## 使用Reactive编程模式实现一个Orchestration-based Saga

在一个松散耦合的微服务架构中，理想的情形下，微服务之间是没有相互依赖关系的。因此在一次交易执行的过程中，存在对一些微服务的调用可以并发执行的情形，正好适合多线程中的Fork-Join运行方式；当然，由于业务上的需求，有时又确实存在对调用先后关系的要求。Reactive编程模式中的Parallel、Merge、及Chain的组合使用，可以在某种程序上简化微服务的调用组合。下面我们就以Spring Reactor为例，实现一个Orchestration-based Saga。

众所周知，微服务应用中的Consistency strategies有五种——

|  |  |  |
| --- | --- | --- |
| # | Name | Strategy |
| 1 | Compensating action | Perform an action that undoes prior action(s) |
| 2 | Retry | Retry until success or timeout |
| 3 | Ignore | Do nothing in the event of errors |
| 4 | Restart | Reset to the original state and start again |
| 5 | Tentative operation | Perform a tentative operation and confirm (or cancel) later |

在我们的程序中，使用strategy #1——Compensating action；并且，考虑到银行应用对数据一致性的要求，一般会在有需要的微服务、或一个微服务中的某些API调用中，增加应用锁机制。如此，一次交易执行可划分为两个阶段：

* 第一阶段由Command Handler向各微服务提交交易请求，微服务在执行交易的同时加应用锁；
* 第二阶段，如果各个微服务都返回交易成功信息，Command Handler则对有应用锁的微服务发出Commit请求，微服务收到请求后解除应用锁；否则，Command Handler则对全部微服务发出compensating请求（有可能交易虽被微服务正确执行、但其并未成功返回结果，故需要对全部微服务发出请求），其后微服务通过补偿交易回滚之前的Submit。

需要注意的是，在整个交互过程中，Command Handler应在Event Store中记下与各个微服务之间的所有交互过程。这一做法源自下面的目的：

* 保证Command Handler本身的弹性扩展。
* 若在Compensating/commit阶段发生意外错误，后续可根据Event Store中的记录进行自动或手工修正。
* 此外，Event Store也可以作为CQRS模式中查询数据的初始来源。比如，用Change Data Capture复制、并发布到Kafka，经加工，最终写入查询数据库。当然，我们也可以让Command Handler直接将所需要的数据发布到Kafka或其他Event Bus上，从而摒弃对CDC的需求。

简而言之，最终的架构将与下图类似——



以下，我们对程序的主要部分进行一些解释：

1. 首先需要对本次交易中所涉及的微服务进行定义，由于在本例中的微服务皆以REST API方式发布服务，所以对微服务的定义主要为各个URL的组成。我们将微服务的定义、及交易数据，以JSON格式传入Command Handler之postTransaction服务。比如：

[

{

"id": 1,

"baseUrl": "http://localhost:8081/balance",

"trxUrl": "/{id}/{amount}/{transactionId}",

"compensatingUrl": "/{id}/{transactionId}",

"holdFlag": true,

"commitUrl": "/{id}",

"retrieveUrl": "/{id}",

"debitCreditFlag": false,

"amount": 98.7

},

{

"id": 2,

"baseUrl": "http://localhost:8082/balance",

"trxUrl": "/{id}/{amount}/{transactionId}",

"compensatingUrl": "/{id}/{transactionId}",

"holdFlag": true,

"commitUrl": "/{id}",

"retrieveUrl": "/{id}",

"debitCreditFlag": false,

"amount": 100.00

},

{

"id": 3,

"baseUrl": "http://localhost:8083/balance",

"trxUrl": "/{id}/{amount}/{transactionId}",

"compensatingUrl": "/{id}/{transactionId}",

"holdFlag": true,

"commitUrl": "/{id}",

"retrieveUrl": "/{id}",

"debitCreditFlag": true,

"amount": 198.7

}

]

该内容代表分别从微服务1、2扣减98.7、100，并将198.7存入微服务。Service的具体定义请参见Service.java。

1. 以UUID方式生成Transaction ID，并将交易的开始步骤写入Store Event:

// Before post, write transaction info into event store

String eventId = Generators.timeBasedGenerator().generate().toString();

Event event = new Event(eventId, "Post transaction with id " + id, "start");

eventRepository.save(event).subscribe();

1. 生成针对各个微服务的Submit调用：

// Build Web API call of submits

List<Mono<Entry>> entries = new ArrayList();

Flux.fromArray(services)

.parallel()

.subscribe(stp -> {

float trxAmount = stp.getDebitCreditFlag() ? stp.getAmount() : -stp.getAmount();

Mono<Entry> entry = WebClient.create()

.put()

.uri(stp.getBaseUrl() + stp.getTrxUrl(), id, trxAmount, eventId)

.accept(MediaType.APPLICATION\_JSON)

.retrieve()

.onStatus(HttpStatus::is4xxClientError, clientResponse ->

Mono.error(new ResourceLockedException()))

.bodyToMono(Entry.class)

.log()

.doOnSuccess(p -> {

logger.debug("Call " + stp.getTrxUrl() + " completed with response " + p.toString());

stp.setSubmitStatus("succeed");

//save step status into event store

String stepId = Generators.timeBasedGenerator().generate().toString();

Step step = new Step(stepId, eventId, stp.getId(), "submit", "succeed");

stepRepository.save(step).subscribe();

})

.doOnError(ex -> {

logger.debug("Call " + stp.getTrxUrl() + " completed with error " + ex.getMessage());

stp.setSubmitStatus("fail");

// save step status into event store

String stepId = Generators.timeBasedGenerator().generate().toString();

Step step = new Step(stepId, eventId, stp.getId(), "submit", "fail");

stepRepository.save(step).subscribe();

});

entries.add(entry);

});

1. 使用whenDelayError算子发出各个Submit调用，在所有调用返回结果后，使用then算子chain其后的Commit/Compensating过程。最后，同样使用then算子生成最终的返回结果：

// Merge all API call

return Mono.whenDelayError(entries)

.subscribeOn(Schedulers.parallel())

.onErrorResume(ex -> {

logger.debug("Catch submit zip exception: " + ex.getMessage());

return Mono.empty();

})

.then(Mono.defer(() -> commitOrCompensating(services, id, eventId)))

.then(Mono.defer(() -> saveEventStatus(event, services)));

1. Commit/Compensating的实现见commitOrCompensating(…)，其实现方式与Submit雷同，这里就不解释了。

完整的程序请见https://github.com/DAYUCS/commandhandler/blob/master/src/main/java/com/eximbills/commandhandler/CommandHandlerApplication.java，上面解释的这一段自124行起。

三个被调用的微服务项目也放在了Github上，分别为

<https://github.com/DAYUCS/backservice1>

<https://github.com/DAYUCS/backservice2>

<https://github.com/DAYUCS/backservice3>

backservice1/2/3数据库都使用MySQL。Event Store数据库使用MongoDB。

附，Event Store数据存贮样本：



