尚硅谷大数据技术之 SeaTunnel

（作者：尚硅谷研究院）

版本：V1.0

第 1 章 Seatunnel 概述

1.1 SeaTunnel 是什么

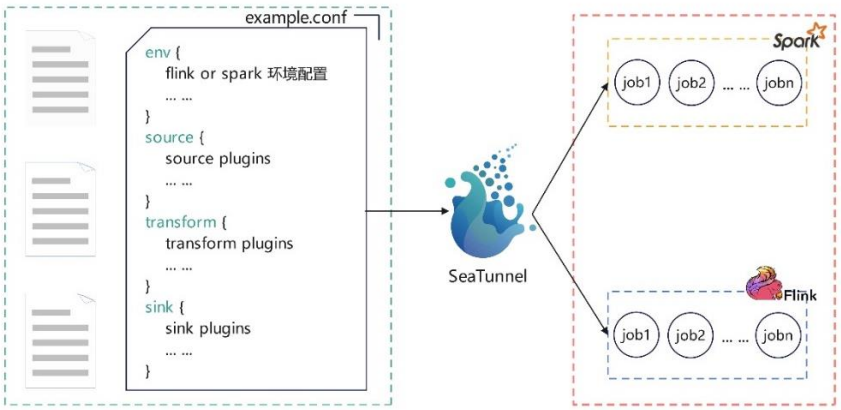
SeaTunnel 是一个简单易用的数据集成框架，在企业中，由于开发时间或开发部门不通用，往往有多个异构的、运行在不同的软硬件平台上的信息系统同时运行。数据集成是把不同来源、格式、特点性质的数据在逻辑上或物理上有机地集中，从而为企业提供全面的数据共享。SeaTunnel 支持海量数据的实时同步。它每天可以稳定高效地同步数百亿数据。并已用于近 100 家公司的生产。

SeaTunnel 的前身是 Waterdrop（中文名：水滴）自 2021 年 10 月 12 日更名为 SeaTunnel。2021 年 12 月 9 日，SeaTunnel 正式通过 Apache 软件基金会的投票决议，以全票通过的优秀表现正式成为 Apache 孵化器项目。2022 年 3 月 18 日社区正式发布了首个 Apache 版本 v2.1.0。

1.2 SeaTunnel 在做什么

本质上，SeaTunnel 不是对 Saprk 和 Flink 的内部修改，而是在 Spark 和 Flink 的基础上做了一层包装。它主要运用了**控制反转**的设计模式，这也是 SeaTunnel 实现的基本思想。

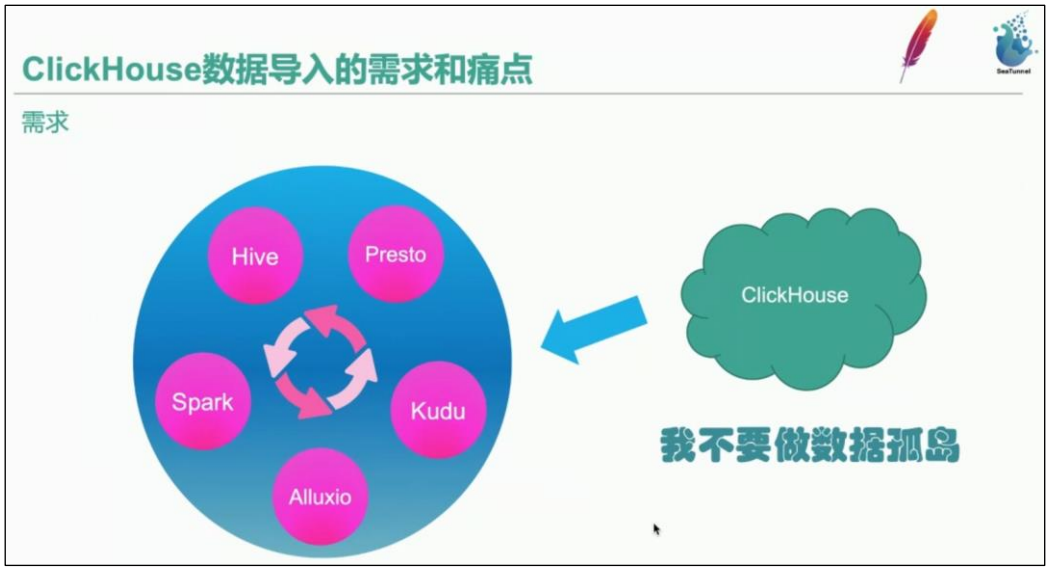
SeaTunnel 的日常使用，就是编辑配置文件。编辑好的配置文件由 SeaTunnel 转换为具体的 Spark 或 Flink 任务。如图所示。



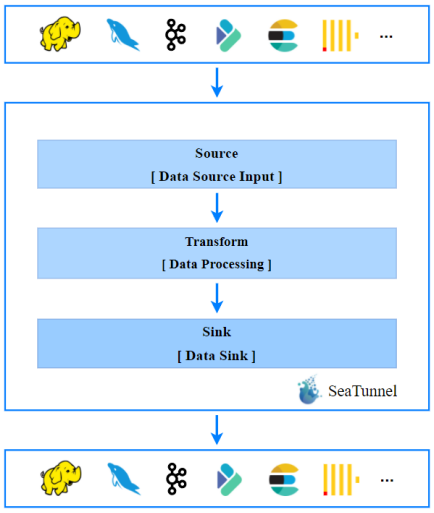
1.3 SeaTunnel 的应用场景

SeaTunnel 适用于以下场景	SeaTunnel 的特点
● 海量数据的同步 ● 海量数据的集成 ● 海量数据的 ETL ● 海量数据聚合 ● 多源数据处理	● 基于配置的低代码开发，易用性高，方便维护。 ● 支持实时流式传输 ● 离线多源数据分析 ● 高性能、海量数据处理能力 ● 模块化的插件架构，易于扩展 ● 支持用 SQL 进行数据操作和数据聚合 ● 支持 Spark structured streaming ● 支持 Spark 2.x

目前 SeaTunnel 的长板是他有丰富的连接器，又因为它以 Spark 和 Flink 为引擎。所以可以很好地进行分布式的海量数据同步。通常 SeaTunnel 会被用来做出仓入仓工具，或者被用来进行数据集成。比如，唯品会就选择用 SeaTunnel 来解决数据孤岛问题，让 ClcikHouse 集成到了企业中先前的数据系统之中。如图所示：



下图是 SeaTunnel 的工作流程：



1.5 SeaTunnel 目前的插件支持

1.5.1 Spark 连接器插件(Source)

Spark 连接器插件	数据库类型	Source	Sink
Batch	Fake	√	
	ElasticSearch	√	√
	File	√	√
	Hive	√	√
	Hudi	√	√
	Jdbc	√	√
	MongoDB	√	√
	Neo4j	√	
	Phoenix	√	√
	Redis	√	√
	Tidb	√	√
	Clickhouse		√
	Doris		√
	Email		√
	Hbase	√	√
	Kafka		√
	Console		√
	Kudu	√	√
	Redis	√	√
Stream	FakeStream	√	
	KafkaStream	√	
	SocketSTream`	√	

1.5.2 Flink 连接器插件(Source)

Flink 连接器插件	数据库类型	Source	Sink
	Druid	√	√
	Fake	√	
	File	√	√
	InfluxDb	√	√
	Jdbc	√	√
	Kafka	√	√
	Socket	√	
	Console		√
	Doris		√
	ElasticSearch		√

1.5.3 Spark & Flink 转换插件

转换插件	Spark	Flink
Add		
CheckSum		
Convert		
Date		
Drop		

Grok		
Json	√	
Kv		
Lowercase		
Remove		
Rename		
Repartition		
Replace		
Sample		
Split	√	√
Sql	√	√
Table		
Truncate		
Uppercase		
Uuid		

