源自知乎：Project Reactor全解

转载：Project Reactor全解 - 知乎 (zhihu.com)

现在， Java 的各种基于 Reactor 模型的响应式编程库或者框架越来越多了，像是 RxJava，Project Reactor，Vert.x 等等等等。在 Java 9， Java 也引入了自己的 响应式编程的一种标准接口，即 `java.util.concurrent.Flow` 这个类。这个类里面规定了 Java 响应式编程所要实现的接口与抽象。我们这个系列要讨论的就是 `Project Reactor` 这个实现。

这里也提一下，为了能对于没有升级到 Java 9 的用户也能兼容， `java.util.concurrent.Flow` 这个类也被放入了一个 jar 供 Java 9 之前的版本，依赖是：

```
<dependency>
  <groupId>org.reactivestreams</groupId>
  <artifactId>reactive-streams</artifactId>
  <version>1.0.3</version>
</dependency>
```

本系列所讲述的 Project Reactor 就是 reactive-streams 的一种实现。 首先，我们先来了解下，什么是响应式编程，Java 如何实现

什么是响应式编程，Java 如何实现

我们这里用通过唯一 id 获取知乎的某个回答作为例子，首先我们先明确下，一次HTTP请求到服务器上处理完之后，将响应写回这次请求的连接，就是完成这次请求了，如下：

```
public void request(Connection connection, HttpRequest request) {
    //处理request,省略代码
    connection.write(response);//完成响应
}
```

假设获取回答需要调用两个接口，获取评论数量还有获取回答信息，传统的代码可能会这么去写：

```
//获取评论数量
public void getCommentCount(Connection connection, HttpRequest request) {
    Integer commentCount = null;
    try {
        //从缓存获取评论数量，阻塞IO
        commentCount = getCommnetCountFromCache(id);
    } catch(Exception e) {
        try {
            //缓存获取失败就从数据库中获取，阻塞IO
            commentCount = getVoteCountFromDB(id);
        } catch(Exception ex) {

        }
    }
    connection.write(commentCount);
}

//获取回答
public void getAnswer(Connection connection, HttpRequest request) {
    //获取点赞数量
    Integer voteCount = null;
    try {
        //从缓存获取点赞数量，阻塞IO
        voteCount = getVoteCountFromCache(id);
    } catch(Exception e) {
        try {
            //缓存获取失败就从数据库中获取，阻塞IO
            voteCount = getVoteCountFromDB(id);
        } catch(Exception ex) {

        }
    }
    //从数据库获取回答信息，阻塞IO
    Answer answer = getAnswerFromDB(id);
    //拼装Response
    ResultVO response = new ResultVO();
    if (voteCount != null) {
        response.setVoteCount(voteCount);
```

```
    }
    if (answer != null) {
        response.setAnswer(answer);
    }
    connection.write(response);//完成响应
}
```

在这种实现下，你的进程只需要一个线程池，承载了所有请求。这种实现下，有两个弊端：

1. 线程池 IO 阻塞，导致某个存储变慢或者缓存击穿的话，所有服务都堵住了。假设现在评论缓存突然挂了，全都访问数据库，导致请求变慢。由于线程需要等待 IO 响应，导致唯一一个线程池被堆满，无法处理获取回答的请求。
2. 对于获取回答信息，获取点赞数量其实和获取回答信息是可以并发进行的。不用非得先获取点赞数量之后再获取回答信息。

现在，NIO 非阻塞 IO 很普及了，有了非阻塞 IO，我们可以通过响应式编程，来让我们的线程不会阻塞，而是一直在处理请求。这是如何实现的呢？

传统的 BIO，是线程将数据写入 Connection 之后，当前线程进入 Block 状态，直到响应返回，之后接着做响应返回后的动作。NIO 则是线程将数据写入 Connection 之后，将**响应返回后需要做的事情以及参数缓存到一个地方**之后，直接返回。在有响应返回后，NIO 的 Selector 的 Read 事件会是 Ready 状态，扫描 Selector 事件的线程，会告诉你的线程池数据好了，然后线程池中的某个线程，拿出**刚刚缓存的要做的事情还有参数**，继续处理。

那么，怎样实现缓存响应返回后需要做的事情以及参数的呢？Java 本身提供了两种接口，一个是基于回调的 Callback 接口（Java 8 引入的各种Functional Interface），一种是 Future 框架。

