основное

4 февраля 2021 г. 13:28

наше приложение для обработки потоков будет читать из одного потока событий и записывать в два потока событий : happy path + failure path

- Он имеет много общего с концепцией трех стандартных потоков Unix

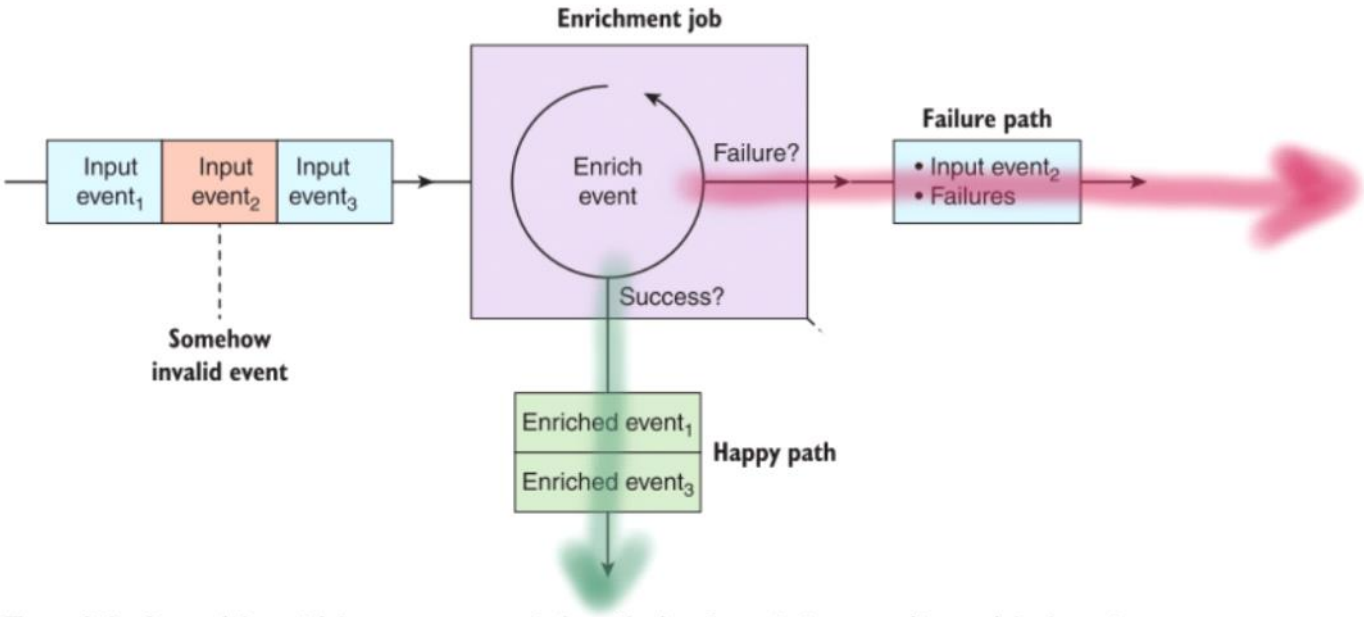

Figure 8.4 Our enrichment job processes events from the input event stream, writes enriched events to our happy path event stream, and writes input events that failed enrichment, plus the reasons why they failed enrichment, to our failure path event stream.

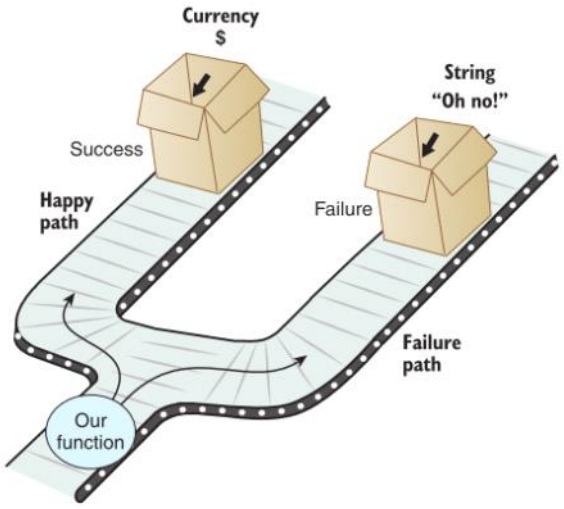

Figure 8.9 For the happy path, our function returns a `Currency` boxed inside a `Success`. For the failure path, our function returns an error `String` boxed inside a `Failure`. `Success` and `Failure` are the two modes of the Scalaz `Validation` type.