这部分内容来官网，可以看出社区目前规划了大量的插件，但截至 V2.1.0 可用的 transform 插件的数量还是很少的。同学们有兴趣也可以在业余时间尝试参与开源贡献。

官方网址：https://seatunnel.apache.org/zh-CN/

第 2 章 Seatunnel 安装和使用

注意 v2.1.0 中有少量 bug，要想一次性跑通所有示例程序，需使用我们自己编译的包，可以在资料包里获取。具体如何修改源码，可以参考文档第 5 章。

2.1 SeaTunnel 的环境依赖

截至 SeaTunnel V2.1.0。

SeaTunnel 支持 Spark 2.x（尚不支持 Spark 3.x）。支持 Flink 1.9.0 及其以上的版本。

Java 版本需要>=1.8

我们演示时使用的是 flink 版本是 1.13.0

2.2 SeaTunnel 的下载和安装

1）使用 wget 下载 SeaTunnel，使用-O 参数将文件命名为 seatunnel-2.1.0.tar.gz

```
wget
https://downloads.apache.org/incubator/seatunnel/2.1.0/apache-
seatunnel-incubating-2.1.0-bin.tar.gz -O seatunnel-2.1.0.tar.gz2
```

2）解压下载好的 tar.gz 包

```
tar -zxvf seatunnel-2.1.0.tar.gz -C /opt/module/
```

3）查看解压的目标路径，apache-seatunnel-incubating-2.1.0 的目录就是我们已经安装好的 seatunnel。Incubating 的意思是孵化中。

2.3 SeaTunnel 的依赖环境配置

在 config/目录中有一个 seatunnel-env.sh 脚本。我们可以看一下里面的内容。

```
# Home directory of spark distribution.
SPARK_HOME=${SPARK_HOME:-/opt/spark}
# Home directory of flink distribution.
FLINK_HOME=${FLINK_HOME:-/opt/flink}

# Control whether to print the ascii logo
export SEATUNNEL_PRINT_ASCII_LOGO=true
```

这个脚本中声明了 SPARK_HOME 和 FLINK_HOME 两个路径。默认情况下 seatunnel-env.sh 中的 SPARK_HOME 和 FLINK_HOME 就是系统环境变量中的 SPARK_HOME 和 FLINK_HOME。

在 shell 脚本中:-的意思是如果:-前的内容为空，则替换为后面的。

例如，环境变量中没有 FLINK_HOME。那么 SeaTunnel 运行时会将 FLINK_HOME 设为/opt/flink。

如果你机器上的环境变量 SPARK_HOME 指向了 3.x 的一个版本。但是想用 2.x 的 Spark 来试一下 SeaTunnel。这种情况下，如果你不想改环境变量，那就直接在 seatunnel-env.sh 中将 2.x 的路径赋值给 SPARK_HOME 即可。

比如：

```
# Home directory of spark distribution.
SPARK_HOME=/opt/module/spark-2.4.8

# Home directory of flink distribution.
FLINK_HOME=${FLINK_HOME:-/opt/flink}

# Control whether to print the ascii logo
export SEATUNNEL_PRINT_ASCII_LOGO=true
```

2.4 示例 1: SeaTunnel 快速开始

我们先跑一个官方的 flink 案例。来了解它的基本使用。

1）选择任意路径，创建一个文件。这里我们选择在 SeaTunnel 的 config 路径下创建一个 example01.conf

```
[atguigu@hadoop102 config]$ vim example01.conf
```

2）在文件中编辑如下内容

```
# 配置 Spark 或 Flink 的参数
env {
  # You can set flink configuration here
  execution.parallelism = 1
  #execution.checkpoint.interval = 10000
  #execution.checkpoint.data-uri = "hdfs://hadoop102:9092/checkpoint"
}
```

```
# 在 source 所属的块中配置数据源
source {
    SocketStream{
          host = hadoop102
          result_table_name = "fake"
          field_name = "info"
    }
}
# 在 transform 的块中声明转换插件
transform {
  Split{
    separator = "#"
    fields = ["name","age"]
  }
  sql {
    sql = "select info, split(info) as info_row from fake"
  }
}
# 在 sink 块中声明要输出到哪
sink {
  ConsoleSink {}
}
```

3）开启 flink 集群

```
[atguigu@hadoop102 flink-1.11.6]$ bin/start-cluster.sh
```

4）开启一个 netcat 服务来发送数据

```
[atguigu@hadoop102 ~]$ nc -lk 9999
```

5）使用 SeaTunnel 来提交任务。

在 bin 目录下有以下内容

```
[atguigu@hadoop102 bin]$ ll
total 20
-rwxr-xr-x 1 atguigu atguigu 2661 Feb 18 06:08 start-seatunnel-flink.sh
-rwxr-xr-x 1 atguigu atguigu 5623 Feb 18 06:08 start-seatunnel-spark.sh
-rw-r--r-- 1 atguigu atguigu 2610 Feb 18 06:08 start-seatunnel-sql.sh
drwxr-xr-x 2 atguigu atguigu 4096 Mar 27 13:47 utils
```

start-seatunnel-flink.sh 是用来提交 flink 任务的。start-seatunnel-spark.sh 是用来提交 Spark 任务的。这里我们用 flink 演示。所以使用 start-seatunnel-flink.sh。

用--config 参数指定我们的应用配置文件。

```
bin/start-seatunnel-flink.sh --config config/example01.sh
```

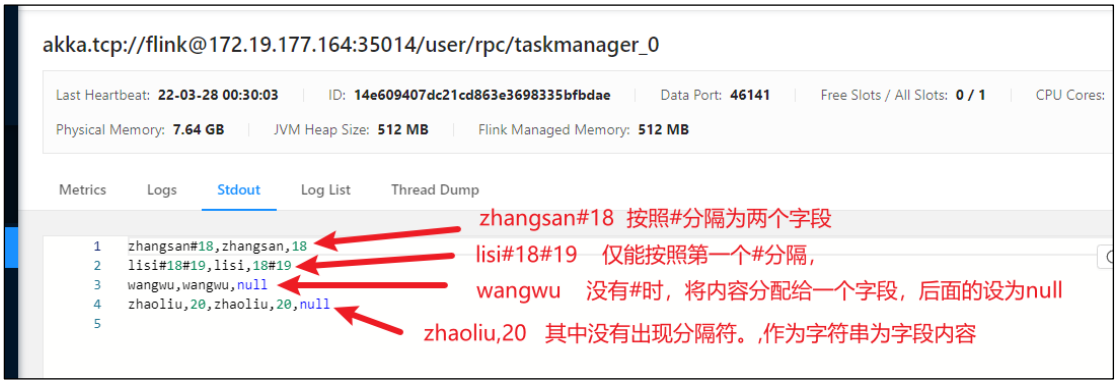
等待弹出 Job 已经提交的提示

```
[atguigu@hadoop102 apache-seatunnel-incubating-2.1.0]$ bin/start-seatunnel-flink.sh --config config/example01.sh
Job has been submitted with JobID bb05ad916cb717b63a50cc3f5f9e1342
```

6）在 netcat 上发送数据

```
[atguigu@hadoop102 ~]$ nc -lk 9999
zhangsan#18
lisi#18#19
wangwu
zhaoliu,20
```

7）在 Flink webUI 上查看输出结果。



8）小结

至此，我们已经跑完了一个官方案例。它以 Socket 为数据源。经过 SQL 的处理，最终输出到控制台。在这个过程中，我们并没有编写具体的 flink 代码，也没有手动去打 jar 包。我们只是将数据的处理流程声明在了一个配置文件中。