基于 Callback 的实现：

```
//获取回答
public void getAnswer(Connection connection, HttpRequest request) {
    ResultVO resultVO = new ResultVO();
    getVoteCountFromCache(id, (count, throwable) -> {
        //异常不为null则为获取失败
        if (throwable != null) {
            //读取缓存失败就从数据库获取
            getVoteCountFromDB(id, (count2, throwable2) -> {
                if (throwable2 == null) {
                    resultVO.setVoteCount(voteCount);
                }
                //从数据库读取回答信息
                getAnswerFromDB(id, (answer, throwable3) -> {
                    if (throwable3 == null) {
                        resultVO.setAnswer(answer);
                        connection.write(resultVO);
                    } else {
                        connection.write(throwable3);
                    }
                });
            });
        } else {
            //获取成功，设置voteCount
            resultVO.setVoteCount(voteCount);
            //从数据库读取回答信息
            getAnswerFromDB(id, (answer, throwable2) -> {
                if (throwable2 == null) {
                    resultVO.setAnswer(answer);
                    //返回响应
                    connection.write(resultVO);
                } else {
                    //返回错误响应
                    connection.write(throwable2);
                }
            });
        }
    });
}
```

可以看出，随着调用层级的加深，callback 层级越来越深，越来越难写，而且啰嗦的代码很多。并且，基于 CallBack 想实现获取点赞数量其实和获取回答信息并发是很难写的，这里还是先获取点赞数量之后再获取回答信息。

那么基于 Future 呢？我们用 Java 8 之后引入的 `CompletableFuture` 来试着实现下。

TEXT 复制 全屏

```
//获取回答
public void getAnswer(Connection connection, HttpRequest request) {
    ResultVO resultVO = new ResultVO();
        //所有的异步任务都执行完之后要做的事情
        CompletableFuture.allOf(
                getVoteCountFromCache(id)
                        //发生异常，从数据库读取
                        .exceptionallyComposeAsync(throwable -> getVoteCountFromDB(id))
                        //读取完之后，设置VoteCount
                        .thenAccept(voteCount -> {
                    resultVO.setVoteCount(voteCount);
                }),
                getAnswerFromDB(id).thenAccept(answer -> {
                    resultVO.setAnswer(answer);
                })
        ).exceptionallyAsync(throwable -> {
            connection.write(throwable);
```

4. 记一次nacos启动报错：Cou
ass org.hibernate.validator.i
lueextraction.ValueExtractor
5. springboot集成rabbitMq时

推荐排行榜

1. centos安装mysql遇到的问题
2. 记一次@WebServlet不生效

```
        }).thenRun(() -> {
            connection.write(resultVO);
        });
}
```

这种实现就看上去简单多了，并且读取点赞数量还有读取回答内容是同时进行的。 Project Reactor 在 Completableuture 这种实现的基础上，**增加了更多的组合方式以及更完善的异常处理机制，以及面对背压时候的处理机制，还有重试机制。**

响应式编程里面遇到的问题 - 背压

由于响应式编程，不阻塞，所以把之前因为基本不会发生而忽视的一个问题带了上来，就是背压（Back Pressure）。

背压是指，当上游请求过多，下游服务来不及响应，导致 Buffer 溢出的这样一个问题。在响应式编程，由于线程不阻塞，遇到 IO 就会把当前参数和要做的事情缓存起来，这样无疑增大了很多吞吐量，同时内存占用也大了起来，如果不限制的话，很可能 OutOfMemory，这就是背压问题。

在这个问题上，Project Reactor 基于的模型，是有处理方式的，Completableuture 这个体系里面没有。

为何现在响应式编程在业务开发微服务开发不普及

主要因为数据库 IO，不是 NIO。

不论是Java自带的Future框架，还是 Spring WebFlux，还是 Vert.x，他们都是一种非阻塞的基于Ractor模型的框架（后两个框架都是利用netty实现）。

在阻塞编程模式里，任何一个请求，都需要一个线程去处理，如果io阻塞了，那么这个线程也会阻塞在那。但是在非阻塞编程里面，基于响应式的编程，线程不会被阻塞，还可以处理其他请求。举一个简单例子：假设只有一个线程池，请求来的时候，线程池处理，需要读取数据库 IO，这个 IO 是 NIO 非阻塞 IO，那么就将请求数据写入数据库连接，直接返回。之后数据库返回数据，这个链接的 Selector 会有 Read 事件准备就绪，这时候，再通过这个线程池去读取数据处理（相当于回调），这时候用的线程和之前不一定是同一个线程。这样的话，线程就不用等待数据库返回，而是直接处理其他请求。这样情况下，即使某个业务 SQL 的执行时间长，也不会影响其他业务的执行。

但是，这一切的基础，是 IO 必须是非阻塞 IO，也就是 NIO（或者 AIO）。**官方JDBC没有 NIO，只有 BIO 实现。**这样无法让线程将请求写入链接之后直接返回，必须等待响应。但是也就解决方案，就是通过其他线程池，专门处理数据库请求并等待返回进行回调，也就是业务线程池 A 将数据库 BIO 请求交给线程池B处理，读取完数据之后，再交给 A 执行剩下的业务逻辑。这样A也不用阻塞，可以处理其他请求。但是，这样还是有因为某个业务 SQL 的执行时间长，导致B所有线程被阻塞住队列也满了从而A的请求也被阻塞的情况，这是不完美的实现。真正完美的，需要 JDBC 实现 NIO。