We have done away with exit values.

- те теперь коды ошибок ненужны
- The success or failure of a given unit of work is reflected in whether output was written to the happy event stream or the failure event stream.

We have removed any ambiguity in the outputs. (на выходе есть только два потока и сообщение выйдет **ЛИБО** из одного **ЛИБО** из другого)

- Единица работы приводит к выводу либо в счастливый поток, либо в поток ошибок, но ни в то, ни в другое. Три входных события означают всего три выходных события A unit of work results in output either to the happy stream or to the failure stream, never both, nor neither. Three input events mean a total of three output events

We are using the same in-band tools to work with both our successes and our failures (те если используем кафку то и использовать кафку для ошибочных сообщений тоже)

- Сбои будут завершаться в виде хорошо структурированных записей в одном потоке событий; успешные результаты будут отображаться в виде хорошо структурированных записей в другом потоке событий. Failures will end up as well-structured entries in one event stream; successes will end up as well-structured entries in the other event stream.

We should terminate our job only if one of the following occurs:

We encounter an unrecoverable error during the initialization phase of our job.

- we have to move this unit of work into our failure path,
  Затем, как нам справиться с неисправимой, но не неожиданной ошибкой в единице работы?

Здесь нет способа обойтись - мы должны переместить эту единицу работы на наш путь отказа, но при этом соблюдая несколько важных правил:

- Наш путь неудач не должен выходить за рамки допустимого. Нам не нужно полагаться на сторонние инструменты ведения журнала; мы реализуем единый журнал, так что давайте его использовать! (те если используем кафку то и использовать кафку для ошибочных сообщений тоже)
- Записи в нашем пути отказа должны содержать причину или причины отказа в хорошо структурированной форме, которую могут прочитать как люди, так и машины. (те в отличие от обычного сообщения здесь должно быть подробное описание ошибки)
- Записи в нашем пути отказа должны содержать исходные входные данные (например, обработанное событие), чтобы единица работы потенциально могла быть воспроизведена, если и когда основная проблема может быть исправлена (должны быть все данные из входных сообщения чтобы unit of work можно было повторно воспроизвести работу над входным сообщением)
    - события не удалось обработать с неисправимой ошибкой один раз. Что заставляет нас думать, что эту ошибку можно исправить в будущем? На самом деле, есть много причин, по которым мы пока не должны отбрасывать это неудачное событие:

We encounter a novel error while processing a unit of work inside our job.

- Новая ошибка означает ошибку, которую мы раньше не видели: неизвестное неизвестное в стиле Рамсфельда.
- Хотя в этом случае может возникнуть соблазн продолжить обработку, чтобы свести к минимуму перебои, прекращение работы заставляет нас оценивать любую новую ошибку, как только мы с ней сталкиваемся.
- Затем мы можем определить, как эту новую ошибку следует обрабатывать в будущем - например, можно ли исправить ее, не нарушая единицу работы?

failure event

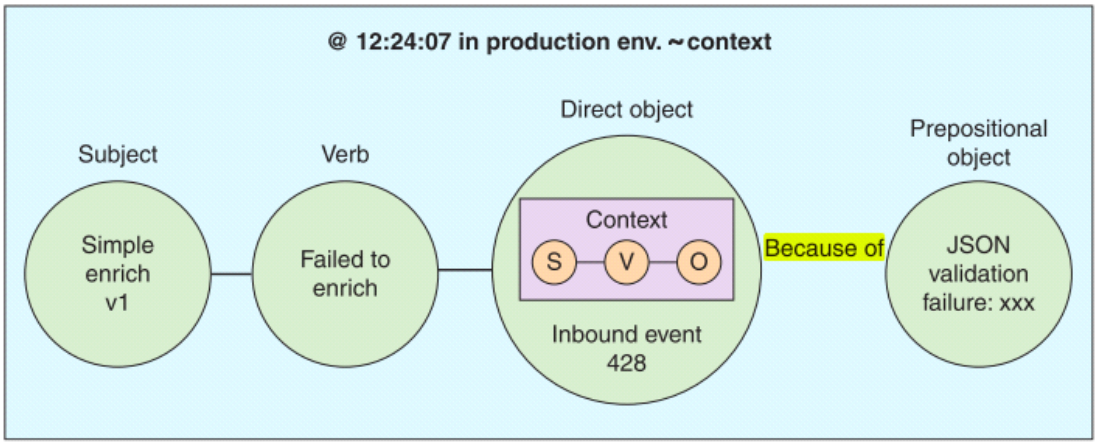

Figure 8.5 We are representing our enrichment as an event: the subject is the enrichment job itself, the verb is "failed to enrich," and the direct object is the event we failed to enrich.

