailbox相当于是一个小型的队列，一旦Sender发送消息，就是将该消息入队到Mailbox中。入队的顺序按照消息发送的时间顺序。Mailbox有多种实现，默认为FIFO。但也可以根据优先级考虑出队顺序，实现算法则不相同。

消息和信箱

异步地发送消息是用actor模型编程的重要特性之一。消息并不是直接发送到一个actor，而是发送到一个信箱（mailbox）

这样的设计解耦了actor之间的关系——actor都以自己的步调运行，且发送消息时不会被阻塞。虽然所有actor可以同时运行，但它们都按照信箱接收消息的顺序来依次处理消息，且仅在当前消息处理完成后才会处理下一个消息，因此我们只需要关心发送消息时的并发问题即可。

Actor之间通过发送消息来通信，消息的传送是异步的，通过一个邮件队列（mail queue）来处理消息。每个Actor是完全独立的，可以同时执行它们的操作。每一个Actor是一个计算实体，映射接收到的消息到以下动作：

- 发送有限个消息给其它Actor；
- 创建有限个新的Actor；
- 为下一个接收的消息指定行为。

以上三种动作并没有固定的顺序，可以并发地执行。Actor会根据接收到的消息进行不同的处理。

Actor模型有两种任务调度方式：基于线程的调度以及基于事件的调度：

- 基于线程的调度：为每个Actor分配一个线程，在接收一个消息时，如果当前Actor的邮箱（mail box）为空，则会阻塞当前线程。基于线程的调度实现较为简单，但线程数量受到操作的限制，现在的Actor模型一般不采用这种方式；
- 基于事件的调试：事件可以理解为上述任务或消息的到来，而此时才会为Actor的任务分配线程并执行。

综上，我们知道可以把系统中的所有事物都抽象成一个Actor：

- Actor的输入是接收到的消息。
- Actor接收到消息后处理消息中定义的任务。
- Actor处理完成任务后可以发送消息给其它的Actor。

3.2 Actors有邮箱

只得指明的一点是，尽管许多actors同时运行，但是一个actor只能顺序地处理消息。也就是说其它actors发送了三条消息给一个actor，这个actor只能一次处理一条。所以如果你要并行处理3条消息，你需要把这条消息发给3个actors。

消息异步地传送到actor，所以当actor正在处理消息时，新来的消息应该存储到别的地方。Mailbox就是这些消息存储的地方。

Actors通过异步消息沟通，在处理消息之前消息被存放在Mailbox中

3.3 Actors做什么

当一个actor接收到消息后，它能做如下三件事中的一件：

Create more actors; 创建其他actors Send messages to other actors; 向其他actors发送消息 Designates what to do with the next message. 指定下一条消息到来的行为 前两件事比较直观，第三件却很有意思。

我之前说过一个actor能维持一个私有状态。「指定下一条消息来到做什么」意味着可以定义下条消息来到时的状态。更清楚地说，就是actors如何修改状态。

设想有一个actor像计算器，它的初始状态是数字0。当这个actor接收到add(1)消息时，它并不改变它原本的状态，而是指定当它接收到下一个消息时，状态会变为1。

3.4 容错 Fault tolerance

Erlang 引入了「随它崩溃」的哲学理念，这部分关键代码被监控着，监控者的唯一职责是知道代码崩溃后干什么（如将这个单元代码重置为正常状态），让这种理念成为可能的正是actor模型。

每段代码都运行在process中，process是erlang称呼actor的方式。这个process完全独立，意味着它的状态不会影响其他process。我们有个supervisor，实际上它只是另一个process（所有东西都是actor），当被监控的process挂了，supervisor这个process会被通知并对此进行处理。这就让我们能创建「自愈」系统了。如果一个actor到达异常状态并崩溃，无论如何，supervisor都可以做出反应并尝试把它变成一致状态，这里有很多策略，最常见的是根据初始状态重启actor。

3.5 分布式 Distribution

另一个关于actor模型的有趣方面是它并不在意消息发送到的actor是本地的或者是另外节点上的。

转念一想，如果actor只是一些代码，包含了一个mailbox和一个内部状态，actor只对消息做出响应，谁会关注它运行在哪个机器上呢？只要我们能让消息到达就行了。这允许我们基于许多计算机上构建系统，并且恢复其中任何一台。

3.6. 对比传统java并发编程与Scala Actor编程

Java内置线程模型	**scala actor模型**
“共享数据-锁”模型（share data and lock）	share nothing
每个object有一个monitor，监视多线程对共享数据的访问	不共享数据，actor之间通过message通讯
加锁的代码段用synchronized标识	
死锁问题	
每个线程内部是顺序执行的	每个actor内部是顺序执行的

对于Java，我们都知道它的多线程实现需要对共享资源（变量、对象等）使用synchronized 关键字进行代码块同步、对象锁互斥等等。而且，常常一大块的try...catch语句块中加上wait方法、notify方法、notifyAll方法是让人很头疼的。原因就在于Java中多数使用的是可变状态的对象资源，对这些资源进行共享来实现多线程编程的话，控制好资源竞争与防止对象状态被意外修改是非常重要的，而对象状态的不变性也是较难以保证的。 而在Scala中，我们可以通过复制不可变状态的资源（即对象，Scala中一切都是对象，连函数、方法也是）的一个副本，再基于Actor的消息发送、接收机制进行并行编程