Java 自带的 Future框架可以这么用JDBC：

```
@GetMapping
public DeferredResult<Result> get() {
DeferredResult<Result> deferredResult = new DeferredResult<>();
CompletableFuture.supplyAsync(() -> {
            return 阻塞数据库IO;
        //dbThreadPool用来处理阻塞的数据库IO
        }, dbThreadPool).thenComposeAsync(result -> {
    //spring 的 DeferredResult 来实现异步回调写入结果返回
    deferredResult.setResult(result);
});
return deferredResult;
}
```

WebFlux 也可以使用阻塞JDBC，但是同理：

```
@GetMapping
public Mono<Result> get() {
return Mono.fromFuture(CompletableFuture.supplyAsync(() -> {
            return 阻塞数据库IO;
        //dbThreadPool用来处理阻塞的数据库IO
        }, dbThreadPool));
}
```

Vert.x 也可以使用阻塞的JDBC，也是同理：

```
@GetMapping
public  DeferredResult<Result> get() {
DeferredResult<Result> deferredResult = new DeferredResult<>();
getResultFromDB().setHandler(asyncResult -> {
            if (asyncResult.succeeded()) {
                deferredResult.setResult(asyncResult.result());
            } else {
                deferredResult.setErrorResult(asyncResult.cause());
            }
        });
return deferredResult;
}

private WorkerExecutor dbThreadPool = vertx.createSharedWorkerExecutor("DB", 16);

private Future<Result> getResultFromDB() {
    Future<Result> result = Future.future();
    dbThreadPool.executeBlocking(future -> {
            return 阻塞数据库IO;
        }, false, asyncResult -> {
            if (asyncResult.succeeded()) {
                result.complete(asyncResult.result());
            } else {
```

```
                result.fail(asyncResult.cause());
            }
        });
    return result;
}
```

相当于通过另外的线程池（当然也可以通过原有线程池，反正就是要用和请求不一样的线程，才能实现回调，而不是当次就阻塞等待），封装了阻塞 JDBC IO。

但是，这样几乎对数据库IO主导的应用性能没有提升，还增加了线程切换，得不偿失。所以，需要使用真正实现了 NIO 的数据库客户端。目前有这些 NIO 的 JDBC 客户端，但是都不普及：

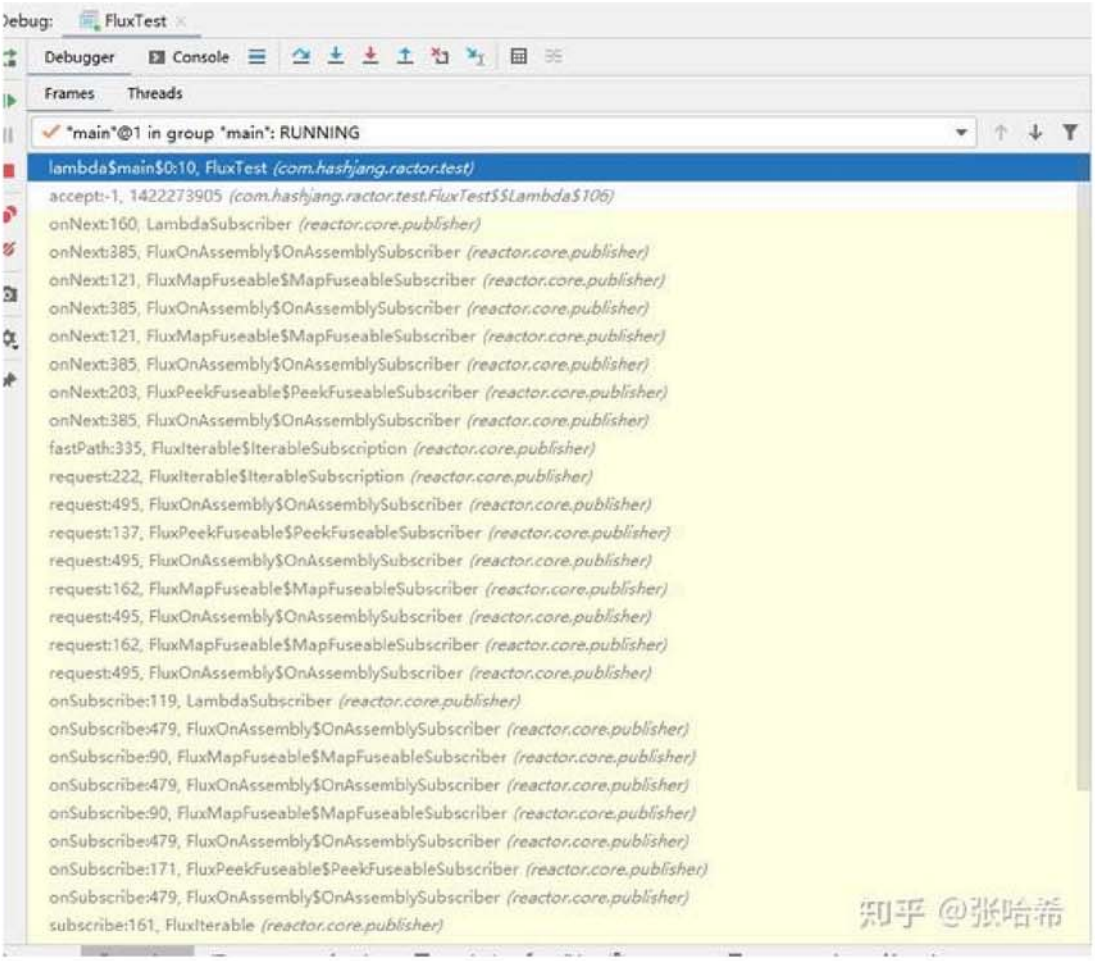
1. Vert.x 客户端：https://vertx.io/docs/vertx-jdbc-client/java/
2. r2jdbc 客户端：http://r2dbc.io/
3. Jasync-sql 客户端：https://github.com/jasync-sql/jasync-sql