Оно содержит все те же грамматические компоненты, что и события,

- *Subject*—In this case, our stream processing job, SimpleEnrich v1, is the entity carrying out the action of this event.
- *Verb*—The action being done by the *subject* is, in this case, "failed to enrich."
- *Direct object*—The entity to which the action is being done is Inbound Event 428.
- *Timestamp*—This tells us exactly when this failure occurred.

context - У нас есть еще одна часть контекста, помимо отметки времени: среда, в которой произошел сбой.

у нас также есть prepositional object- а именно, причина неудачного обогащения

- We call this prepositional because it is associated with the event via a prepositional phrase—in this case, "because of."

повторная обработка событий

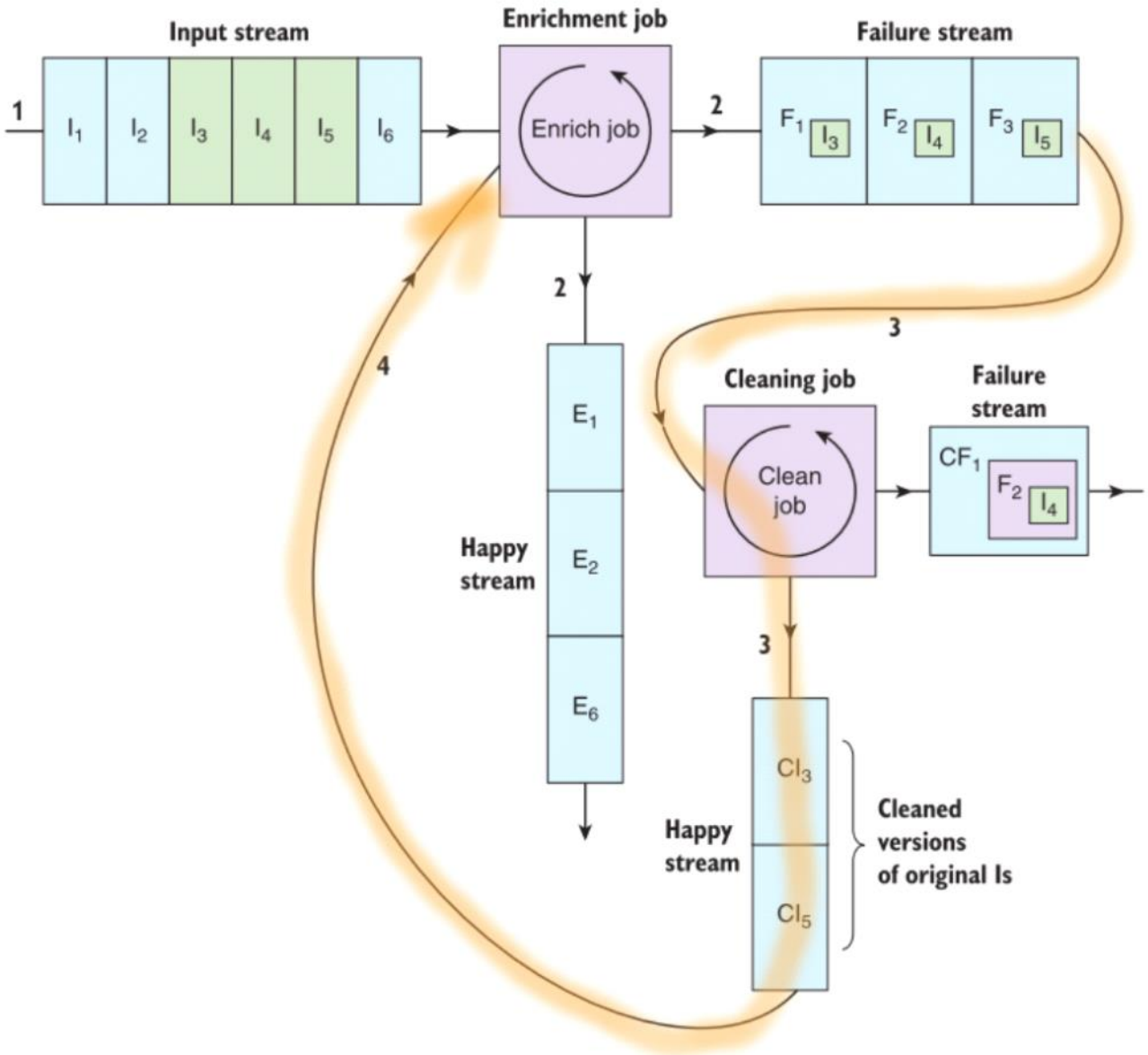

