Linkedin 异步并行框架 ParSeq

--京东商城-成都研究院

--李俊林

目录

一、热身

EPT 库存系统库存预占流程描述：

1、 通过 SkuId 查询 SKU 信息。(seq1)

2、 参数校验、辅助数据填充。(seq2)

3、 查询库存数量（自动切换普通库存和活动库存）。(seq3)

4、 根据 SKU 维度查询总的剩余活动库存数量。(seq3)

5、 查询普通活动库存数量。(seq3)

6、 查询预占库存数量。(seq3)

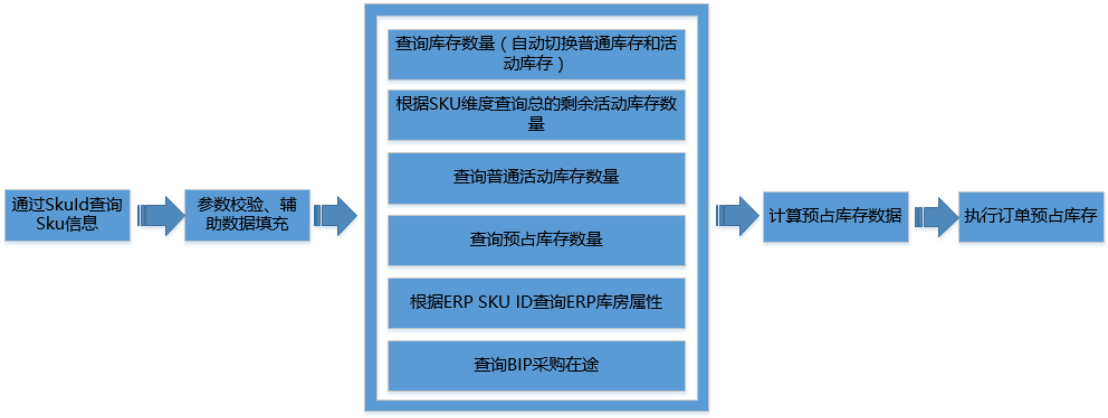
7、 根据 ERP SKU ID 查询 ERP 库房属性。(seq3)

8、 查询 BIP 采购在途。(seq3)

9、 计算预占库存数量详细、补货数量等数据。(seq4)

10、 执行订单预占库存逻辑。(seq5)

异步并行流程图：



二、简介

ParSeq 是 Linkedin 开源的异步并行框架。具有如下优点：

1、异步操作并行化处理。

2、顺序执行非阻塞性计算。

3、通过任务组合实现代码重用。

4、简单的错误传播和恢复机制。

5、执行跟踪和可视化。

6、批量执行异步操作。

获取 ParSeq ,目前最新版本是 v2.4.2 ,使用 ParSeq 的 v2.x 需要 jdk1.8.x 以上支持。引入 Maven 依赖：

```
<dependency>
        <groupId>com.linkedin.parseq</groupId>
        <artifactId>parseq</artifactId>
        <version>2.0.0</version>
</dependency>
```

三、关键概念解释

1、Task

Task 是 ParSeq 系统中一系列工作的基础，类似于 Java 的 Callable；但是 Task 可以异步的获取结果。Task 不能被用户直接执行，必须通过 Engine 执行。Task 实现了类似于 Java Future 的 Promise 接口。Task 可以被转换

和组合并最终执行得到预期的结果。

2、Plan

Plan 是一系列 Task 的集合，作为一个运行根 Task 的结果。

3、Engine

Task 的执行者，通常一个普通的应用程序使用一个 Engine 实例。

Engine 实例的代码如下：

```
import com.linkedin.parseq.Engine;
import com.linkedin.parseq.EngineBuilder;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
// ...
final int numCores = Runtime.getRuntime().availableProcessors();
final ExecutorService taskScheduler =
Executors.newFixedThreadPool(numCores + 1);
final ScheduledExecutorService timerScheduler =
    Executors.newSingleThreadScheduledExecutor();
   final Engine engine = new EngineBuilder()
          .setTaskExecutor(taskScheduler)
          .setTimerScheduler(timerScheduler)
```

```
        .build();
    //注：ParSeq 将会用 numCores + 1 个线程执行所有与 Task，一个线程
    用于调度定时器。
    这是一个比较合理的配置，但是你也可以自定义。
    //停止一个 Engine
    engine.shutdown();
    engine.awaitTermination(1, TimeUnit.SECONDS);
    taskScheduler.shutdown();
    timerScheduler.shutdown();
    //执行这段代码，Engine 将等待一秒钟之后关闭，在这个过程之中新的
任务不被执行，允许正在运行的任务执行完成。这个操作也会关闭被 ParSeq
使用的 Executors，但是 ParSeq 本身并不会管理这些 Executors 的生命周
期。
```

