**EventBridge自研Runtime架构设计及实现**

**背景**

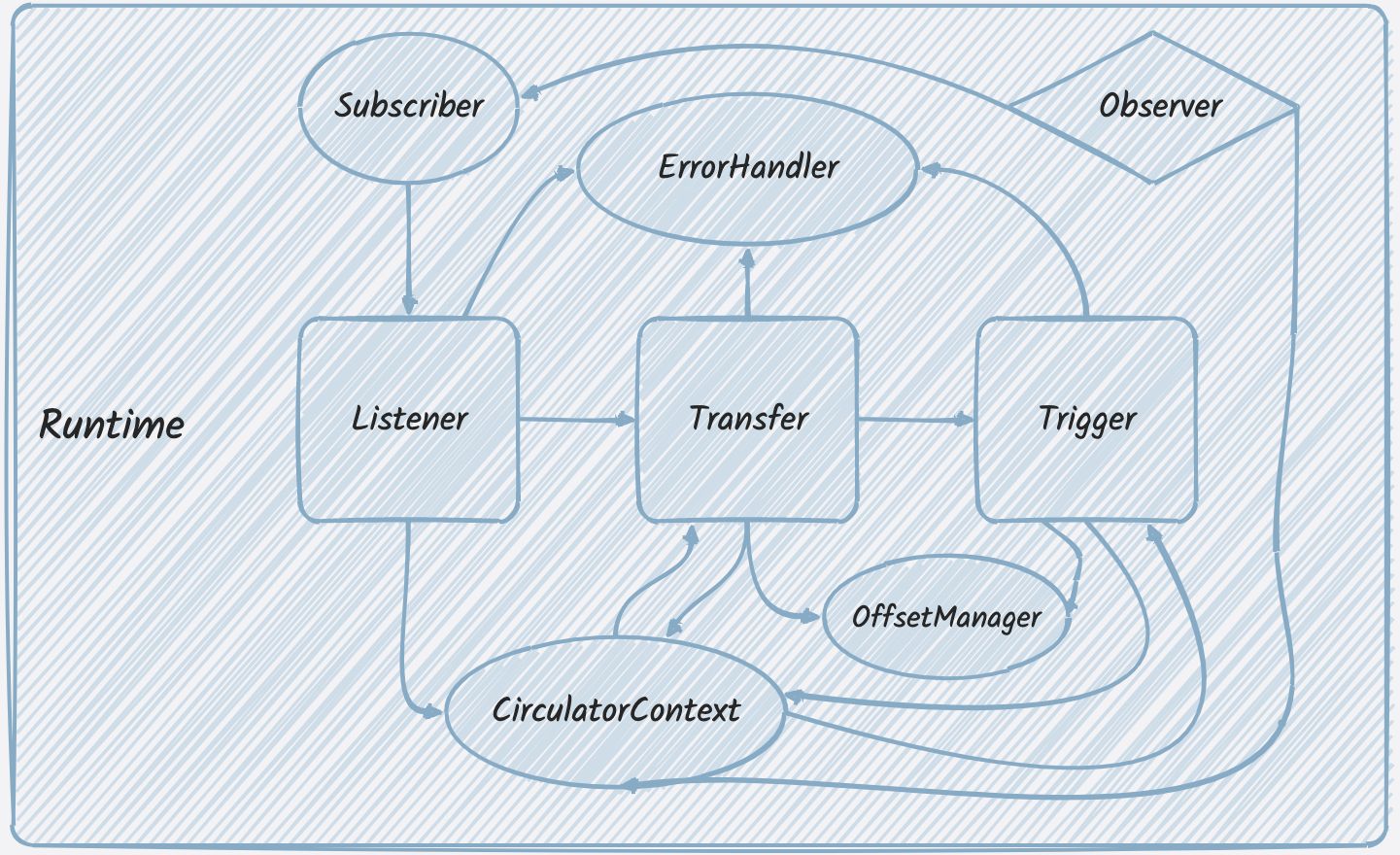
EventBridge做为完整的实现EDA领域的基础服务，其开源第一个版本所依赖的核心运行引擎，为数据流链接工具RocketMQ Connect项目的Runtime运行时。RocketMQ Connect的Runtime虽然具备数据流转的能力，但针对事件的消费吞吐能力、事件目标的触达实时性、运行器的可观测能力等方面，对于开源EventBridge的使用者来说，无疑是存在不足的。针对上述问题，设计一款满足EDA场景下运行引擎Runtime是十分必要的。

**架构设计**

目前开源版本Runtime，核心模块分为：

* EventBusListerner，事件总线监听器（以下简称Listener），批量拉取事件并作本地缓存，供Transfer消费解析。
* EventRuleTransfer，事件规则转换器（以下简称Transfer）， 解析并流转批量事件至指定的事件记录(包含目标信息)，供Trigger拉取并推送。
* EventTargetTrigger，事件目标触发器（以下简称Trigger），将批量事件记录推送至目标端。

架构运行图如下：



围绕Listerner、Transfer、Trigger，有如下关键组件：

1. Subscriber，订阅不同消息存储介质的抽象简称，用于拉取事件消息并作本地缓存，供Listener消费。
2. Observer，为动态监听目标配置(RunnerConfig)的观察者，具体有DbObserver, FileObserver, ControllerObserver。订阅者为，EventSubscriber和CirculatorContext。
3. OffsetManager，为位移管理器，用于提交事件在Transfer和Trigger完成其生命周期，所对应的消息存储介质的位移。
4. CirculatorContext，循环器上下文，用来维护Listener、Transfer、Trigger的所依赖的上下文元数据。
5. ErrorHandler，异常处理器，用于处理事件在不同阶段的消息流转，支持重试推送事件。

