Twitter future异步编程

基本语法

future构建

1. 构建一个异步future

1      long startTime = System.currentTimeMillis();

2      Future<String> future = FuturePools.unboundedPool().apply(() -> {

3        sleep(1);

4        return "sleep 1";

5      }).onSuccess(v->{

6          printTimeUse("success return", startTime);

7          return BoxedUnit.UNIT;

8      }).onFailure(t->{

9          t.printStackTrace();

10          return BoxedUnit.UNIT;

11      });;

12      sleep(5);

13      printTimeUse(startTime);

在主线程起一个future，该future会异步进行处理。同时可以为该future添加回调函数。

onSuccess( v –> {} )在future执行完成后进行调用；onFailure( Throwable t -> {} )在执行过程中发生异常会执行该回调函数，其中t是执行中抛出的异常。

1. 构建一个同步future, 阻塞主线程

1  Future<String> future2 = Future.apply(() -> {

2     sleep(3);

3     return "sleep 3";

4  });

因为编程中难免会有阻塞操作，所以要避免此类方式生成future。

1. 并发执行future，互相无影响，不影响主线程

1      Future<String> future1 = FuturePools.unboundedPool().apply(() -> {

2          sleep(2);

3          System.out.println("return future1");

4          return "suc";

5      });

6      Future<String> future2 = FuturePools.unboundedPool().apply(() -> {

7          sleep(1);

8          System.out.println("return future2");

9          return "suc";

10      });

11    //do something else

可以理解为向线程池提交了任务，但是没有执行get操作。

future组合

future通过map、flatmap将不同的future连接，可以组成一个执行链条，这也是future如此流行的基础。

1. future通过map映射为一个新的future，新future运行依赖原有future

1  public Future<String> getA() {

2        System.out.println("get A from testDaoRx begin");

3        Future<Integer> future = testDaoRx.getA();

4        return future.map(x->{

5            System.out.println("get A from testDaoRx sucss");

6            return String.valueOf(x) + " Str";

7        });

8    }

其中getA方法获取一个future，返回值是int。调用该future的map方法会返回另外一个新的future。return的类型也就是新future的类型，这个例子将getA返回的int转换成了String并返回，也就是getA的返回类型。

```

graph RL

future1:int-->future2:String

```

1. 多future并行执行，全部完成之后返回新的future。

方式1：使用join

1    public Future<String> getSumAB(){

2        System.out.println("get sumAB begin ");

3        Future<Integer> futureA = testDaoRx.getA();

4        Future<Integer> futureB = testDaoRx.getB();

5        return futureA.join(futureB).map(x->{

6            System.out.println("get a " + x.\_1 + "  get b " + x.\_2);

7            return "sum:" + (x.\_1 + x.\_2);

8        });

9    }

graph RL

future1:int-->future3:String

future2:int-->future3:String

方式2：使用collect

1    public Future<String> getSumABCollect() {

2        System.out.println("get sumAB collect begin ");

3        Future<Integer> futureA = testDaoRx.getA();

4        Future<Integer> futureB = testDaoRx.getB();

5        Future<List<Integer>> listFuture = Future.collect(Lists.newArrayList(futureA, futureB));

6        return listFuture.map(x->{

7            System.out.println("get list" + x);

8            return "sum:" + (x.stream().reduce((sum, n) -> sum + n).orElse(0));

9        });

10  }

其中collect方式要求子future返回类型相同

1. future串行执行，futureA执行之后再执行futureB

1    public Future<String> getSumABSeq() throws Exception {

2        System.out.println("get sumAB begin ");

3        Future<Integer> futureA = testDaoRx.getA();

4        Future<String> futureStr = futureA.flatMap(x -> {

5          return testDaoRx.getB().map(y -> {

6                return x + y;

7            });

8        }).map(sum->{

9            return "str:"+sum;

10      });

11      return futureStr;

12  }

graph RL

future1:int-->future2:int

future2:int-->future3:String

```

flatMap之后返回的类型是Future<Integer>，之后使用map转换结果类型成String。下面是lambda简写方式：

1    public Future<String> getSumABSeqSimplified() throws Exception {

2        System.out.println("get sumAB begin ");

3        Future<Integer> futureA = testDaoRx.getA();

4        return futureA.flatMap(x -> testDaoRx.getB().map(y -> x + y)).map(sum-> "str:"+sum);

5    }

通过并行和串行两种组织方式就可组织成适用于复杂业务场景的调用链

```

graph RL

F\_db1-->F\_bus1

F\_db2-->F\_bus1

F\_rpc1-->F\_bus2

F\_bus1-->F\_bus2

F\_rpc3-->F\_rpc2

F\_rpc2-->F\_bus2

F\_bus2-->F\_Service1

```

future使用

1. block等待结果

future支持同步等待future运行结果。针对一个同步返回的rpc接口，使用future首先要构建调用链条。然后同步等待结果返回。

1    public void testSumAB() throws Exception {

2        Long startTime = System.currentTimeMillis();

3        Future<String> futureStr = testBizRx.getSumAB();

4  //        Future<String> futureStr = testBizRx.getSumABCollect();

5        String str = Await.result(futureStr);

6        System.out.println(str + " timeUse " + (System.currentTimeMillis() - startTime) / 1000 + "s");

7

8    }

1. block等待，增加超时限制

1    public void testGetAWithTimeoutAwait() throws Exception {

2        Long startTime = System.currentTimeMillis();

3        Future<String> futureStr = testBizRx.getA();

4

5        String str = Await.result(futureStr, Duration.fromMilliseconds(3000));

6

7        System.out.println(str + " timeUse " + (System.currentTimeMillis() - startTime) / 1000 + "s");

8

9    }

使用Await方式设置超时时间，Duration.fromMilliseconds(3000)代表3s超时

1. Future单独设置超时限制

将普通函数调用包装成future时进行超时的指定

方式1：使用Promise，超时之后返回null但是不抛出异常

1    public <R> Future<R> get(Function0<R> function, int milliseconds) {

2        Future<R> future = futurePool.getWrapperFuturePool().apply(function);

3        Promise<R> p = new Promise<>();

4        future.within(timer, Duration.fromMilliseconds(milliseconds))

5              .onSuccess(v->{

6                  p.setValue(v);

7                  return BoxedUnit.UNIT;

8              })

9              .onFailure(v1 -> {

10                p.setValue(null);

11                return BoxedUnit.UNIT;

12      });

13

14      return p;

15  }

方式2：超时后返回timeoutException

1    public <R> Future<R> getWithException(Function0<R> function, int milliseconds) {

2        Future<R> future = futurePool.getWrapperFuturePool().apply(function);

3        return future.within(timer, Duration.fromMilliseconds(milliseconds));

4    }

1. Future方法内部抛出异常

1    public Future<Integer> getException() {

2        if(hasError)

3            return Future.exception(new RuntimeException(" throw exception"));

4        //something else

5        ......

6    }