在背后，是 SeaTunnel 帮我们把配置文件翻译为具体的 flink 任务。配置化，低代码，易维护是 SeaTunnel 最显著的特点。

第 3 章 SeaTunnel 基本原理

3.1 SeaTunnel 的启动脚本

3.1.1 启动脚本的参数

截至目前，SeaTunnel 有两个启动脚本。

提交 spark 任务用 start-seatunnel-spark.sh。

提交 flink 任务则用 start-seatunnel-flink.sh。

本文档主要是结合 flink 来使用 seatunnel 的，所以用 **start-seatunnel-flink.sh** 来讲解。

start-seatunnle-flink.sh 可以指定 3 个参数

分别是：

--config 应用配置的路径

--variable 应用配置里的变量赋值

--check　检查 config 语法是否合法

start-seatunnel-flink.sh源码

```
function usage() {
  echo "Usage: start-seatunnel-flink.sh [options]"
  echo "  options:"
  echo "    --config, -c FILE_PATH        Config file"
  echo "    --variable, -i PROP=VALUE     Variable substitution,
  echo "    --check, -t                   Check config"
  echo "    --help, -h                    Show this help message
}
```

3.1.2 --check 参数

截至本文档撰写时的 SeaTunnel 版本 v2.1.0。check 功能还尚在开发中，因此--check 参数是一个虚设。目前 start-seatunnel-flink.sh 并不能对应用配置文件的语法合法性进行检查。而且 start-seatunnel-flink.sh 中目前没有对--check 参数的处理逻辑。

需要注意！使用过程中，如果没有使用--check 参数，命令行一闪而过。那就是你的配置文件语法有问题。

3.1.3 --config 参数和--variable 参数

--config 参数用来指定应用配置文件的路径。

--variable 参数可以向配置文件传值。配置文件内是支持声明变量的。然后我们可以通过命令行给配置中的变量赋值。

变量声明语法如下。

```
sql {
    sql = "select * from (select info,split(info) from fake)
where age > '"${age}"'"
  }
```

在配置文件的任何位置都可以声明变量。并用命令行参数`--variable key=value`的方式将变量值传进去，你也可以用它的短命令形式`-i key=value`。传递参数时，key 需要和配置文件中声明的变量名保持一致。

如果需要传递多个参数，那就在命令行里面传递多个-i 或--variable key=value。

比如：

```
bin/start-seatunnel-flink.sh --config/xxx.sh -i age=18 -i sex=man
```

3.1.4 示例 2：配置中使用变量

1）我们在 example01.conf 的基础上创建 example02.conf。

```
[atguigu@hadoop102 config]$ cp example01.sh example02.sh
```

2）修改文件

```
[atguigu@hadoop102 config]$ vim example02.sh
```

3）给 sql 插件声明一个变量，红色的是我们修改的地方。最终的配置文件如下。

```
env {
  execution.parallelism = 1
}

source {
    SocketStream{
          result_table_name = "fake"
          field_name = "info"
    }
}

transform {
  Split{
    separator = "#"
    fields = ["name","age"]
  }

  sql {
    sql = "select * from (select info, split(info) from fake)
where age > '"${age}"'"
    # 需要套一层子查询，因为 where 先于 select,split 出的字段无法用 where 过
滤
  }
}

sink {
  ConsoleSink {}
}
```

4）开启 netcat 服务

```
[atguigu@hadoop102 ~]nc -l 9999
```

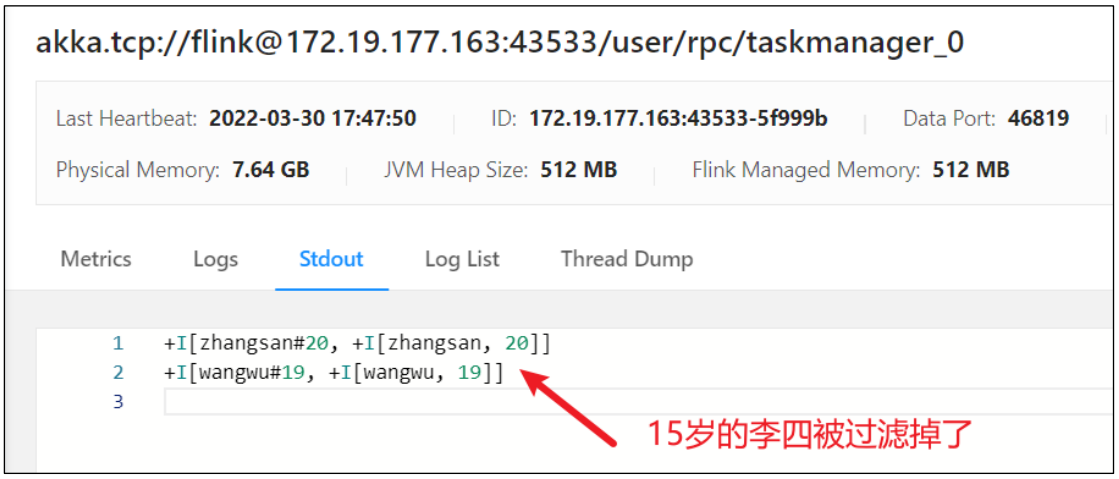
5）使用 SeaTunnel 来提交任务。-i age=18 往命令行中

```
bin/start-seatunnel-flink.sh --config config/example01.sh -i
age=18
```

6）接着，我们用 nc 发送几条数据看看效果。

```
[atguigu@hadoop102 ~]$ nc -l 9999
zhangsan#20
lisi#15
wangwu#19
```

7）在 flink 的 webUI 上我们看一下控制台的输出。最终发现未满 18 岁的李四被过滤掉了。



8）小结

通过传递变量，我们可以实现配置文件的复用。让同一份配置文件就能满足不同的业务需求。

3.1.5 给 flink 传递参数

start-seatunnel-flink.sh源码 文件尾部

```
assemblyJarName=$(find ${PLUGINS_DIR} -name seatunnel-core-flink*.jar)

if [ -f "${CONF_DIR}/seatunnel-env.sh" ]; then
    source ${CONF_DIR}/seatunnel-env.sh
fi

string_trim() {
    echo $1 | awk '{$1=$1;print}'
}

export JVM_ARGS=$(string_trim "${variables_substitution}")

exec ${FLINK_HOME}/bin/flink run \
    ${PARAMS} \
    -c org.apache.seatunnel.SeatunnelFlink \
    ${assemblyJarName} --config ${CONFIG_FILE}
```

在启动脚本的尾部，我们可以看到，start-seatunnel-flink.sh 会执行(exec)一条命令，这个命令会使用 flink 的提交脚本去向集群提交一个任务。而且在调用 bin/flink run 的时候，还传递了 PARAMS 作为 flink run 的参数。

如下图所示，我们可知，凡是--config 和 --variable 之外的命令行参数都被放到 PARAMS 变量中，最后相当于给 flink run 传递了参数。注意！命令行参数解析过程中没有涉及--check 参数处理。这也是为什么说它目前不支持--check 操作。

比如，我们可以在 seatunnel 启动脚本中，指定 flink job 并行度。

```
bin/start-seatunnel-flink.sh --config config/ -p 2\
```