3.7. Actor方法执行顺序

1.首先调用start()方法启动Actor

2.调用start()方法后其act()方法会被执行

3.向Actor发送消息

3.8. 发送消息的方式

!	**发送异步消息，没有返回值。**
!?	发送同步消息，等待返回值。
!!	发送异步消息，返回值是 Future[Any]。

4. Actor实战

4.1. 第一个例子

```
import scala.actors.Actor

object MyActor1 extends Actor{
  //定义act方法
  def act(){
    for(i <- 1 to 10){
      println("actor-1 " + i)
      Thread.sleep(2000)
    }
  }
}

object MyActor2 extends Actor{
  //定义act方法
  def act(){
    for(i <- 1 to 10){
      println("actor-2 " + i)
      Thread.sleep(2000)
    }
  }
}

object ActorExample1 extends App{
  //启动Actor
  MyActor1.start()
  MyActor2.start()
}
```

说明：上面分别调用了两个单例对象的start()方法，他们的act()方法会被执行，相同与在java中开启了两个线程，线程的run()方法会被执行。

注意：这两个Actor是并行执行的，act()方法中的for循环执行完成后actor程序就退出了

4.2. 第二个例子

receive相当于是创建线程和销毁线程的过程。

可以不断地接收消息

接收消息方式1：receive

特点：要反复处理消息，receive外层用while(..), 不用的话只处理一次。

```
import scala.actors.Actor

class ActorExample2 extends Actor {

  override def act(): Unit = {
    while (true) {
 //通过调用Actor.receive来接收消息
//actor发送消息时，它并不会阻塞，当actor接收消息时，它也不会被打断。
//发送的消息在接收actor的邮箱中等待处理，直到actor调用receive方法
      receive {
        case "start" => {
          println("starting ...")
          Thread.sleep(5000)
          println("started")
        }
        case "stop" => {
          println("stopping ...")
          Thread.sleep(5000)
          println("stopped ...")
        }
      }
    }
  }
}

object ActorExample2 {
  def main(args: Array[String]) {
    val actor = new ActorExample2
    actor.start()
    //发送异步消息，感叹号就相当于是一个方法
    actor ! "start"
    actor ! "stop"
    println("消息发送完成！")
  }
}
```

说明：在act()方法中加入了**while** (**true**) 循环，就可以不停的接收消息

注意：发送start消息和stop的消息是异步的，但是Actor接收到消息执行的过程是同步的按顺序执行

4.3. 第三个例子

react类似线程池机制，可以复用线程。

不断地接收消息，使用react

特点：

(1) 从不返回; (2) 要反复执行消息处理，react外层用loop，不能用while(..); (3) 通过复用线程，比receive更高效，应尽可能使用react;

react方式会复用线程，比receive更高效

```
import scala.actors.Actor

class ActorExample3 extends Actor {

  override def act(): Unit = {
    loop {
      react {
        case "start" => {
          println("starting ...")
          Thread.sleep(5000)
          println("started")
        }
        case "stop" => {
          println("stopping ...")
          Thread.sleep(8000)
          println("stopped ...")
        }
      }
    }
  }
}

object ActorExample3 {
  def main(args: Array[String]) {
    val actor = new ActorExample3
    actor.start()
    actor ! "start"
    actor ! "stop"
    println("消息发送完成！")
  }
}
```

说明： react 如果要反复执行消息处理，react外层要用loop，不能用while

4.4. 第四个例子

结合case class发送消息

```
import scala.actors.{Actor, Future}

object ActorDemo2 extends Actor {
  override def act(): Unit = {
    while (true) {
      // 用到了偏函数
      receive {
        case "start"
        => println("starting...")
        case AsyncMsg(id, msg) => {
          println(s"id: $id, msg: $msg")
          sender ! ReplyMsg(3, "sucess")
        }
        case SyncMsg(id, msg) => {
          println(s"id: $id, msg: $msg")
          Thread.sleep(2000)
          sender ! ReplyMsg(4, "success")
        }
      }
    }
  }
}

object ActorTest{
  def main(args: Array[String]): Unit = {
    val actor = ActorDemo2.start()

    // 发送异步消息，没有返回值
    //    actor ! AsyncMsg(1, "xiaofang is my goddess")
    //    println("异步消息，没有返回值")

    // 发送同步消息，有返回值，会线程等待
    //    val content: Any = actor !? SyncMsg(2, "yaoyao is my right girl")
    //    println("同步消息发送完毕")
    //    println(content)

    // 发送异步消息，有返回值，返回值是 Future[Any]
    val reply: Future[Any] = actor !! AsyncMsg(3, "mimi is my love")
    Thread.sleep(1000)
    if (reply.isSet) {
      val applyMsg: Any = reply.apply()
      println(applyMsg)
    } else {
      println("Nothing ...")
    }

  }
}

case class AsyncMsg(id: Int, msg: String)
```

```scala
case class ReplyMsg(id: Int, msg: String)

case class SyncMsg(id: Int, msg: String)
```