响应式编程的首要问题 - 不好调试

我们在分析传统代码的时候，在哪里打了断点，就能看到直观的调用堆栈，来搞清楚，谁调用了这个代码，之前对参数做了什么修改，等等。但是在响应式编程中，这个问题就很麻烦。来看下面的例子。

```
public class FluxUtil1 {
    public static Flux<Integer> test(Flux<Integer> integerFlux) {
        return FluxUtil2.test2(integerFlux.map(Object::toString));
    }
}
public class FluxUtil2 {
    public static Flux<Integer> test2(Flux<String> stringFlux) {
        return stringFlux.map(Integer::new);
    }
}
public class FluxTest {
    public static void main(String[] args) {
        Flux<Integer> integerFlux = Flux.fromIterable(List.of(1, 2, 3));
        FluxUtil1.test(integerFlux.log()).subscribe(integer -> {
            System.out.println(integer);
        });
    }
}
```

我们调试到 subscribe 订阅消费（这个后面会讲），我们一般会想知道我们订阅的这个东西，之前经过了怎样的处理，但是在 `System.out.println(integer)` 打断点，看到的却是：



根本看不出来是 `FluxUtil1` ， `FluxUtil2` 处理过这个Flux。简单的代码还好，复杂起来调试简直要人命。官方也意识到了这一点，所以提供了一种在操作时捕捉堆栈缓存起来的机制。

这里我们先给出这些机制如何使用，后面我们会分析其中的实现原理。

1. 通过打开全局 Operator 堆栈追踪

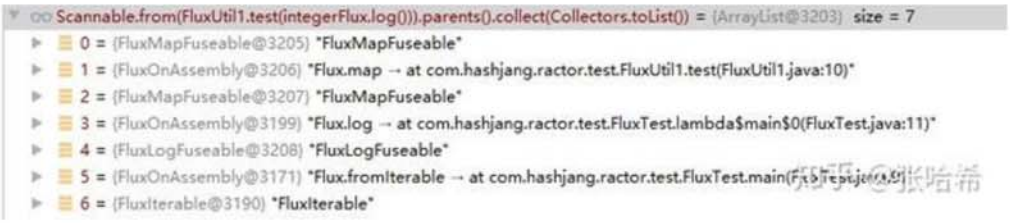
设置 `reactor.trace.operatorStacktrace` 这个环境变量为 true，即启动参数中加入 `-Dreactor.trace.operatorStacktrace=true` ，这样启动全局 Operator 堆栈追踪。

这个也可以通过代码动态打开或者关闭：

```
//打开
Hooks.onOperatorDebug();
//关闭
Hooks.resetOnOperatorDebug();
```