3.2 SeaTunnel 的配置文件

3.2.1 应用配置的 4 个基本组件

我们从 SeaTunnel 的 app 配置文件开始讲起。

一个完整的 SeaTunnel 配置文件应包含四个配置组件。分别是：

env{} source{} --> transform{} --> sink{}

在 Source 和 Sink 数据同构时，如果业务上也不需要对数据进行转换，那么 transform 中的内容可以为空。具体需根据业务情况来定。

3.2.2 env 块

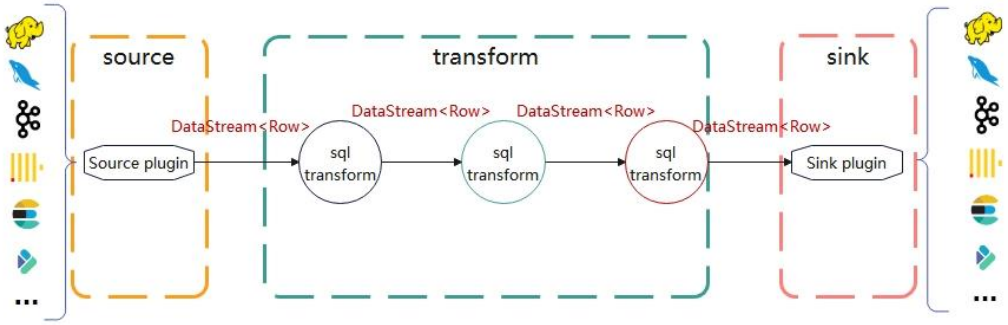
env 块中可以直接写 spark 或 flink 支持的配置项。比如并行度，检查点间隔时间。检查点 hdfs 路径等。在 SeaTunnel 源码的 ConfigKeyName 类中，声明了 env 块中所有可用的 key。如图所示：

```
public class ConfigKeyName {
    public static final String TIME_CHARACTERISTIC = "execution.time-characteristic";
    public static final String BUFFER_TIMEOUT_MILLIS = "execution.buffer.timeout";
    public static final String PARALLELISM = "execution.parallelism";
    public static final String MAX_PARALLELISM = "execution.max-parallelism";
    public static final String CHECKPOINT_INTERVAL = "execution.checkpoint.interval";
    public static final String CHECKPOINT_MODE = "execution.checkpoint.mode";
    public static final String CHECKPOINT_TIMEOUT = "execution.checkpoint.timeout";
    public static final String CHECKPOINT_DATA_URI = "execution.checkpoint.data-uri";
    public static final String MAX_CONCURRENT_CHECKPOINTS = "execution.max-concurrent-checkpoints";
    public static final String CHECKPOINT_CLEANUP_MODE = "execution.checkpoint.cleanup-mode";
    public static final String MIN_PAUSE_BETWEEN_CHECKPOINTS = "execution.checkpoint.min-pause";
    public static final String FAIL_ON_CHECKPOINTING_ERRORS = "execution.checkpoint.fail-on-error";
    public static final String RESTART_STRATEGY = "execution.restart.strategy";
    public static final String RESTART_ATTEMPTS = "execution.restart.attempts";
    public static final String RESTART_DELAY_BETWEEN_ATTEMPTS = "execution.restart.delayBetweenAttempts";
    public static final String RESTART_FAILURE_INTERVAL = "execution.restart.failureInterval";
    public static final String RESTART_FAILURE_RATE = "execution.restart.failureRate";
    public static final String RESTART_DELAY_INTERVAL = "execution.restart.delayInterval";
    public static final String MAX_STATE_RETENTION_TIME = "execution.query.state.max-retention";
    public static final String MIN_STATE_RETENTION_TIME = "execution.query.state.min-retention";
    public static final String STATE_BACKEND = "execution.state.backend";
    public static final String PLANNER = "execution.planner";
```

3.2.3 SeaTunnel 中的核心数据结构 Row

Row 是 SeaTunnel 中数据传递的核心数据结构。对 flink 来说，source 插件需要给下游的转换插件返回一个 DataStream<Row>，转换插件接到上游的 DataStream<Row>进行处理后需要再给下游返回一个 DataStream<Row>。最后 Sink 插件将转换插件处理好的 DataStream<Row>输出到外部的数据系统。

如图所示：



因为 DataStream<Row>可以很方便地和 Table 进行互转，所以将 Row 当作核心数据结构可以让转换插件同时具有使用代码（命令式）和 sql（声明式）处理数据的能力。

3.2.4 source 块

source 块是用来声明数据源的。source 块中可以声明多个连接器。比如：

```
# 伪代码
env {
```

```
    ...
}

source {
  hdfs { ... }
  elasticsearch { ... }
  jdbc {...}
}

transform {
    sql {
     sql = """
        select .... from hdfs_table
        join es_table
        on hdfs_table.uid = es_table.uid where ..."""
    }
}

sink {
    elasticsearch { ... }
}
```

需要注意的是，所有的 source 插件中都可以声明 result_table_name。如果你声明了 result_table_name。SeaTunnel 会将 source 插件输出的 DataStream<Row>转换为 Table 并注册在 Table 环境中。当你指定了 result_table_name，那么你还可以指定 field_name，在注册时，给 Table 重设字段名(后面的案例中会讲解)。

因为不同 source 所需的配置并不一样，所以对 source 连接器的配置最好参考官方的文档。

3.2.5 transform 块

目前社区对插件做了很多规划，但是截至 v2.1.0 版本，可用的插件总共有两个，一个是 Split，另一个是 sql。

transform{}块中可以声明多个转换插件。所有的转换插件都可以使用 source_table_name，和 result_table_name。同样，如果我们声明了 result_table_name，那么我们就能声明 field_name。

我们需要着重了解一下 Split 插件和 sql 插件的实现。但在此

在 SeaTunnel 中，一个转换插件的实现类最重要的逻辑在下述四个方法中。

1）处理批数据，DataSet<Row>进，DataSet<Row>出

```
DataSet<Row> processBatch(FlinkEnvironment env, DataSet<Row>
data)
```

2）处理流数据，DataStram<Row>进，DataStream<Row>出

```
DataStream<Row> processStream(FlinkEnvironment env, DataStream<Row>
dataStream)
```

3）函数名叫注册函数。实际上，这是一个约定，它只不过是每个 transform 插件作用于流之后调用的一个函数。

```
void registerFunction(FlinkEnvironment env, DataStream<Row> datastream)
```

4）处理一些预备工作，通常是用来解析配置。

```
void prepare(FlinkEnvironment prepareEnv)
```

Split 插件的实现

现在我们需要着重看一下 Split 插件的实现。

先回顾一下我们之前 example01.conf 中关于 transform 的配置。

example01.conf中关于transform的配置

```
transform {
  Split{
    separator = "#"
    fields = ["name","age"]
  }

  sql {
    sql = "select info,split(info) as info_row from fake"
  }
}
```

接着我们再来看一下 Split 的源码实现。

split源码，着重看processStream和registerFunction方法

```
@Override
public DataStream<Row> processStream(FlinkEnvironment env, DataStream<Row> dataStream) {
    return dataStream;
}

@Override
public void registerFunction(FlinkEnvironment flinkEnvironment) {
    if (flinkEnvironment.isStreaming()) {
        flinkEnvironment
                .getStreamTableEnvironment()
                .registerFunction( name: "split", new ScalarSplit(rowTypeInfo, num, separator));
    } else {
        flinkEnvironment
                .getBatchTableEnvironment()
                .registerFunction( name: "split", new ScalarSplit(rowTypeInfo, num, separator));
    }
}
```

我们发现 Split 插件并没有对数据流进行任何的处理，而是将它直接 return 了。反之，它向表环境中注册了一个名为 split 的 UDF（用户自定义函数）。而且，函数名是写死的。这意味着，如果你声明了多个 Split，后面的 UDF 还会把前面的覆盖。

这是开发时需要注意的一个点。

example01.conf中关于transform的配置

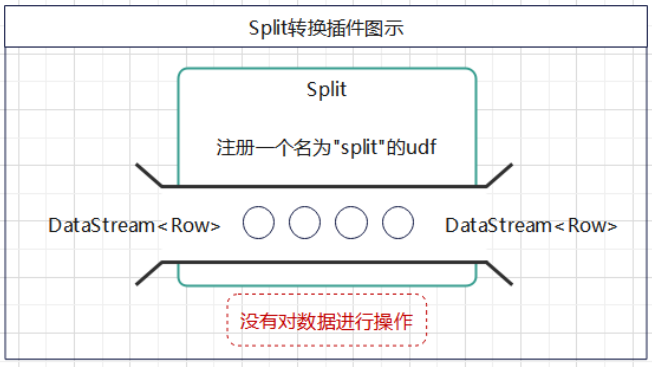
```
transform {
  Split{
    separator = "#"
    fields = ["name","age"]
  }

  sql {
    sql = "select info,split(info) as info_row from fake"
  }
}
```