**设计原理**

在上面架构设计模块中，我们梳理了下Runtime的核心模块及关键组件。有些同学可能会发问，为什么要这样设计？下面我们从领域建模和技术指标两个维度，浅谈核心模块和关键组件的设计原理。

**领域建模**

在EventBridge中，我们定义了事件流转生命周期的若干个核心概念，有事件源、事件总线、事件规则、事件目标等。除了事件源，仅仅用来标识的是事件发送者的信息，而其他的事件总线、事件规则、事件目标则更多的是和事件消费者深度绑定的。

从一个事件消费者的角度来说，我们需要知道这个事件存储在什么地方、这个事件是基于什么样的转换规则投递出去、以及最后如何让运行引擎和事件的消费者完成链接投递？

基于以上三个问题，我们抽象出来以事件总线、事件规则、事件目标为核心领域的Listener、Transfer、Trigger三个核心模块，三个模块构成一条核心链路完成一个事件在EventBridge服务端的运行流转，而这个核心链路就是Runtime的中枢调度器。

**技术指标**

**事件消费吞吐量**

Subscriber模块实现了不同存储介质对事件的长链接拉去消费，缓存在本地队列中，并以RunnerName做为消费者分组批量拉取事件消息以及批量提交位点。该模块能够确保不同事件消费者的消费隔离，进而保证事件消费的高并发。

在循环器上下文CirculatorListener中，有Listener、Transfer、Trigger所需的元数据，同时也有各模块所消费的事件记录本地缓存，从本地存储角度来说，直接提升了事件流转投递的吞吐性能。

此外，在Transfer和Trigger模块，针对CPU密集型和IO密集型分配不同参数的线程池，可进一步保证事件消费的高性能。

**事件投递实时性**

Listener、Transfer、Trigger为独自运行线程，相互之间的数据绑定依赖于本地的事件记录缓存，而这些缓存数据都存储在统一的循环器上下文CirculatorContext中，可以保证事件记录的实时监听于投递。

其次，循环器上下文CirculatorContext和Subscriber同时是事件目标RunnerConfig的配置内容修改的订阅者，可以实时感知事件目标的配置项修改，从而实时的完成各自元数据的更新。进一步保证事件投递的实效性。

而后，位移管理器OffsetManager，可实时协调Transfer和Trigger的事件记录流转状态，确保事件投递的完整生命周期。

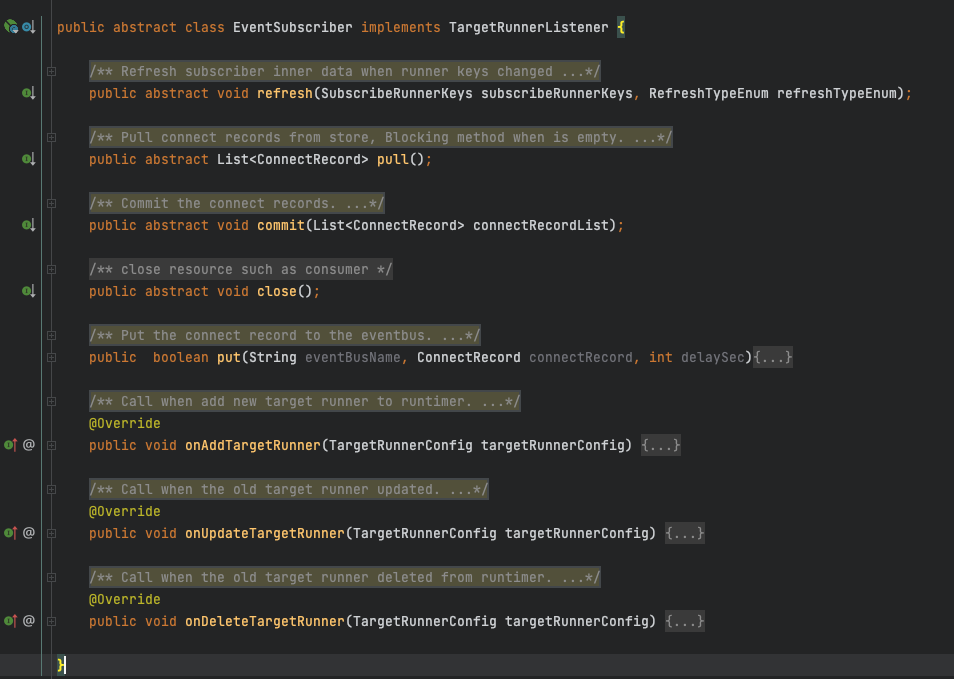
**运行器的可观测**

ErrorHander异常处理器，会处理每个异常投递的事件消息，并做好埋点记录。与此同时，社区方面正在建设Metric基础模块，以此完善EventBridge模块的可观测能力。