四、如何创建和运行一个 Task

初始化任务通过集成 ParSeq 现有库创建，如果 Task 包含非阻塞的计算，将使用 Task.action()或 Task.callable()创建 Task。对于大多数普通的 Task，我们也提供 Task.value 和 Task.failure()。新的 Task 是通过传输和组合现有 Task 完成的。

关于 ParSeq API 的建议：大多数创建带有版本 Task 的方法都接受对任务进行描述，建议给每一个 Task 都附上简单、清晰的描述，因为当使用 ParSeq 的跟踪机制来调试、解决问题的时候这会非常有用。

几乎 Task 接口的方法都将创建一个新的 Task 实例 ,其中也许依赖其它 Task 的执行结果，以及被引擎执行的时间。

Task 是懒惰的，它只是对能够被引擎执行的计算进行了描述，包括做什么、什么时候做。一旦你创建了一个 Task 你可以通过提交到 ParSeq 的执行引擎来执行该 Task（engine.run(task)）。

五、Task 的转换和组合

1、转换

转换 Task 的主要机制是通过 map()方法。假如我们仅仅需要 google 首页 HTTP 内容类型，使用 Task<Response>就能创建一个 HTTP HEAD 的请求：

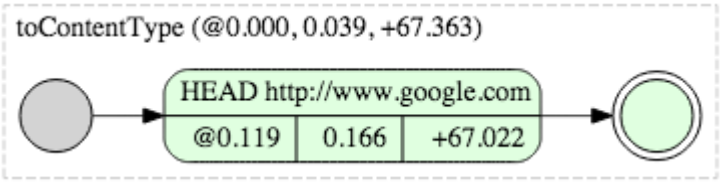
```
Task<Response> head = HttpClient.head("http://www.google.com").task();
```

我们可以使用下面的代码将其转换成一个可返回内容类型的 Task。

```
Task<String> contentType = head.map("toContentType", response -> response.getContentType());
```

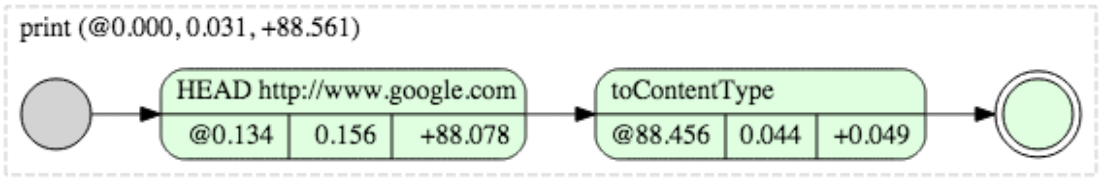
请注意，现有的 head Task 没有被修改。代替的是，一个新的任务被创建，head Task 什么时候第一次被运行，运行完成之后提供的转换是什么。如果这个 head Task 因为各种原因失败，那么 contentType 也应该失败，而且提供的转换不应该被调用。这个机制将在错误处理这一块进行详细解说。

使用 ParSeq 的跟踪工具我们可以得到如下的图形：



如果这里需要处理 Task 产生的结果，使用 andThen 方法就可以了：

Task<String> printContentType = contentType.andThen("print", System.out::println);

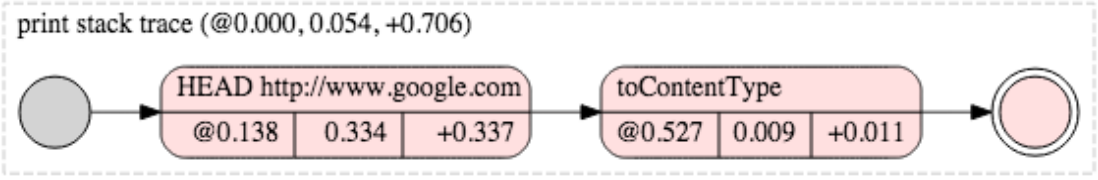


上面的例子中，我们使用 JAVA8 的方法引用，当然我们也可以使用 Lambda 表达式来完成：

Task<String> printContentType = contentType.andThen("print", s->System.out.println(s));