没有直接作用于数据流
而是注册了一个名为split的udf

后面的sql调用了这个函数

但是，需要注意，tranform 接口其实是留给了我们直接操作数据的能力的。也就是 processStream 方法。那么，一个 transform 插件其实同时履行了 process 和 udf 的职责，这是违反单一职责原则的。那要判断一个转换插件在做什么就只能从源码和文档的方面来加以区分了。



最后需要叮嘱的是，指定 soure_table_name 对于 sql 插件的意义不大。因为 sql 插件可以通过 from 子句来决定从哪个表里抽取数据。

3.2.6 sink 块

Sink 块里可以声明多个 sink 插件，每个 sink 插件都可以指定 source_table_name。不过因为不同 Sink 插件的配置差异较大，所以在实现时建议参考官方文档。

3.3 SeaTunnel 的基本原理

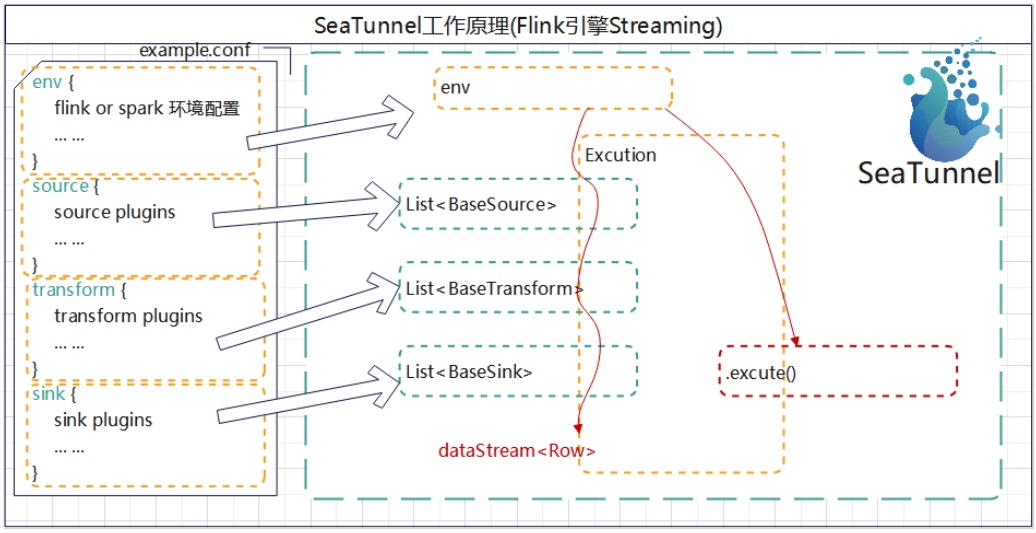
SeaTunnel 的工作原理简单明了。

1）程序会解析你的应用配置，并创建环境

2）配置里 source{}，transform{}，sink{}三个块中的插件最终在程序中以 List 集合的方式存在。

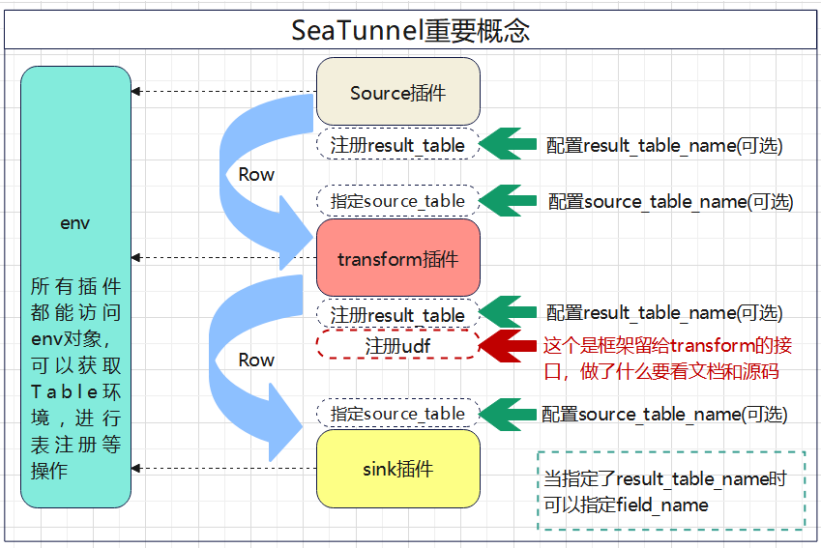
3）由 Excution 对象来拼接各个插件，这涉及到选择 source_table，注册 result_table 等流程，注册 udf 等流程。并最终触发执行

可以参考下图：



3.4 小结

最后我们用一张图将 SeaTunnel 中的重要概念串起来。

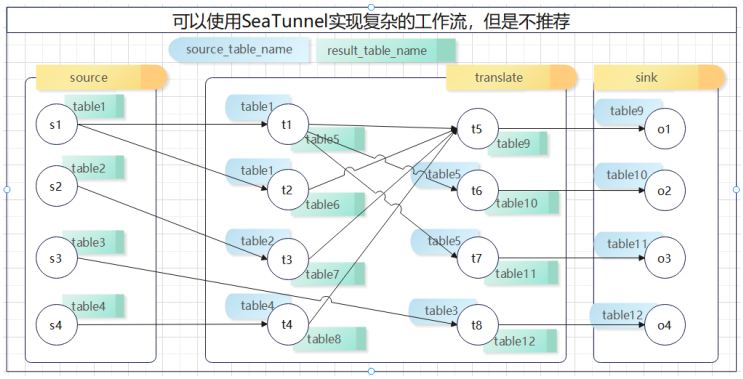


如果你愿意，依托 sql 插件和 udf。单个配置文件也可以定义出比较复杂的工作流。但 SeaTunnel 的定位是一个数据集成平台。核心的功能是依托丰富的连接器进行数据同步，数据处理并不是 SeaTunnel 的长处。所以在 SeaTunnel 中定义复杂的工作流或许是一种不值得提倡的做法。

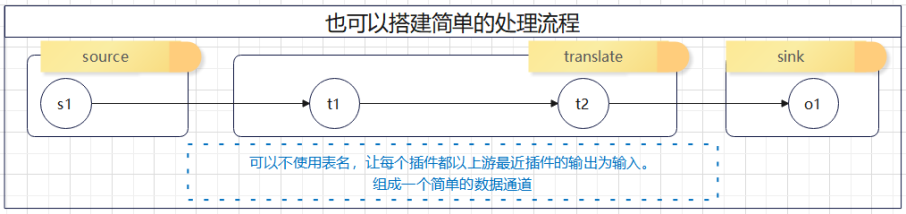
需要提醒的是，如果你不指定 source_table_name，插件会使用它在配置文件上最近的

上一个插件的输出作为输入。

所以，我们可以通过使用依托表名表环境来实现复杂的工作流。



也可以减少表名的使用实现简单的数据同步通道。



第 4 章 应用案例

注意！下述示例请使用我们修改编译好的包。

4.1 Kafka 进 Kafka 出的简单 ETL

4.1.1 需求

对 test_csv 主题中的数据进行过滤，仅保留年龄在 18 岁以上的记录。

4.1.2 需求实现

1）首先，创建为 kafka 创建 test_csv 主题。

```
kafka-topics.sh --bootstrap-server hadoop102:9092 --create --
topic test_csv --partitions 1 --replication-factor 1
```

2）为 kafka 创建 test_sink 主题

```
kafka-topics.sh --bootstrap-server hadoop102:9092 --create --
topic test_sink --partitions 1 --replication-factor 1
```