**核心模块**

**存储介质**

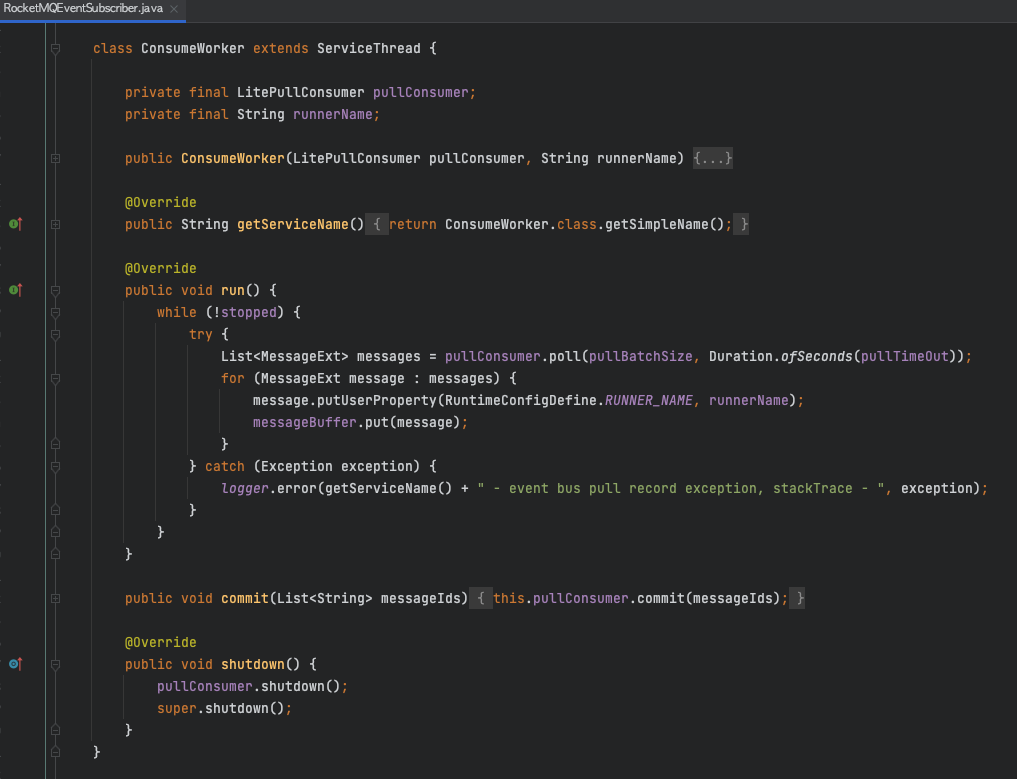
在adapter包下新建storage模块，用来实现消息存储介质的多种策略实现。其中以抽象类EventSubscriber做为基类，可供不同的消息存储介质扩展实现。目前默认实现的是RocketMQEventSubscriber，后续将支持更多的存储介质实现。下图为EventSubscriber抽象类的核心接口，



包含核心四个抽象方法，

* refresh，监听Runner运行器的配置信息，从而动态更新事件Consumer的关键字段。
* pull，拉取事件消息
* commit，事件消息位移提交
* close，关闭指定事件消费者

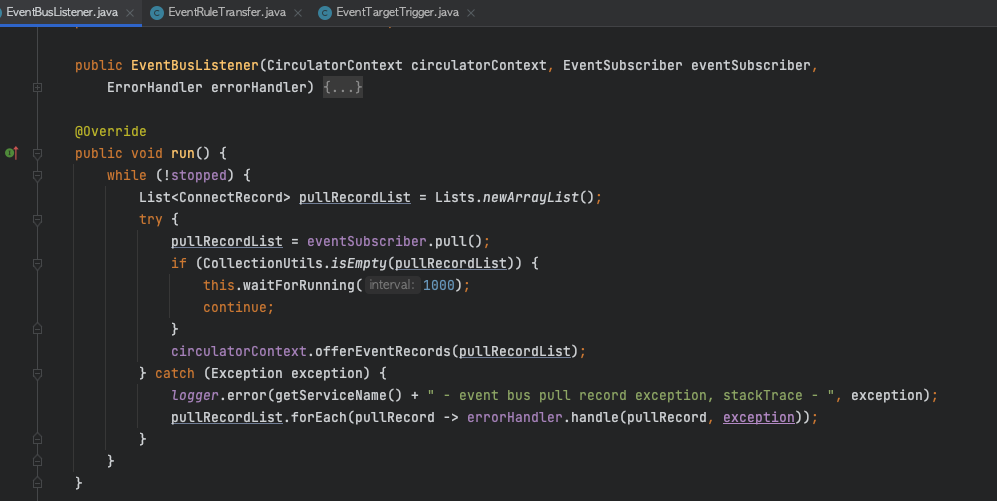
以RoekctMQSubscriber为例，做为Listener的上游入口，提供消息的消费入口并转换为事件记录在本地存储，供下游Runtime模块流转。同时，内置类ConsumeWorker以runnerName做分组，实现不同的目标事件消费者隔离。而做为RunnerConfigObserver的订阅者，方便ConsumeWokrder属性的动态更新。下图是内置类ConsumeWorker的定义及核心实现：



每个ConsumeWorker做为单独线程运行，会接受RocketMQEventSubscriber对其生命周期的管理，对应的入口便是refresh, pull, commit, close四个核心抽象接口，RokectMQEventSubscriber会结合RokectMQ的特性(topic管理、消息分组、位移提交)实现四个接口的特定逻辑。