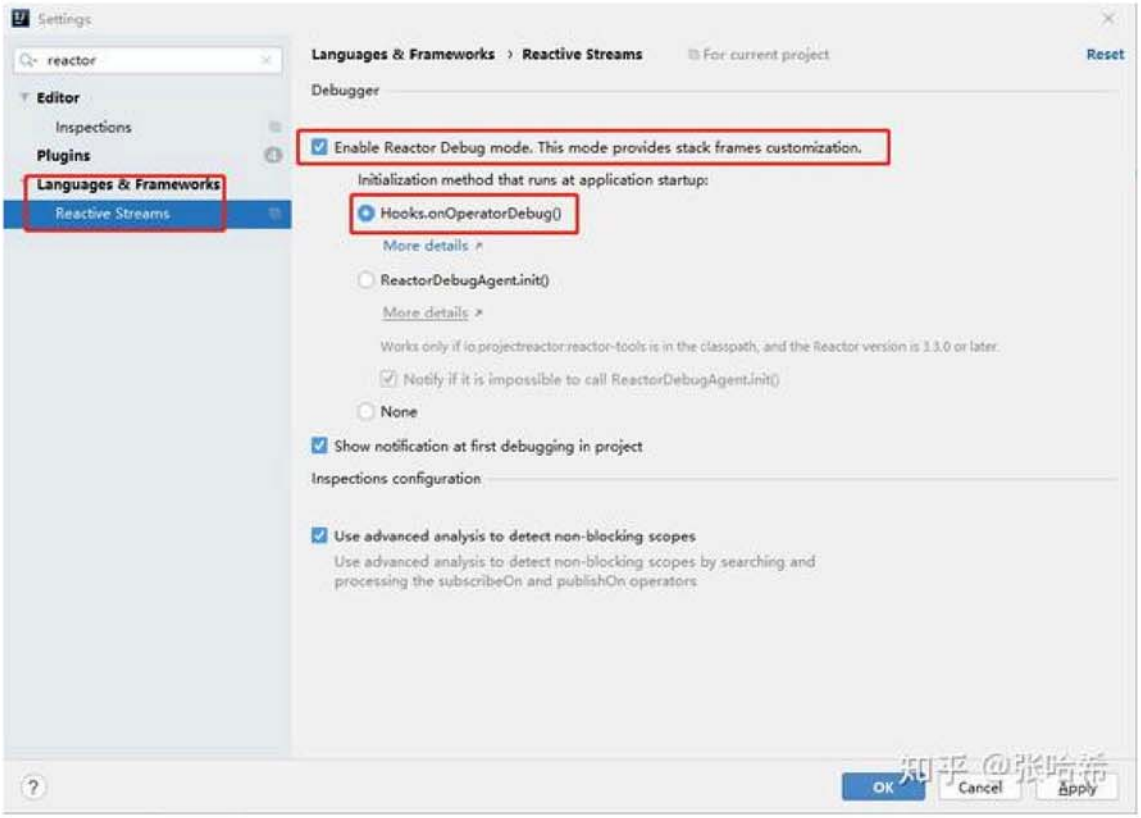
打开这个追踪之后，在每多一个 Operator，就会多出来一个 FluxOnAssembly（这个后面原理会详细说明）。通过这个 FluxOnAssembly，里面就有堆栈信息。怎么获取呢？可以通过 `Scannable.from(某个Flux).parents().collect(Collectors.toList())` 获取里面所有层的 Flux，其中包含了 FluxOnAssembly， FluxOnAssembly 就包含了堆栈信息。

我们这里，在 `System.out.println(integer)` 打断点，加入查看 `Scannable.from(FluxUtil1.test(integerFlux.log())).parents().collect(Collectors.toList())` ，就能看到：