类似的，如果我们需要处理潜在的错误，我们可以使用 onFailure()方法:

Task<String> logFailure = contentType.onFailure("print stack trace", e -> e.printStackTrace())

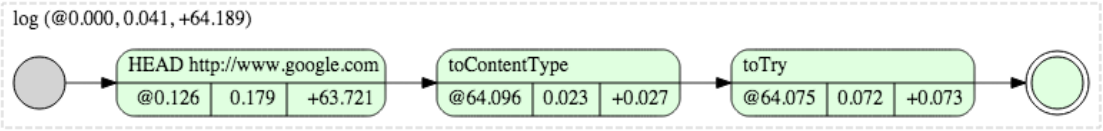


在处理更大潜在错误的时候使用更简单的 toTry()方法更加有用，将 Task<T>转换成 Task<Try<T>>。

Task<Try<String>> contentType = head.map("toContentType", response -> response.getContentType.toTry());

Task<Try<String>> logContentType = contentType.andThen("log",

```
type->{
        if (type.isEnabled()) {
            type.getError().printStackTrace();
        } else {
            System.out.println("ContentType:" + type.get());
        } });
```



上面的例子中，我们使用 JAVA8 的方法引用，当然我们也可以使用 Lambda 表达式来完成：

```
Task<String> printContentType = contentType.andThen("print",
s->System.out.println(s));
```

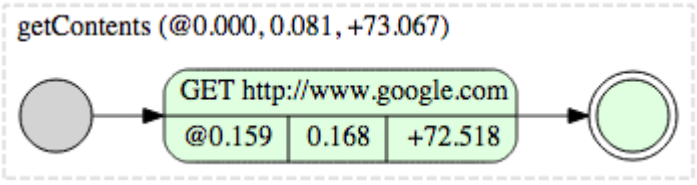
类似的，如果我们需要处理潜在的错误，我们可以使用 onFailure()方法:

```
Task<String> logFailure = contentType.onFailure("print stack trace", e
-> e.printStackTrace());
```

最后，transform()方法将组合 toTry()和 map();

```
Task(Response) get = HttpClient.get("http://www.google.com").task();
Task<Optional<String>> contents = get.transform("getContents",
tryGet -> {
        if (tryGet.isFailed()) {
            return Success.of(Optional.empty());
        } else {
```

```
            return
        Success.of(Optional.of(tryGet.get().getResponseBody()));
        }
    });
```



在上面的列子中，如果 HTTP GET 请求失败，"contents" Task 总是成功完成并返回 Optional 或 Optional.empty()包装的谷歌首页内容。

2、组合

很多 Task 是由其他许多串行或并行的 Task 组成。

(1)、并行组合

假如希望异步抓取几个不同页面内容类型组成。首页，我们需要创建一个帮助性的方法来负责从一个 URL 获取内容类型。

```
private Task <String> getContentType(String url) {
    return HttpClient.get(url).task().map("getContentType", response
    -> response.getContentType());
}
```

然后我们可以使用 Tasl.par()方法组合这些任务来异步运行。

```
final Task<String> googleContentType =
getContentType("http://www.google.com");
final Task<String> bingContentType =
```

getContentType("www.bing.com"); final Task<String> contentTypes = Task.par(googleContentType, bingContentType).map("concatenate", (google, bing)-> "Google:" + goolge + "\n" + "Bing:" + bing + "\n");

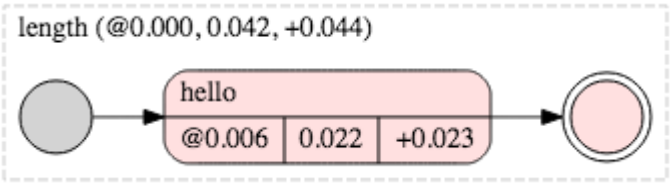
Task.par() 创建一个新的 Task 异步运行 "googleContentType" 和 "bingContentType"。使用 map()方法将执行的结果转换成一个字符串。