**监听器**

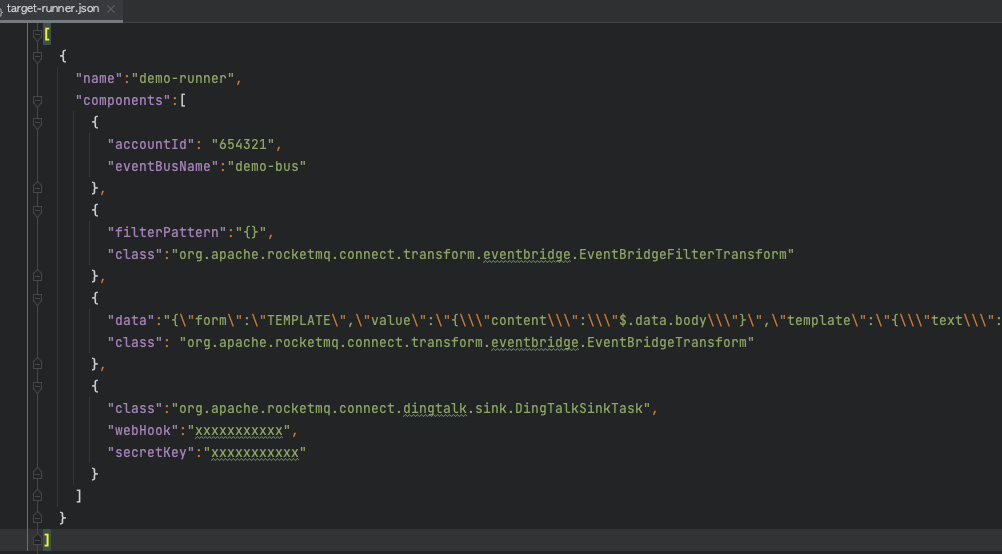
监听器的实现较为简单，主要用来向Subscriber批量拉取本地队列中的事件记录，然后交给循环器上下文暂存，后续由转换器拉取转换。具体实现逻辑如下：



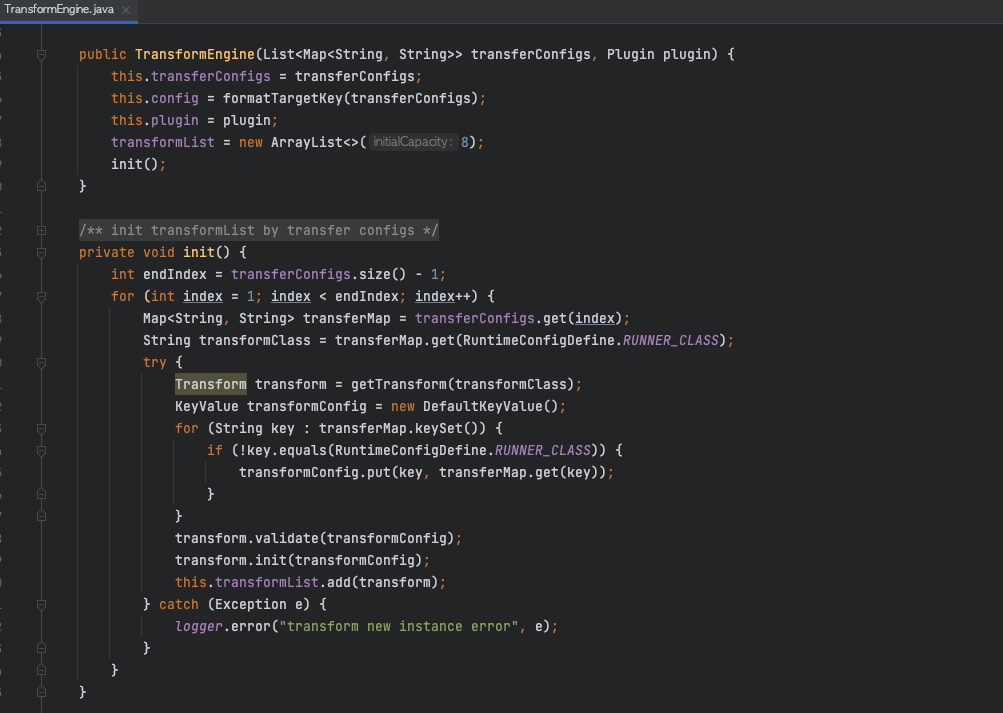
**转换器**

Transfer同样是个独立线程，启动后批量向CirculatorContext拉取事件记录，而后使用TransformEngine对事件进行规则过滤，而后将合法的事件转换为可投递目标的事件记录。这里TransformEngine做为关键组件，用于将关键的Transform插件加载进内存，而后组装不同投递目标的转换列表，供Transfer调用转换事件记录。

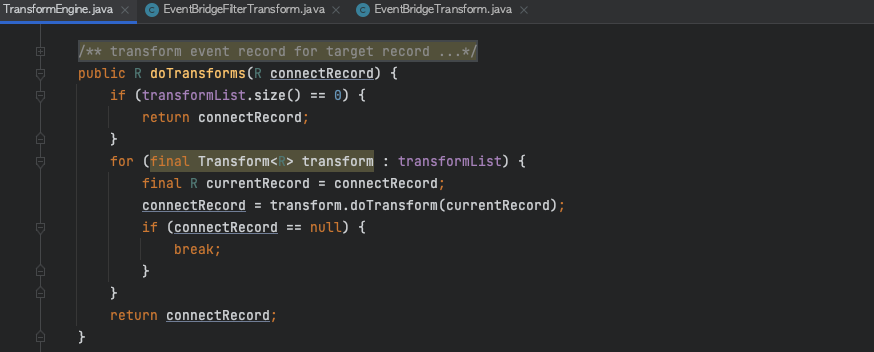
接下来，我们来看下在TransformEngine中是如何完成事件规则解析的。首先，一个标准的RunnerConfig配置信息，可以抽象定义为一个组件的三块内容，分别是：事件总线信息、事件规则转换器信息、以及事件目标触发器。以DingTalk的RunnerConfig为例，其标准JSON格式如下：