Figure 8.6 Our enrichment job reads six input events; three fail enrichment and are written out to our failure stream. We then feed those three failures into a cleaning job that attempts to fix the corrupted events. The job manages to fix two events, which are fed back into the original enrichment job.

railway-oriented processing

4 февраля 2021 г. 18:37

- This is not my own metaphor: railway-oriented programming was coined by functional programmer Scott Wlaschin in his eponymous blog post (https://fsharpforfunandprofit.com/posts/recipe-part2/). Scott's blog post uses the railway metaphor with happy and failure paths to introduce compositional failure handling in the F# programming language.

две отдельные версии happy path

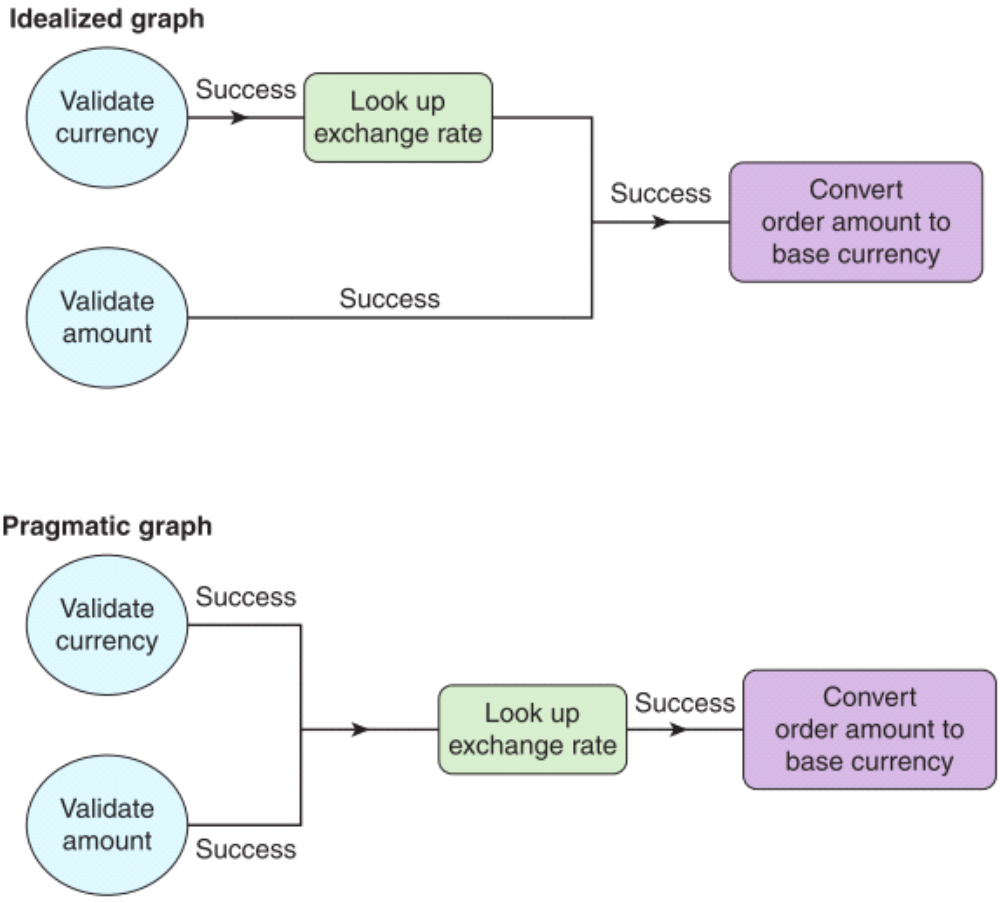

Figure 8.11 The idealized and pragmatic happy paths vary based on how late the exchange rate lookup is performed. In the pragmatic happy path, we validate as much as we can before attempting the exchange rate lookup.