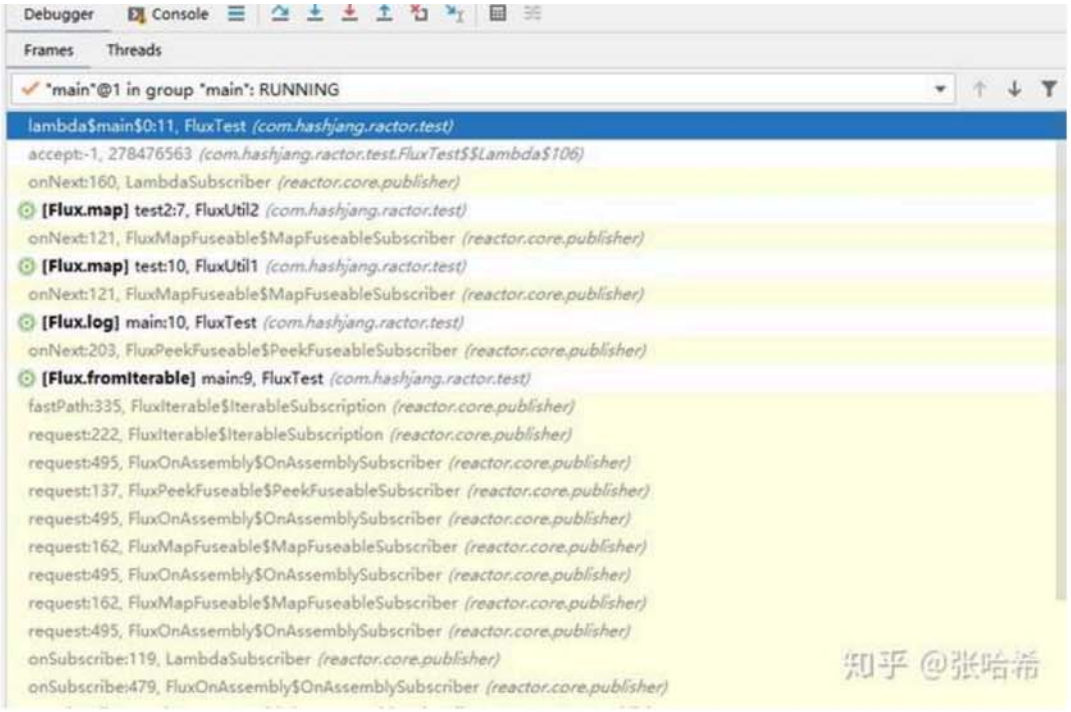


可以看出，每次 `map` 操作究竟发生在哪一行代码，都能看到。

如果使用的是专业版的 IDEA，还可以配置：



然后可以在打断点 Debug 就能看到具体堆栈：



2. 通过加入 ReactorDebugAgent 实现

添加依赖：

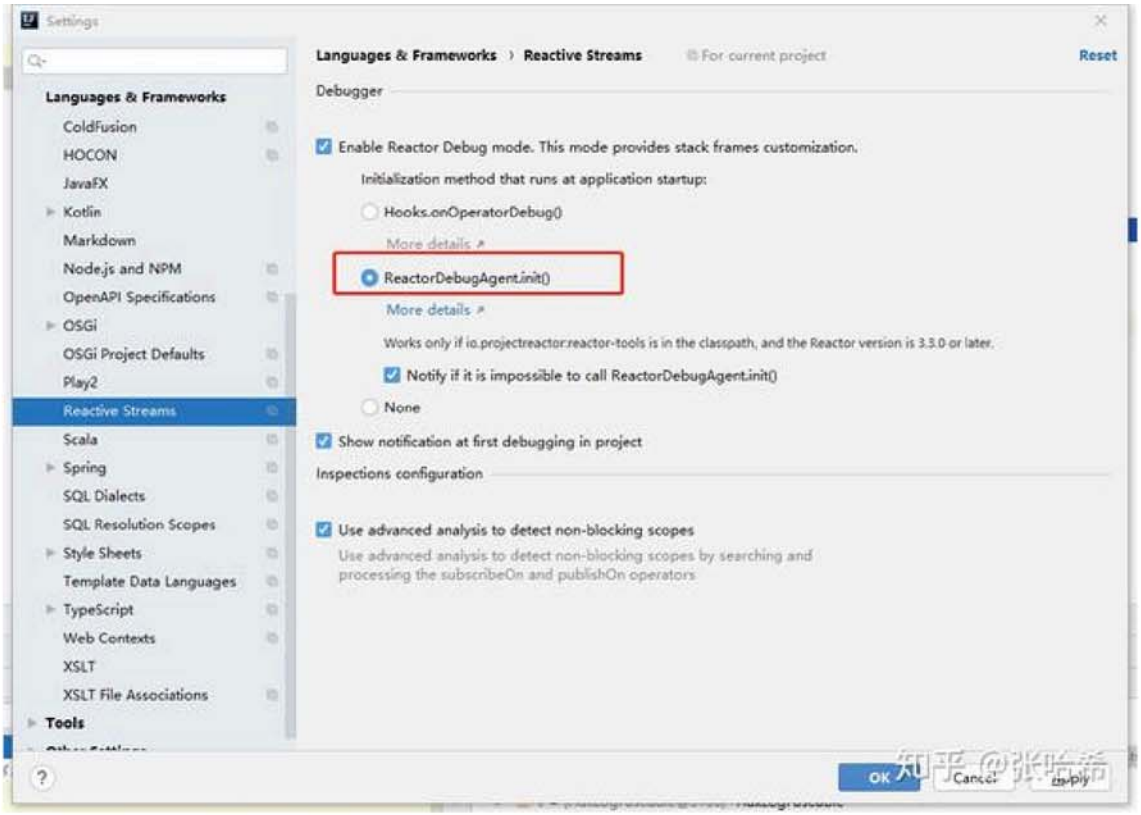
```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-tools</artifactId>
    <version>略</version>
</dependency>
```