而每个RunnerName在分配唯一的TransformEngine初始化时，会将RunnerConfig中组件components信息和EB所需要的插件在构造方法中加载初始化，并构建可用于转换事件记录的转换列表transformList。由DingTalk配置JSON可知，components中间部分是EB目前提供的通用transform插件，分别为EventBridgeFilterTransform和EventBridgeTransform，后续依赖业务场景将适当提供新的transform插件。下图为具体的初始化构建逻辑：



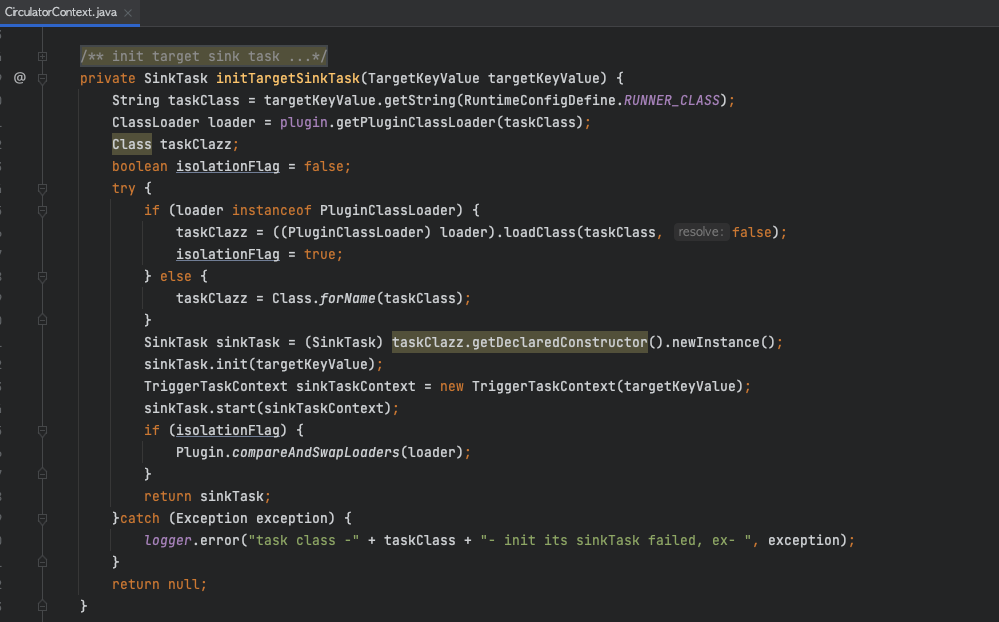
这里需要关注的一点是，TransformEngine在执行doTransform时，则会循环遍历内存中的transform列表，调用实例化的两个transform对象（EventBridgeFilterTransform和EventBridgeTransform）执行解析转换，此类转换过程是个CPU密集型任务，需要关注线程池资源的合理调配。下图为具体转换入口：



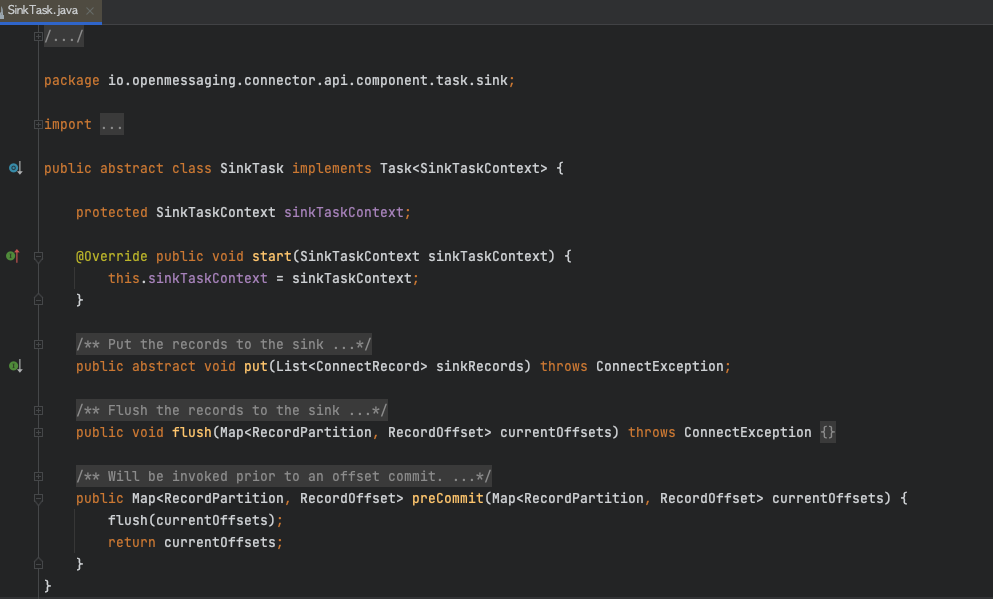
**触发器**

Trigger也是个独立线程，用来指定事件匹配到的SinkTask，然后调用通用接口批量推送至目标端。当然，此处执行推送目标，比如HTTP接口、RPC接口、DingTalk等，都存在网络开销。因此，在多线程资源调配上面，需要针对IO密集型任务，做相应的动态资源调整。