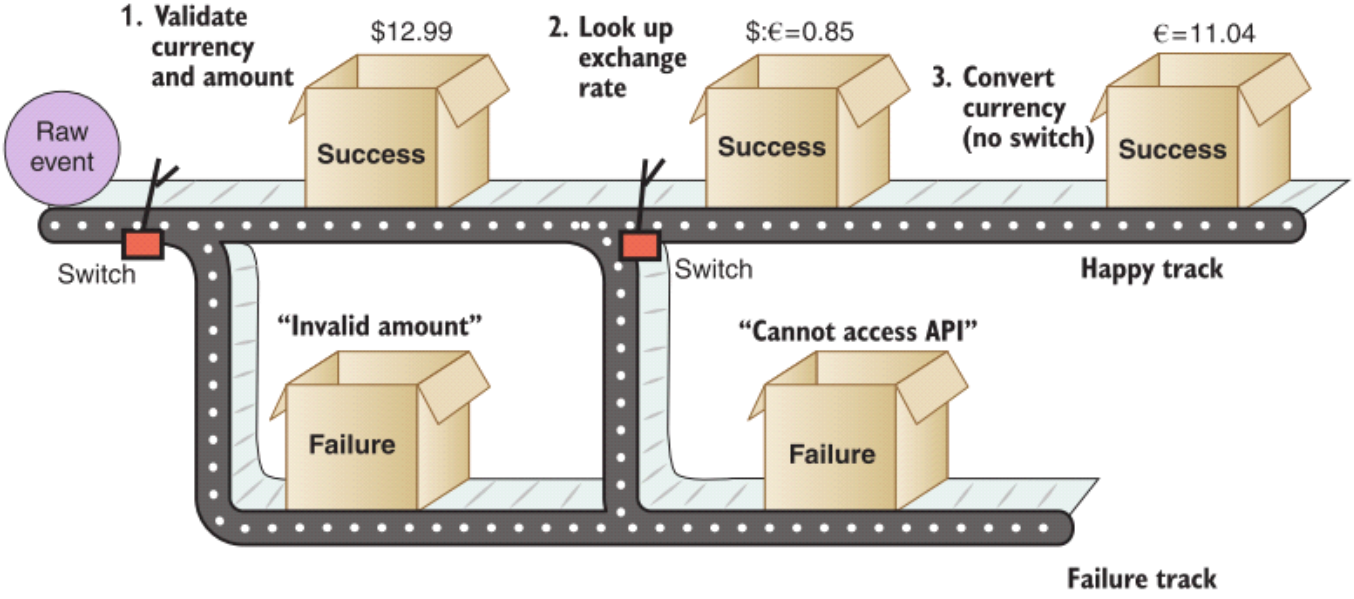

Figure 8.12 Processing of our raw event proceeds down the happy track, but a failure at any given step will switch us onto the failure track. When we are on the failure track, we stay there, bypassing any further happy-track processing.

idealized happy path

- This expresses the dependencies between each piece of work

pragmatic happy path

- Прагматический счастливый путь называется так, потому что поиск валюты из API сторонней службы через HTTP - дорогостоящая операция, тогда как проверка суммы нашего заказа относительно дешева. Мы могли бы выполнять эту обработку для многих миллионов событий, поэтому даже небольшие ненужные задержки будут складываться; поэтому, если мы можем быстро потерпеть неудачу и сэкономить несколько бессмысленных поисков валюты, мы должны это сделать

свойства railway-oriented processing

It composes failures within an individual processing step, but it fails fast across multiple steps.

- This makes sense: if we have an invalid currency code, we can validate the order amount as well (same step), but we can't proceed to getting an exchange rate for the corrupted code (the next step).

The type inside our Success can change between each processing step.

- In figure 8.12, our Success box contains first an OrderTotal, then an exchange rate, and then finally an OrderTotal again.

The type inside our Failure must stay the same between each processing step.

- Therefore, we have to choose a type to record our failures and stick to it. We have used a NonEmptyList[String] in this chapter because it is simple and flexible.

паттерны

Composition in the large

- —We compose complex event-stream-processing workflows out of multiple stream-processing jobs, each of which outputs a happy stream and a failure stream.

Fail-fast as the filling or patty

- —If a stream processing job consists of multiple work steps that have a dependency chain between them, we must fail fast as soon as we encounter a step that did not succeed. Scala's flatMap and map help us to do this

Composition in the small

- —If we have a work step inside our job that contains multiple independent tasks, we can perform all of these and then compose these into a final verdict on whether the step was a Success or a

Failure. Scalaz's Scream operator, |@|, helps us to do this