六、异常处理和错误恢复

ParSeq 中的一个重要原则：错误总是传播给依赖他们的 Task。通常，这里不需要 Catch 或重新抛出异常。

```
Task<String> failing = Task.callable("hello", () -> {
    return "Hello World".sbustring(100);
 });
Task<Integer> length = failing.map("length", s-> s.length());
```

上面关于 length 的列子会因为 java.lang.StringIndexOutOfBoundsException 失败，并且从 failling Task 中传播出来。

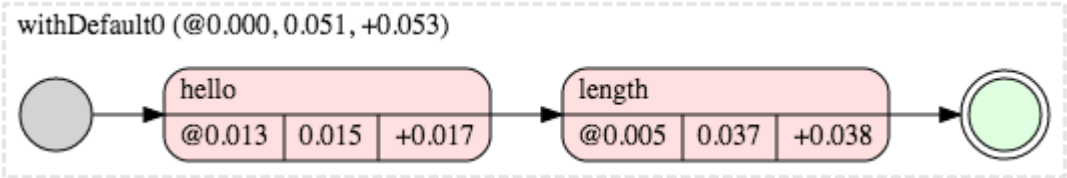


通常降级行为是一个更好的选择相对简单的错误传播。如果存在一个合理错误回滚值，可以使用 recover()从错误中恢复。

```
Task<String> failing = Tasl.callable("hello", ()->{
```

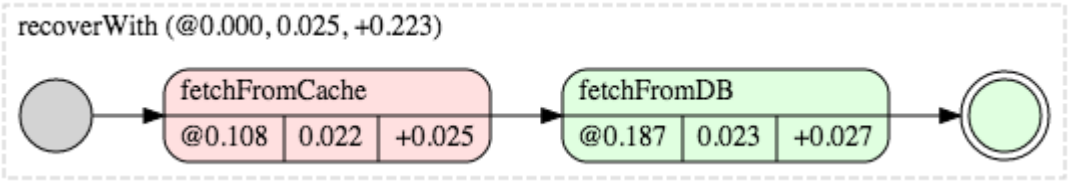
```
        return "Hello World".substring(100);
    });
    Task<Integer> length = failing.map("length", s->length()).recover("withDefault0", e->0);
```

这次 Length Task 将恢复默认值 0 从 java.lang.StringIndexOutOfBoundsException 中恢复。请注意，错误回滚机制允许将导致错误的异常作为一个参数。



有时候我们没有回退值可以使用，但是我们可以使用另外一个 Task 继续完成计算。在这种情况下，我们可以使用 recoverWith()方法。recover()和 recoverWith()方法的区别是后者返回一个包含可退步值将被执行的 Task 实例。下面的例子将演示 ,当我们从缓存中获取用户失败之后从数据库中获取用户信息。

```
Task<Person> user = fetchFromCache(id).recoverWith(e->fetchFromDB(id));
```



七、超时处理

给异步 Task 设置超时时间是一个好的建议，ParSeq 提供了 withTimeout 来完成这项工作。

final Task<Response> google = HttpClient.get("http://google.com").task().withTimeout(10, TimeUnit.MILLISECONDS);

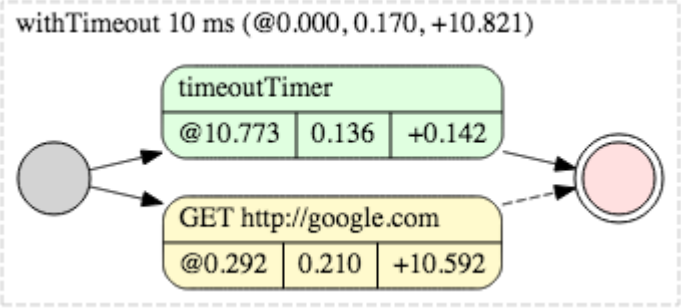
在上面的列子中，如果抓取 google.com 的内容超过 10ms,Task 将会因为 TimeoutException 失败。