3）编辑应用配置

```
[atguigu@hadoop102 config]$ vim example03.conf
```

4）应用配置内容

```
env {
  # You can set flink configuration here
  execution.parallelism = 1
  #execution.checkpoint.interval = 10000
```

```
  #execution.checkpoint.data-uri = "hdfs://hadoop102:9092/checkpoint"
}

# 在 source 所属的块中配置数据源
source {
    KafkaTableStream {
        consumer.bootstrap.servers = "hadoop102:9092"
        consumer.group.id = "seatunnel-learn"
        topics = test_csv
        result_table_name = test
        format.type = csv
        schema =
"[{\"field\":\"name\",\"type\":\"string\"},{\"field\":\"age\",
\"type\": \"int\"}]"
        format.field-delimiter = ";"
        format.allow-comments = "true"
        format.ignore-parse-errors = "true"
    }
}
# 在 transform 的块中声明转换插件
transform {

  sql {
    sql = "select name,age from test  where age > '"${age}"'"
  }
}
# 在 sink 块中声明要输出到哪
sink {
   kafkaTable {
    topics = "test_sink"
    producer.bootstrap.servers = "hadoop102:9092"
        }
}
```

5）提交任务

```
bin/start-seatunnel-flink.sh --config config/example03.conf -i
age=18
```

6）起一个 kafka console producer 发送 csv 数据(分号分隔)

```
[atguigu@hadoop102 ~]$ kafka-console-producer.sh --bootstrap-server hadoop102:9092 --topic test_csv


>lisi;19
>wangwu;20
>
```

7）起一个 kafka console consumer 消费数据

```
[atguigu@hadoop102 ~]$ kafka-console-consumer.sh --bootstrap-server hadoop102:9092 --topic test_sink
{"name":"lisi","age":19}
{"name":"wangwu","age":20}
```

我们成功地实现了数据从 kafka 输入经过简单的 ETL 再向 kafka 输出。

4.2 Kafka 输出到 Doris 进行指标统计

4.2.1 需求

使用回话日志统计用户的总观看视频数，用户最常会话市场，用户最小会话时长，用

户最后一次会话时间。

4.2.2 需求实现

1）在资料中有一个伪数据的生成脚本，将它拷贝到服务器的任意位置

```
[atguigu@hadoop102 fake_data]$ pwd
/opt/module/fake_data
[atguigu@hadoop102 fake_data]$ ll
total 4
-rw-rw-r-- 1 atguigu atguigu 3406 Apr  1 10:27 fake_video.py
[atguigu@hadoop102 fake_data]$
```

2）执行以下命令安装 python 脚本需要的两个依赖库

```
pip3 install Faker
pip3 install kafka-python
```

3）使用 mysql 客户端连接 doris

```
[atguigu@hadoop102 fake_data]$ mysql -h hadoop102 -P 9030 -
uatguigu  -p123321
```

4）手动创建 test_db 数据库。

```
create database test_db;
```

5）使用下述 sql 语句建表

```
CREATE TABLE `example_user_video` (
  `user_id` largeint(40) NOT NULL COMMENT "用户 id",
  `city` varchar(20) NOT NULL COMMENT "用户所在城市",
  `age` smallint(6) NULL COMMENT "用户年龄",
  `video_sum` bigint(20) SUM NULL DEFAULT "0" COMMENT "总观看视频数
",
  `max_duration_time` int(11) MAX NULL DEFAULT "0" COMMENT "用户最
长会话时长",
  `min_duration_time` int(11) MIN NULL DEFAULT "999999999"
COMMENT "用户最小会话时长",
  `last_session_date` datetime REPLACE NULL DEFAULT "1970-01-01
00:00:00" COMMENT "用户最后一次会话时间"
) ENGINE=OLAP
AGGREGATE KEY(`user_id`, `city`, `age`)
COMMENT "OLAP"
DISTRIBUTED BY HASH(`user_id`) BUCKETS 16
;
```

6）在 config 目录下， 编写如下的配置文件。

```
env {
   execution.parallelism = 1
}

source {
    KafkaTableStream {
        consumer.bootstrap.servers = "hadoop102:9092"
        consumer.group.id = "seatunnel5"
        topics = test
```

```
        result_table_name = test
        format.type = json
        schema =
"{\"session_id\":\"string\",\"video_count\":\"int\",\"duration_ti
me\":\"long\",\"user_id\":\"string\",\"user_age\":\"int\",\"city\
":\"string\",\"session_start_time\":\"datetime\",\"session_end_ti
me\":\"datetime\"}"
        format.ignore-parse-errors = "true"
    }
}

transform{
    sql {
        sql = "select user_id,city,user_age as age,video_count as
video_sum,duration_time as max_duration_time,duration_time as
min_duration_time,session_end_time as last_session_date from
test"
        result_table_name = test2
    }
}

sink{
    DorisSink {
        source_table_name = test2
        fenodes = "hadoop102:8030"
        database = test_db
        table = example_user_video
        user = atguigu
        password = 123321
        batch_size = 50
        doris.column_separator="\t"

    doris.columns="user_id,city,age,video_sum,max_duration_time,mi
n_duration_time,last_session_date"
    }
}
```

7）使用 python 脚本向 kafka 中生成伪数据

```
[atguigu@hadoop102 fake_data]$ python3 fake_video.py --bootstrap-
server hadoop102:9092 --topic test_video
```

8）查看 doris 中的结果。

```
Select * from `example_user_video`;
```

第 5 章 如何参与开源贡献

5.1 基本概念

5.1.1 参与开源贡献的常见方法

1）参与解答

在社区中，帮助使用过程中遇到困难的人，帮他们解释框架的用法也算是一种贡献。

2）文档贡献

帮助框架来完善文档，比如说将英文文档翻译为中文，纠正文档里面的错误单词，这是很多人参与开源贡献的第一步。

3）代码贡献

经过阅读源码，发现源码中有 Bug，修改后将代码提交给社区。或者，框架有一个新的特性亟待开发，你为新功能的实现提供了解决方案，这属于代码贡献，也是一种重要的参与开源贡献的方式。

5.1.2 开源社区中常见的三个身份标签

1）contributor（贡献者）

只要参与过一次贡献就算是贡献者，

2）committer（提交者）

成为 contributor 后，如果你能保证持续贡献，而且有扎实的技术功底，经 PMC（管理委员会）投票或讨论决定后，可以决定让你成为一名 committer。Committer 和 contributor 的区别在于，commiter 对于项目的仓库是具有写的权限的。他可以审核并合并 contributor 的代码。而且如果成为 commiter，你还会获得一个后缀为@apache.org 的邮箱。