不同的推送目标对应不同的SinkTask，而这些SinkTask则依赖于管控侧创建事件目标时指定的SinkTask实现类。这些SinkTask实现类所在的链接器Connector则以插件的形式在项目启动时被预加载至内存中。不同的事件目标，其配置信息（参考DingTalk的JSON配置文件）会关联唯一的SinkTask。下图为初始化事件目标所关联唯一SinkTask的实现：



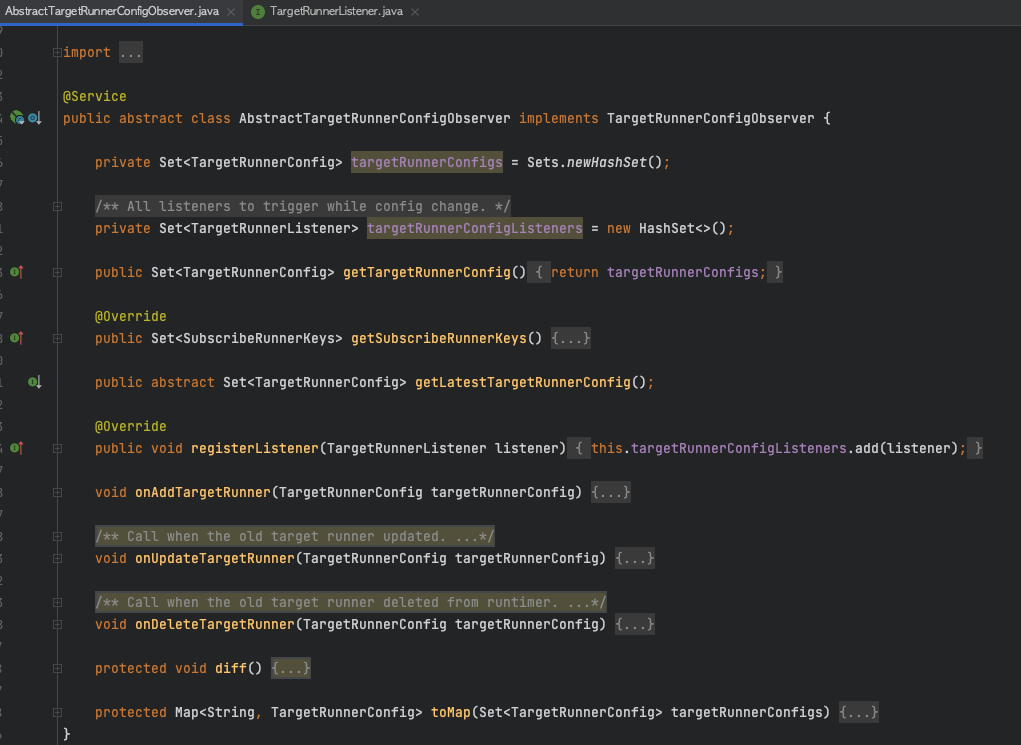
这里要介绍下，SinkTask为分布式消息领域的开发标准OpenMessaging所提供的抽象类，不同的连接器继承该抽象类并实现抽象方法。EB所需要的这些SinkTask所对应的具体连接器Connector，后续会和其目前所在的项目为RocketMQ Connect共建。下图为抽象类SinkTask的数据结构定义：



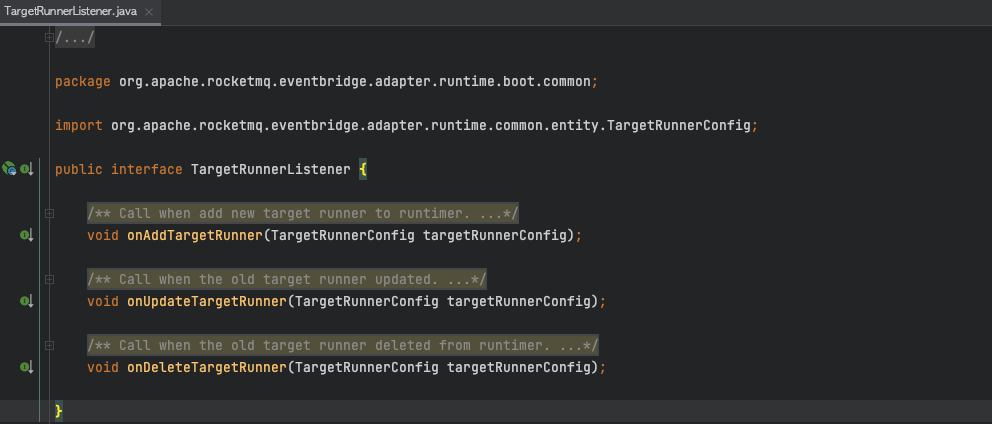
**观察者**

事件目标的配置项，存在修改的场景，需实时同步至Runtime。对于配置项更新所产生的事件动作，这里面有两个入口，一个是事件的发布者，另一个是事件的订阅者。

发布者，目前提供三种模式，一是DbOberserver，实时监听数据库中表event\_target\_runner中的run\_context的修改记录；其次是FileObserver，用于监听本地文件的实时修改；最后是ControllerObserver，用于支持管控端直接对某个配置项的修改触发，待实现。三种模式，可通过配置项选择其中一种使用。发布者的抽象类为AbstractTargetRunnerConfigObserver，核心数据结构如下图所示：