八、Task 取消

ParSeq 支持取消 Task。取消一个 Task 意味着导致这个 Task 不再有任何相关性。Task 在任何时候都能够被取消。

Task 实现了当 Task 被取消时能够给侦测到，并且做出相应的反应。通过 CancellationException 完成取消功能，因此，这个行为向一个失败了的 Task 一样将会将取消传递给所有依赖这个 Task 的 Task。虽然取消动作是一个高效的失败，当 Task 被取消之后 recover()，recoverWith()，onFailure()将不能继续呗调用。原因就是，取消 Task 意味着导致一个 Task 无相关，因此不能尝试从这种常见进行错误恢复。使用 cancel()取消一个 Task。 自动取消：通常一个 Task 的目的是通过计算得到一个值。你可以将一个 Task 作为一个异步功能。一旦值被计算出来，就不必继续运行这个 Task。因此任务 TaskParSeq 只运行一次，引擎能够识别已经完成或已经启动的 Task 并且不再执行他们。

在一个 Task 执行完成或开始运行之前已经获取结果值这是可能的。其中一种情况是当我们为一个 Task 设置一个超时时间。指定超时时间的 Task 可能因为 timeoutTime 失败，但是原始的 Task 可能仍然在继续执行。在这种情况下，ParSeq 将通过 EarlyFinishException 来自动取消原始的 Task。



10ms 之后，计算结果的 Task 将会失败，如红色部分；原始的 Task 会通过 EarlyFinishException 自动取消，黄色部分。

九、执行过程跟踪

https://github.com/linkedin/parseq/wiki/Tracing

十、单元测试

ParSeq 提供了一个 test 模块包含一个 BaseEngineTest 可以被用作 ParSeq 相关测试用例的基类。将会自动的创建、关闭执行引擎为每一个测试用例 并且提供很多有用的方法用户执行、跟踪 Task。引入 Maven 依赖：

```
<dependency>
    <groupId>com.linkedin.parseq</groupId>
    <artifactId>parseq</artifactId>
    <version>2.0.0</version>
    <classifier>test</classifier>
    <scope>test</scope>
</dependency>
```

十一、集成 ParSeq

这个部分描述了 ParSeq 已经存在的异步库，我们提供了下面两种例子也许对进一步的指导比较有用。 parseq-http-client 集成了异步的 http client 并且提供了执行异步 HTTP 请求的 Task。 parseq-exec 集成了 JAVA`S Process API 并且提供了异步运行本地程序的 Task。

1、集成异步 API

使用 Task.async()方法创建用于异步完成任务的 Task 实例。接受 Callable 或 Function1 参数，返回一个 Promise 实例。

2、集成阻塞 API

不是每个库都提供异步 API，如 JDBC。我们不应该在 ParSeq 内部直接阻塞代码，因为这样会影响其他异步 Task。我么可以通过 Task.blocking()方法集成阻塞 API。接受两个参数：

Callable：将被执行的代码

Executor：callable 将被调用的实例。

十二、总结

1、 异步并行的定义和使用场景

2、 ForkJoin

3、 京东的异步并行框架 Sirector

异步并发开发框架 Sirector 为 Service Director 的简称，意为服务导演。Sirector 的目标是简化具有复杂依赖关系的任务编排，提高整体任务的执行并发度。

3.1 、引入 Maven 依赖

```
<dependency>
    <groupId>com.jd.sirector</groupId>
    <artifactId>sirector-core</artifactId>
    <version>0.2.2-beta</version>
</dependency>
```

3.2、使用要点

a、异步事件类型的设计，事件在 Sirector 中为范型，既可以是一个 POJO 对象，也可以是一个 Map 对象。开发者应根据自己的需要设计最合适的事件类型。

b、进行整个事务依赖关系分析，找出可以并行的阶段，分别使用事件处理器进行抽象，然后将事件处理器进行编排。

c、非关键的事件处理器应该保证不应抛出异常，任何事件处理器抛出的异

常将会导致在该次事件处理中，还未调用的事件处理器不再被调用。

3.3 使用步骤

一个 Sirector 对应一种事务类型，事务类型描述了事件处理器的先后依赖关系。简单来说，Sirector 使用包括下面的三个步骤：

a、准备事件处理器实例

b、构建编排事件处理器，构建事务类型；

c、发布事件

3.3.1 编排事件处理器

编排事件处理器主要涉及以下方法：

```
Sirector.begin(EventHandler...eventHandlers);
Sirector.after(EventHandler... eventHandlers);
EventHandlerGroup.then(EventHandler... eventHandlers);
Sirector.ready();
```