3）PMC（管理委员会）

Committer 中表现优秀的话，是可以成为 PMC 的。PMC 要负责整个项目的走向，做出一些重要的决策，要具备前瞻性的技术眼光。

5.2 如何修改 bug

5.2.1 背景

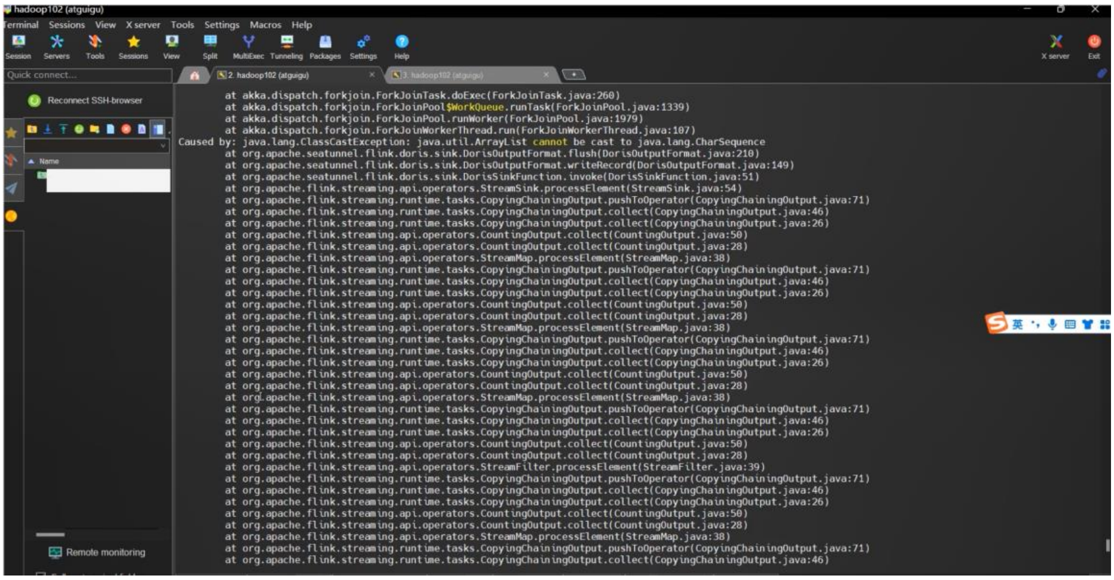
在我们准备这项课程的时候，实际上 kafka 输入插件，kafka 输出插件和 doris 输出插件是各有一个 bug 的，当时 kafka 输入插件的 bug 在社区中已经有了一个解决方案。Kafka 输

出插件的 bug，和 doris 输出插件的 bug 是我们来做的修改，而且修改后的结果提交给了 SeaTunnel 社区，并且成功实现了代码合并。下面我们复现一个 doris 输出插件 bug 的场景，并且在这个基础上向大家讲解如何一步步去参与开源贡献，成为一名源码贡献者。

5.2.2 问题复现

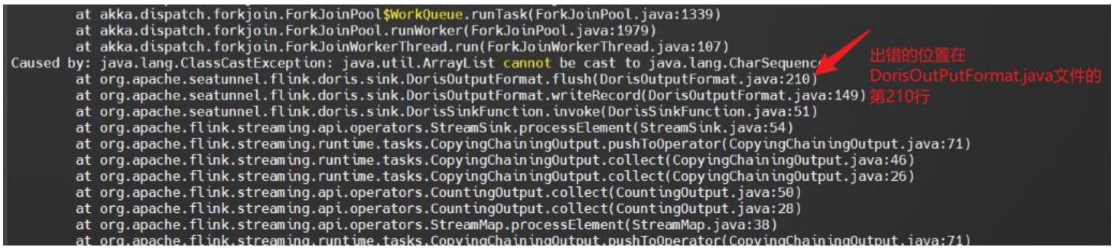
1）场景

当时，向 doris 插入数据时会抛出一个 ClassCastException，也就是类型的强转错误。这里会报 Java.Util.ArratList 不能强转为 java.lang.CharSequence。在反复确认我们的配置文件写的没问题后。我们仔细阅读了一下控制台打印的栈追踪信息。



2）问题定位

通过最后打印的栈追踪信息，我们可以知道出错的位置在 DorisOutPutFormat.java 文件的第 210 行，于是我们需要去 idea 里面打开源码看一下这里的代码是怎么写的。



5.2.3 分析问题

定位到 210 行后，我们看到下面的问题。

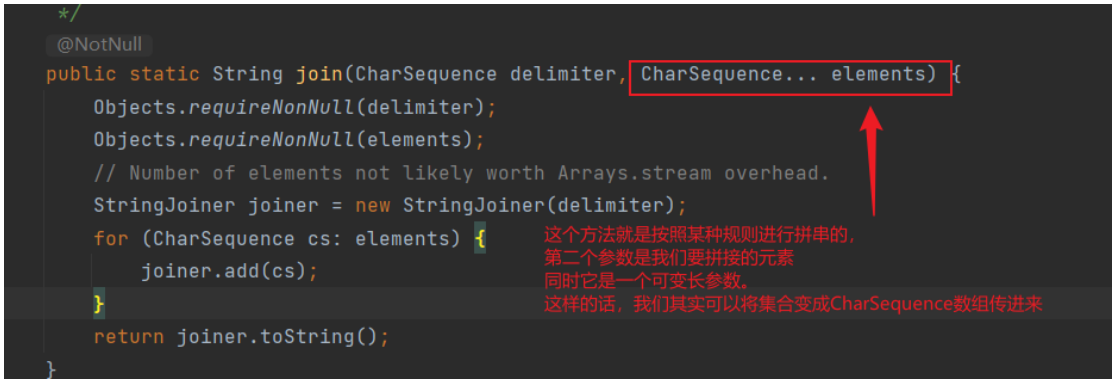
它要将一个 batch（它是一个 ArrayList 集合）强转为 CharSequence（字符序列）。这显然是错误的。

要想解决这个问题，我们要了解这段代码的意图。

这需要一定的背景知识，SeaTunnel 的 dorisSink 其实是依托于 doris 的 stream load 这种导入方式来实现的。而 stream load 其实是通过 http 请求的形式，向 doris 导入数据。而且 doris 提倡提交数据的时候一定要成批地向 doris 导入数据。如此一来，我们知道 bacth 就是用来积攒数据的一个集合，而向远端通过 http 发送数据必然要经过一个序列化的过程。结合上下文来看，我们可以判断这段代码的目的，就是要将 batch 里的所有数据，按照某个规则转为字符串，为 http 请求做准备。分析过程如图所示。

5.2.4 确定问题的解决方案

我们需要看一下 String 提供的这个 join 静态方法，对参数的要求。



我们发现，join 方法的第二个参数是一个 CharSequence 类型的可变长参数，这意味着我们可以向里面传递一个 CharSequence 类型的数组。那么代码可以修改成下面这个样子。

```
        }
    } else {
        result = String.join(this.lineDelimiter, (CharSequence) batch);
    }
```

```
} else {
    result = String.join(this.lineDelimiter, batch.toArray(new CharSequence[batch.size()]));
}
for (int i = 0; i <= maxRetries; i++) {
```

5.2.5 方案验证

1）重新打包

接着，我们可以重新编译这个包，把重新编译的包放到我们的集群上，再跑一次任务看看能不能通过。在这个过程中，因为跨平台性的问题(windows 和 linux 的路径不通用，其实也是个 bug)，有一些单元测试我们无法通过，因此我们取个巧，用下面的方式进行编译打包，跳过单元测试和代码的格式审查。

```
mvn clean package -D maven.test.skip=true -D checkstyle.skip=true
```

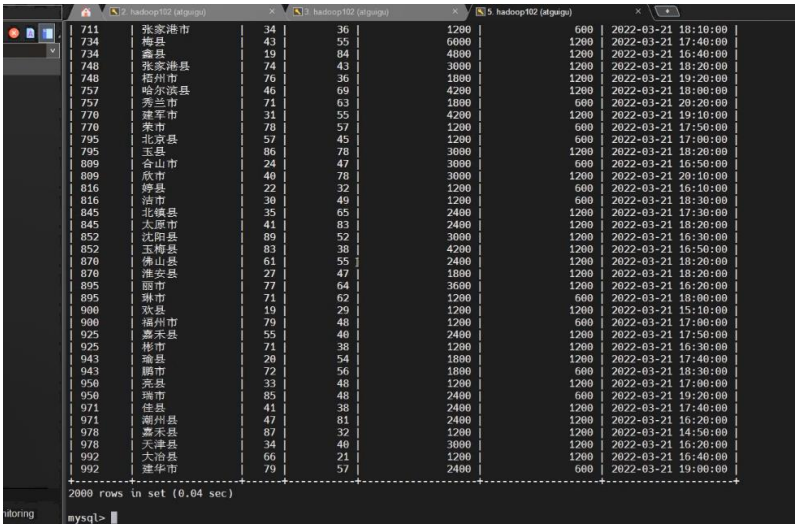
2）使用新的包

接着，我们使用重新编译过的 SeaTunnel 执行我们之前向 Doris 导入数据的命令。

```
bin/start-seatunnel-flink.sh --config config/example04.conf
```