之后，可以通过这两个代码，开启

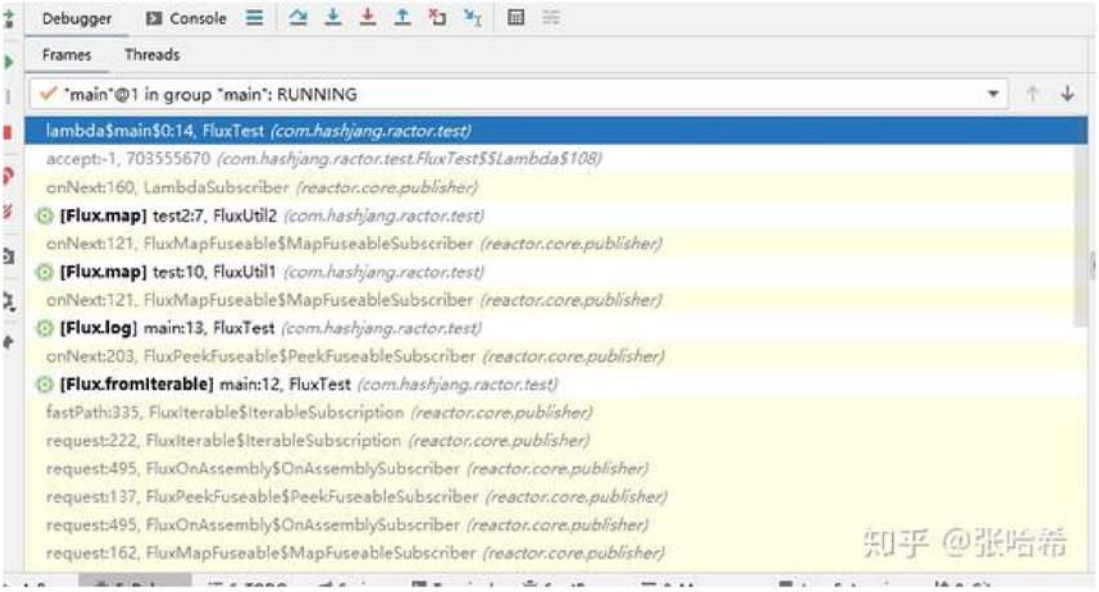
```
//启用
ReactorDebugAgent.init();
//如果有类没有生效，例如初始化没加载，后来动态加载的类，可以调用这个重新处理启用
ReactorDebugAgent.processExistingClasses();
```

这样，可以动态修改线上应用开启 `Debug` 模式，例如通过 Arthas 这个工具的 ognl 调用静态方法的功能（https://alibaba.github.io/arthas/ognl.html）。

如果使用的是专业版的 IDEA，还可以配置：



然后可以在打断点 Debug 就能看到具体堆栈：



响应式编程 - Flow 的理解

之前说过 FLow 是 Java 9 中引入的响应式编程的抽象概念，对应的类就是： `java.util.concurrent.Flow` Flow 是一个概念类，其中定义了三个接口供实现。这三个接口分别是： `Publisher` ， `Subscriber` 和 `Subscription` 。

```
//标注是一个FunctionalInterface，因为只有一个抽象方法
@FunctionalInterface
public static interface Publisher<T> {
    public void subscribe(Subscriber<? super T> subscriber);
}
public static interface Subscriber<T> {
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
}
public static interface Subscription {
    public void request(long n);
    public void cancel();
}
```

`Publisher` 是负责生成 item 的，其中的 `subscribe` 方法就是注册 `Subscriber` 进去，用于消费。注册成功后，会调用 `Subscriber` 的 `onSubscribe` 方法，传 `Subscription` 进来。这个 `Subscription` 里面的 request 用于请求 `Publisher` 发送多少 item 过来，cancel 用于告诉 `Publisher` 不要再发 item 过来了。每次 `Publisher` 有 item 生成并且没有超过 `Subscription` request 的个数限制， `onNext` 方法会被调用用于发送这个 item。当有异常发生时， `onError` 就会被调用。当 `Publisher` 判断不会有新的 item 或者异常发生的时候，就会调用 `onComplete` 告诉 `Subscriber` 消费完成了。大体上就是这么个流程。

Project Reactor 就是 `Flow` 的一种实现。并且在 `Flow` 这个模型的基础上，参考了 Java 8 Stream 的接口功能设计，加入了流处理的机制。

Project Reactor - `Flux` 如何实现 `Flow` 的接口

Flux就是一串相同类型数据的流，他包括并且会发射 0~n 个对象，例如：

```
Flux<String> just = Flux.just("1", "2", "3");
```

这样，我们就生成了一个包含三个字符串的Flux流（底层实现实际上就是FluxArray，这个我们以后会说的）

然后，我们按照之前 Flow 里面提到的流程，先进行简单的 `subscribe`

```
Flux.just("test1", "test2", "test3")
    //打印详细流日志
    .log()
    //订阅消费
    .subscribe(System.out::println);
```

运行代码，我们会看到日志输出：

```
07:08:13.816 [main] INFO reactor.Flux.Array.1 - | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
07:08:13.822 [main] INFO reactor.Flux.Array.1 - | request(unbounded)
07:08:13.823 [main] INFO reactor.Flux.Array.1 - | onNext(test1)
test1
07:08:13.823 [main] INFO reactor.Flux.Array.1 - | onNext(test2)
test2
07:08:13.823 [main] INFO reactor.Flux.Array.1 - | onNext(test3)
test3
07:08:13.824 [main] INFO reactor.Flux.Array.1 - | onComplete()
```