- Sirector.begin 表示给 Sirector 对应的事务类型中添加没有任何依赖的 EventHandler；
- Sirector.begin 和 Sirector.after 均可以返回 EventHandlerGroup，并接着调用 EventHandlerGroup.then。EventHandlerGroup.then 将会建立 EventHandler 之间的依赖关系。

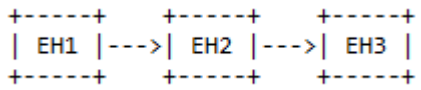
sirector.begin(handler1, handler2).then(handler3);

sirector.after(handler4, handler5).then(handler6).then(handler7);

- Sirector.after 要求参数中 EventHandler 已经添加到事务类型中，否则会抛出异常
- Sirector.begin 和 EventHandlerGroup.then 在 EventHandler 参数实例不存在的时候均会向事务类型中添加 EventHandler。

编排示例 1

下图中的 EH1，EH2，EH3 分别表示三种不同的事件处理器，表示的依赖关系为典型的 Pipeline 类型依赖关系：



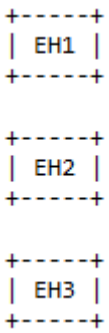
Sirector 相应的事务编排代码如下：

sirector.begin(eh1).then(eh2).then(eh3);

sirector.ready();

编排示例 2

下图中的 EH1，EH2，EH3 分别表示三种不同的事件处理器，下图表示三个事件处理器完全没有任何依赖关系：

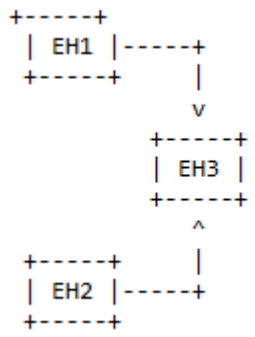


Sirector 相应的事务编排代码如下：

```
sirector.begin(eh1, eh2, eh3);
sirector.ready();
```

编排示例 3

下图中的 EH1，EH2，EH3 分别表示三种不同的事件处理器，下图表示三个事件处理器的依赖关系：

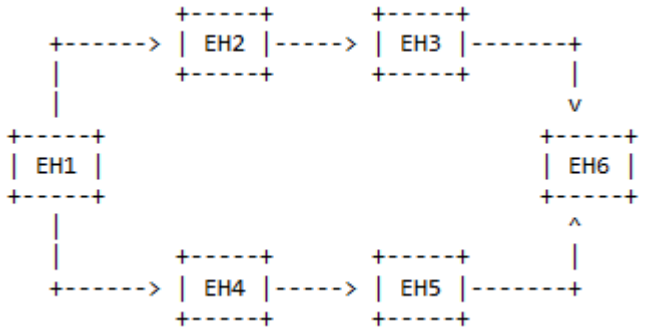


Sirector 相应的事务编排代码如下：

```
sirector.begin(eh1, eh2).then(e3);
sirector.ready();
```

编排示例 4

下图中的 EH1，EH2，EH3，EH4，EH5，EH6 分别表示六种不同的事件处理器，下图表示六个事件处理器之间比较复杂的依赖关系：



Sirector 相应的事务编排代码如下：

```
sirector.begin(eh1).then(eh2, eh4);
sirector.after(eh2).then(eh3);
sirector.after(eh4).then(eh5);
sirector.after(eh3, eh5).then(eh6);
sirector.ready();
```

3.3.4 Sirector 发布事件

Sirector 发布事件有同步和异步两种方式，同步方法会在整个事务完成后，直接返回结果；异步方法则会在事务完成或者抛出异常时进行回调。

同步发布事件方法示例

```
//编排已经完成，现在可以使用 sirector 了
```

```
//构建一个事件，事件的类型我们可以根据业务需要来定义
Event event = new Event(...);
try{
  Event result = sirector.publish(event);
} catch(SirectorException e) {
   //处理异常
}
```

异步发布事件方法示例

```
//编排已经完成，现在可以使用 sirector 了
//构建一个事件，事件的类型我们可以根据业务需要来定义
Event event = new Event(...);
SimpleCallback callback = new SimpleCallback();
sirector.publish(event, callback);
class SimpleCallback implement Callback<Event>{
    public void onSuccess(Event event){
        //整个事件处理已经完成，event 为结果
    }
    public void onError(Event event, Throwable throwable){
       //处理异常
    }
}
```