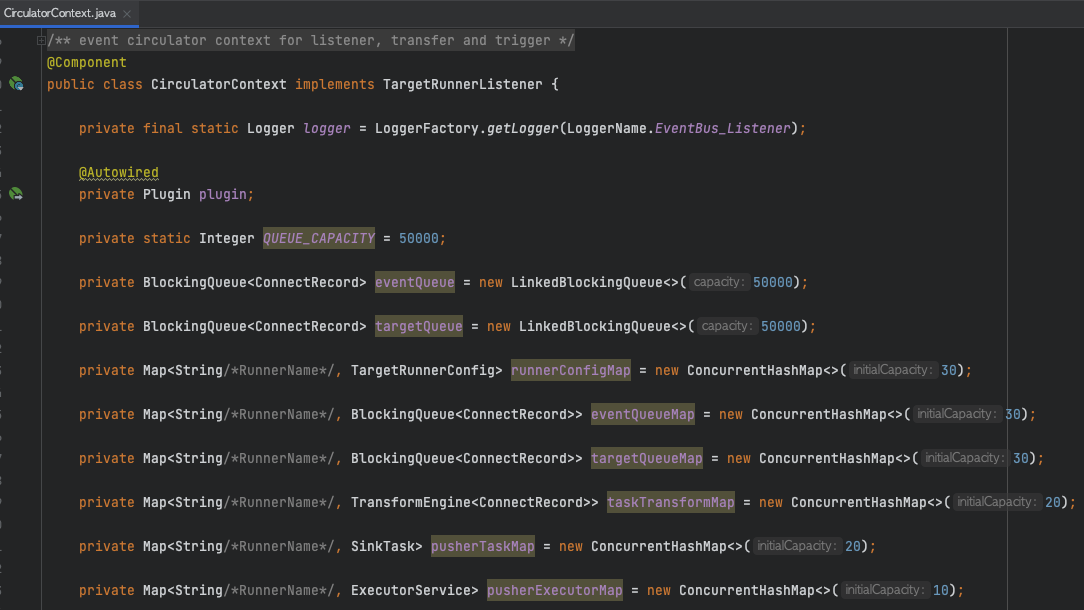


订阅者，配置项的修改，直接影响到的是消息存储介质的订阅主题和Runtime核心三个组件的元数据。因此，EventSubscriber和CirculatorContext，是两个主要的订阅方。订阅者的订阅接口为TargetRunnerListener，接口信息定义如下：

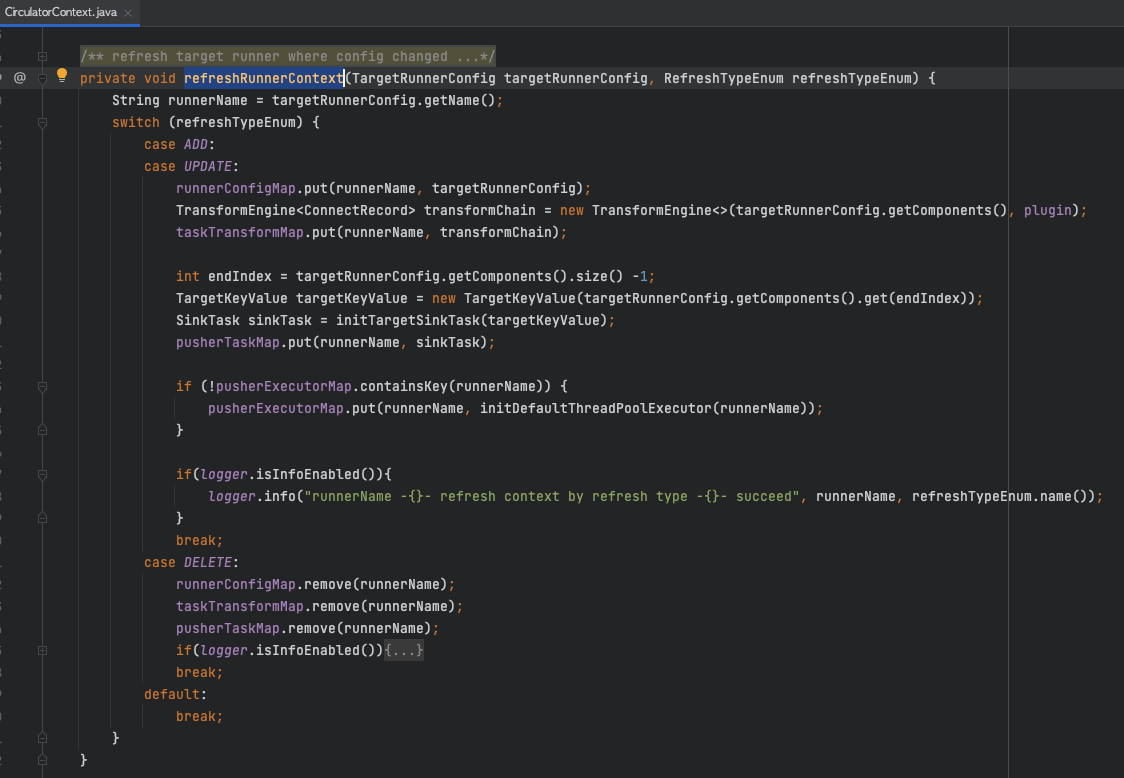


**循环器上下文**

CirculatorContext，做为Runtime核心三个组件的上下文协调器，存储着Listener、Transfer、Trigger三者的关键元数据。比如，和RunnerConfig相关的TransformEngine, SinkTask，以及不同时期的事件记录阻塞队列。核心元数据如下图所示：