这些日志很清楚的说明了 `subscribe` 究竟是如何工作的：

1. 首先在 `subscribe` 的同时， `onSubscribe` 首先被调用
2. 然后调用 `request(unbounded)` ，这里 `request` 代表请求多少个数据， `unbounded` 代表请求无限个，就是所有的数据
3. 对于每个数据对象，调用 `onNext` 方法：onNext(test1)，onNext(test2)，onNext(test3)
4. 在最后完成的时候， `onComplete` 会被调用，如果说遇到了异常，那么 `onError` 会被调用，就不会调用 `onComplete` 了 这些方法其实都是 `Subscriber` 的方法， `Subscriber` 是Flux的订阅者，配置订阅者如何消费以及消费的具体操作。

```
Subscriber<String> subscriber = new Subscriber<String>() {
    //在订阅成功的时候，如何操作
    @Override
    public void onSubscribe(Subscription subscription) {
        //取最大数量的元素个数
        subscription.request(Long.MAX_VALUE);
    }

    //对于每个元素的操作
    @Override
    public void onNext(String o) {
        System.out.println(o);
    }

    //在发生错误的时候
    @Override
    public void onError(Throwable throwable) {
        log.error("error: {}", throwable.getMessage(), throwable);
    }

    //在完成的时候，发生错误不算完成
    @Override
    public void onComplete() {
        log.info("complete");
    }
};

Flux.just("test1", "test2", "test3")
    //打印详细流日志
    .log()
    //订阅消费
    .subscribe(subscriber);
```

运行后，日志是：

```
07:28:27.227 [main] INFO reactor.Flux.Array.2 - | onSubscribe([Synchronous Fuseable] FluxArray.ArraySubscription)
07:28:27.227 [main] INFO reactor.Flux.Array.2 - | request(unbounded)
07:28:27.228 [main] INFO reactor.Flux.Array.2 - | onNext(test1)
test1
07:28:27.228 [main] INFO reactor.Flux.Array.2 - | onNext(test2)
test2
07:28:27.228 [main] INFO reactor.Flux.Array.2 - | onNext(test3)
test3
07:28:27.228 [main] INFO reactor.Flux.Array.2 - | onComplete()
07:28:27.235 [main] INFO com.test.TestMonoFlux - complete
```

subscribe还有如下几个api：

```
//在不需要消费，只需要启动Flux中间处理的话，用这个
subscribe();
//相当于：
new Subscriber() {
    @Override
    public void onSubscribe(Subscription subscription) {
        //取最大数量的元素个数
        subscription.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(Object o) {
    }
    @Override
    public void onError(Throwable throwable) {
    }
    @Override
    public void onComplete() {
    }
};


//指定消费者消费
subscribe(Consumer<? super T> consumer);
//相当于：
new Subscriber() {
    @Override
    public void onSubscribe(Subscription subscription) {
        //取最大数量的元素个数
        subscription.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(Object o) {
        consumer.accept(o);
    }
    @Override
    public void onError(Throwable throwable) {
    }
    @Override
    public void onComplete() {
    }
};


//指定消费者，还有异常处理者
subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer);
//相当于：
new Subscriber() {
    @Override
    public void onSubscribe(Subscription subscription) {
        //取最大数量的元素个数
        subscription.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(Object o) {
        consumer.accept(o);
    }
    @Override
    public void onError(Throwable throwable) {
        errorConsumer.accept(throwable);
    }
    @Override
    public void onComplete() {
    }
};


//指定消费者，异常处理着还有完成的时候的要执行的操作
subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer, Runnable completeConsumer);
//相当于：
new Subscriber() {
    @Override
    public void onSubscribe(Subscription subscription) {
```

```
        //取最大数量的元素个数
        subscription.request(Long.MAX_VALUE);
    }
    @Override
    public void onNext(Object o) {
        consumer.accept(o);
    }
    @Override
    public void onError(Throwable throwable) {
        errorConsumer.accept(throwable);
    }
    @Override
    public void onComplete() {
        completeConsumer.run();
    }
};

//指定Subscriber所有需要的元素
subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer, Runnable completeConsumer, Consum
//相当于:
new Subscriber() {
    @Override
    public void onSubscribe(Subscription subscription) {
        subscriptionConsumer.accept(subscription);
    }
    @Override
    public void onNext(Object o) {
        consumer.accept(o);
    }
    @Override
    public void onError(Throwable throwable) {
        errorConsumer.accept(throwable);
    }
    @Override
    public void onComplete() {
        completeConsumer.run();
    }
};
```

这样，就和之前所说的 `Flow` 的设计对应起来了。

分类: java

posted @ 2021-08-07 02:02 冰肌玉骨小香脐 阅读(959) 评论(0) 编辑 收藏 举报

阅读排行：

- 园子周边第3季-博客园T恤：设计初稿第3版预览
- 5款开源、美观、强大的WPF UI组件库
- 使用 Docker 部署 TailChat 开源即时通讯平台
- Llama3-8B到底能不能打？实测对比
- C#S7.NET实现西门子PLCDB块数据采集的完整步骤