3.4 示例代码

3.4.1 基本编排示例

```
package com.jd.sirector.example;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import com.jd.sirector.Callback;
import com.jd.sirector.EventHandler;
import com.jd.sirector.Sirector;
@SuppressWarnings("unchecked")
public class SirectorHelloWorld {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();
        Sirector<HelloWorldEvent> sirector = new Sirector<SirectorHelloWorld.HelloWorldEvent>(
                executorService);
                //准备事件处理器实例和回调实例
        HelloWorldEventHandler onceHandler = new HelloWorldEventHandler(1);
        HelloWorldEventHandler twiceHandler = new HelloWorldEventHandler(2);
```

```
        HelloWorldEventHandler threeTimesHandler = new
HelloWorldEventHandler(3);
        HelloWorldEventHandler fourTimesHandler = new
HelloWorldEventHandler(4);
        Callback<HelloWorldEvent> alertCallback = new AlertCallback();
                //编排事件处理器
        sirector.begin(onceHandler).then(twiceHandler);
        sirector.after(onceHandler).then(threeTimesHandler);
        sirector.after(twiceHandler,
threeTimesHandler).then(fourTimesHandler);
        sirector.ready();
                //同步发布事件
        HelloWorldEvent event = sirector.publish(new HelloWorldEvent());
        System.out.println("hello world are called " + event.callCount
                + " times");
                //异步发布事件
        sirector.publish(new HelloWorldEvent(), alertCallback);
    }
    static class HelloWorldEvent {
        private int callCount;
        public void increaseCallCount() {
            callCount++;
```

```
    }
    public int getCallCount() {
        return callCount;
    }
}
static class HelloWorldEventHandler implements
        EventHandler<HelloWorldEvent> {
    private final int times;
    public HelloWorldEventHandler(int times) {
        this.times = times;
    }
    public void onEvent(HelloWorldEvent t) {
        for (int i = 0; i < times; i++) {
            t.increaseCallCount();
        }
    }
}
static class AlertCallback implements Callback<HelloWorldEvent>{
        public void onError(HelloWorldEvent event, Throwable
throwable){
            //处理异常
        }
```

```
        public void onSuccess(HelloWorldEvent event) {
        System.out.println("hello world are called " + event.callCount
                + " times");
        }
       }
}
```

3.4.2 超时示例

```
package com.jd.sirector.example;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import com.jd.sirector.EventHandler;
import com.jd.sirector.Sirector;
import com.jd.sirector.TimeoutException;
@SuppressWarnings("unchecked")
public class SirectorSynTimeout {
    public static void main(String[] args) {
        ExecutorService executorService =
Executors.newCachedThreadPool();
        Sirector<Event> sirector = new
Sirector<SirectorSynTimeout.Event>(executorService);
        SleepHandler handler = new SleepHandler();
```

```
        sirector.begin(handler);
        sirector.ready();
        try{
            sirector.publish(new Event(), 1000/*timeout in millisecond*/);
        }catch (TimeoutException e) {
            /*handle timeout exception*/
            e.printStackTrace();
        }
    }
    static class Event{
    }
    static class SleepHandler implements EventHandler<Event>{
        public void onEvent(Event event) {
            try{
                Thread.sleep(10000);
            }catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

备注：

(1)、源码地址：

http://source.jd.com/app/sirector.git

(2)、文档地址：

http://jpcloud.jd.com/pages/viewpage.action?pageId=753788

4、 Linkedin 的异步并行框架 ParSeq

https://github.com/linkedin/parseq

十三、个人代码库

个人 GitHub 地址：https://github.com/walkerljl，项目介绍：

1、https://github.com/walkerljl/orgwalkerljl-commons.git

常用工具包，包含邮件管理（发送/接收）、AOP、IOC、JMX、IO、Thread、包扫描、配置管理、本地缓存等各种小工具。

2、https://github.com/walkerljl/orgwalkerljl-db.git

ORM 工具，常用数据库连接池适配。

3、https://github.com/walkerljl/orgwalkerljl-boss.git

单点登录、权限管理、MVC 模板、JS、CSS、读写分离集成、Web 开发模板等。

更少的依赖，更强的内聚性；用更少的代码解决问题。

Thanks.

Email：lijunlins@163.com