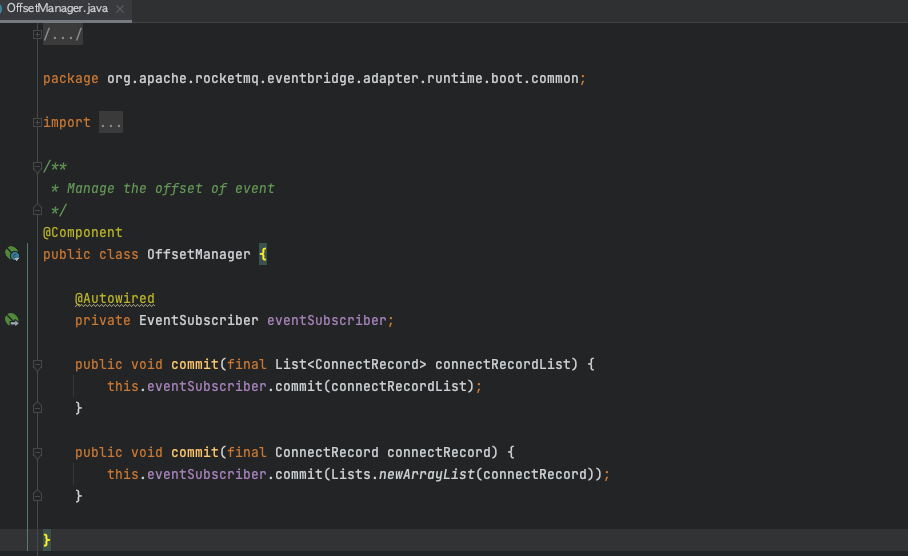


而对于上述元数据，在RunnerConfig信息变更后的数据管理，则由核心方法refreshRunnerContext负责调度整合，核心逻辑如下图所示：



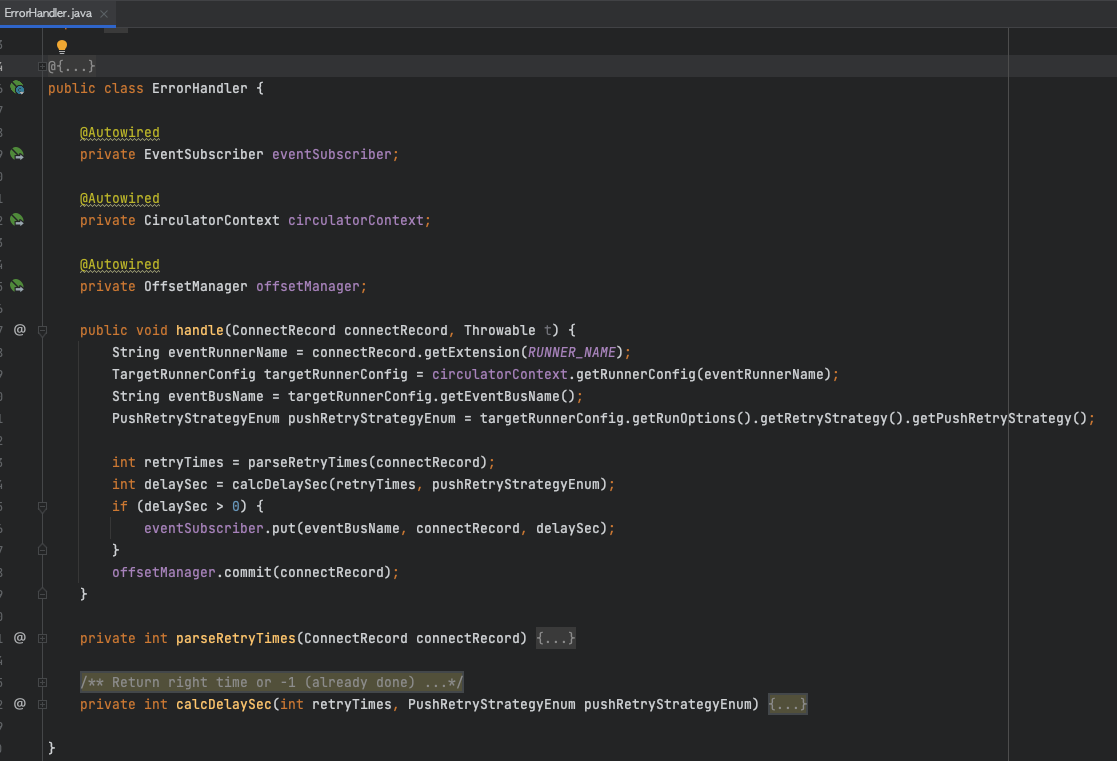
**位移管理**

OffsetManager，为Transfer和Trigger提供事件记录流转中的不同场景下的位移管理，进而结束一条事件消息的生命周期。具体的位移实现，交由所依赖的消息存储介质Subcriber实现（存储介质模块已提及）。下图为位移管理器所提供的两个方法，支持单个或批量提交事件记录。



**异常处理**

ErrorHandler，异常处理器的使用方和CirculatorContext循环器上下文一致，遍布于Listener、Transfer、Trigger三者之中，任何处理单元发生异常，都支持重新投递（默认170次）。在PushRetryStrategyEnum中提供了两种重试策略，重试策略是在RunnerConfig中指定的。异常处理器提供的handler方法，实现逻辑如下：



**关键指标**

**链路吞吐量**

1. 样本时间: 259s
2. 峰值tps：15800
3. 平均tps：8816

**机器指标**

1. OS：MacOS
2. CPU: Intel Core i5 2GHz， 4核
3. Memory: 16G
4. DISK: SSD