The behaviours and responsibilities of different types of streams

While you don't need to know much about the internals of Promistreams to use the libraries, there are a few things that are useful to know, mostly around which streams are responsible for what. In the Promistreams design, much of the behaviour is 'emergent'; it's not enforced by some central runtime or orchestrator, but rather is the emergent result of different parts of the system behaving in certain defined ways.

For example, you might think that the `pipe` function does error handling, but it doesn't! All of the error handling is emergent from the design, and simply a result of how Promises work - a pipeline is essentially just a very long chain of nested Promise callbacks, internally. All that `pipe` does is a bit of `bind` magic to pass the previous stream into the next one.

However, some things *do* need to be defined to make things like error handling and concurrency work correctly. The decision was made to shift this burden to the source and sink streams, as these are the least likely to require a custom implementation - the result is that a transform stream is not much more than an `async` function, and does not need to care about error handling at all if it doesn't want to act on those errors. Errors will simply propagate through them with the usual throw/rejection mechanisms of Promises.

The source and sink streams need to do a bit more; they are responsible for emitting 'markers' and handling rejections, respectively. The 'markers' are `EndOfStream` and `Aborted`, and these are rejected and propagated like an error would be, but they are specially recognized by (some of the) streams inbetween, as well as the sink stream. They're used for teardown code and, in the case of the sink stream, to generate the appropriate 'consumer-facing' error to throw from the pipeline as a whole.

The basic read process looks like this: you call the `read` function on the pipeline, which calls the `read` function on the last stream in it, the **sink stream**. The **sink stream** is responsible for 'driving' the pipeline in some way, though exactly what that looks like will depend on the stream implementation. It is *valid* for a read on the pipeline to only trigger a single upstream read, but that is generally not useful - more typically, the sink stream will start a **read loop**. The stream upstream from it will call `read` on *its* upstream, and so on, recursively, until a value is read from the **source stream**. Any stream inbetween may modify the result, discard values, combine them, read more times, read less times, and so on. Once the **source stream** runs out of values, it will start dispensing `EndOfStream` markers, which will propagate down like an error, and ultimately signal to the **sink stream** that it should stop any read loops.

The basic abort process looks like this: `abort` is called on any stream in the pipeline, that stream calls `abort` on its upstream, which does the same recursively, until it ends up at the **source stream**. The **source stream** internally 'latches' into 'aborted' mode, and starts dispensing `Aborted` markers on subsequent reads, which are thrown/rejected and therefore propagate back *downstream*, until they eventually end up at the **sink stream**, which unpacks the original error stored within the `Aborted` marker and throws it from its `read` call (and therefore the pipeline's `read` call). Subsequent attempts at reading the sink stream will throw the `Aborted` marker itself, so the original error is not duplicated.

(The details are more complicated, and if the `abort` is a *happy* abort, rather than one based on an Error, the same latching occurs but with `EndOfStream` instead of `Aborted`. Further details will be in the spec.)

When any stream in the pipeline throws an error or rejects a Promise in its `read` callback, this propagates downstream like any error would, until it is received by the **sink stream**. It then initiates the abort process described above.

Note that because `Aborted` and `EndOfStream` markers are thrown/rejected, transform streams inbetween the source and the sink do not need to care about them, unless they intend to implement some kind of teardown logic, in which case they can be intercepted and then re-thrown. But normally they propagate like any rejection would in a chain of Promises, because that is essentially what they are!

Revision #1
Created 2024-12-10 23:47:16 UTC by joepie91
Updated 2024-12-10 23:47:29 UTC by joepie91