3）到我们的 Doris 上查看数据是否成功导入

这次我们的数据成功导进了 doris。而且我们的程序并没有因为类型转换错误而崩溃。



711	张家港市	34	36	1200	600	2022-03-21 18:10:00
734	梅县	43	55	6000	1200	2022-03-21 17:40:00
734	鑫县	19	84	4800	1200	2022-03-21 16:40:00
748	张家港县	74	43	3000	1200	2022-03-21 18:20:00
748	梧州市	76	36	1800	1200	2022-03-21 19:20:00
757	哈尔滨县	46	69	4200	1200	2022-03-21 18:00:00
757	秀兰市	71	63	1800	600	2022-03-21 20:20:00
770	建军市	31	55	4200	1200	2022-03-21 19:10:00
770	荣市	78	57	1200	600	2022-03-21 17:50:00
795	北京县	57	45	1200	600	2022-03-21 17:00:00
795	玉县	86	78	3000	1200	2022-03-21 18:20:00
809	合山市	24	47	3000	600	2022-03-21 16:50:00
809	欣市	40	78	3000	1200	2022-03-21 20:10:00
816	婷县	22	32	1200	600	2022-03-21 16:10:00
816	洁市	30	49	1200	600	2022-03-21 18:30:00
845	北镇县	35	65	2400	1200	2022-03-21 17:30:00
845	太原市	41	83	2400	1200	2022-03-21 18:20:00
852	沈阳县	89	52	3000	1200	2022-03-21 16:30:00
852	玉梅县	83	38	4200	1200	2022-03-21 16:50:00
870	佛山县	61	55	2400	1200	2022-03-21 18:20:00
870	淮安县	27	47	1800	1200	2022-03-21 18:20:00
895	丽市	77	64	3600	1200	2022-03-21 16:20:00
895	琳市	71	62	1200	600	2022-03-21 18:00:00
900	欢县	19	29	1200	1200	2022-03-21 15:10:00
900	福州市	79	48	1200	600	2022-03-21 17:00:00
925	嘉禾县	55	40	2400	1200	2022-03-21 17:50:00
925	彬市	71	38	1200	1200	2022-03-21 16:30:00
943	璐县	20	54	1800	1200	2022-03-21 17:40:00
943	鹏市	72	56	1800	600	2022-03-21 18:30:00
950	亮县	33	48	1200	1200	2022-03-21 17:00:00
950	瑞市	85	48	2400	600	2022-03-21 19:20:00
971	佳县	41	38	2400	1200	2022-03-21 17:40:00
971	潮州县	47	81	2400	1200	2022-03-21 16:20:00
978	嘉禾县	87	32	1200	1200	2022-03-21 14:50:00
978	天津县	34	40	3000	1200	2022-03-21 16:20:00
992	大冶县	66	21	1200	1200	2022-03-21 16:40:00
992	建华市	79	57	2400	600	2022-03-21 19:00:00

2000 rows in set (0.04 sec)

mysql>

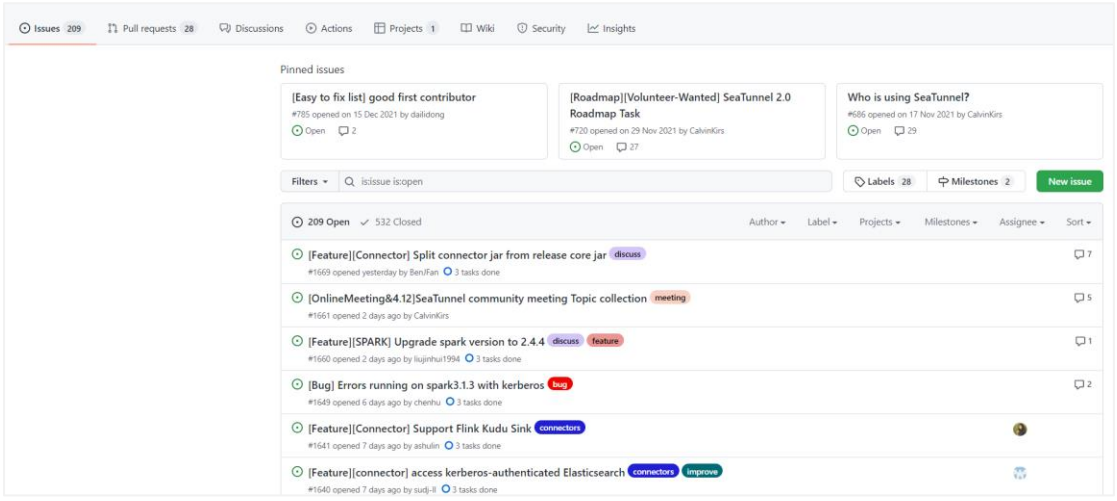
4）小结

经过上面的这些步骤，我们确信问题是出在源码的问题上。接下来我们要开始向社区汇报这个 bug，并向社区提供我们的解决方案。

5.3 创建 issue

5.3.1 什么是 issue

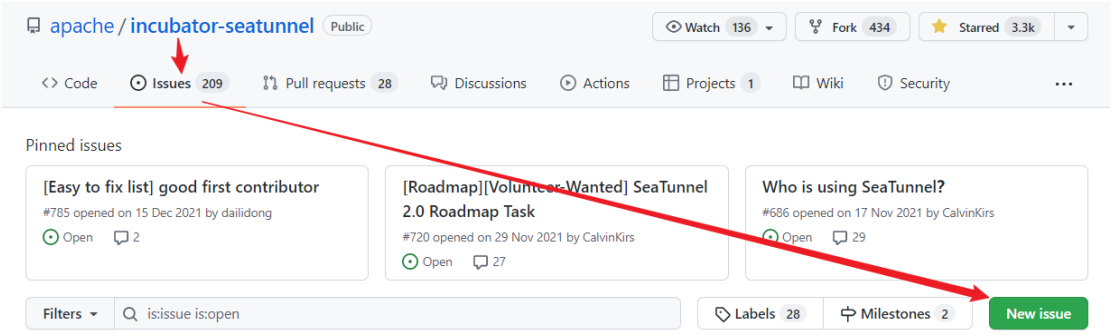
每个 github 的仓库下都会有一个项目独立的 issue 板块。在这个板块里面，大家可以提出自己的问题，也可以去和大家讨论 SeaTunnel 是否要添加一些特性。而且，这是一个可以汇报 bug 的地方。



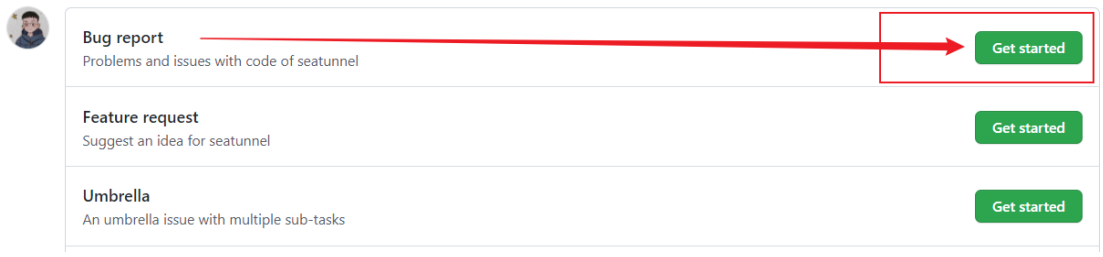
开源社区通常会要求你在提交代码合并的请求前，先去创建一个 issue。这是一个好的习惯，就像是我们抓贼要先立案，逮捕要先有逮捕令。创建 pull request 之前先创建 issue，然后把 pr 关联到我们创建的 issue 上，让每一次改动，都有据可查。

5.3.2 如何创建 issue

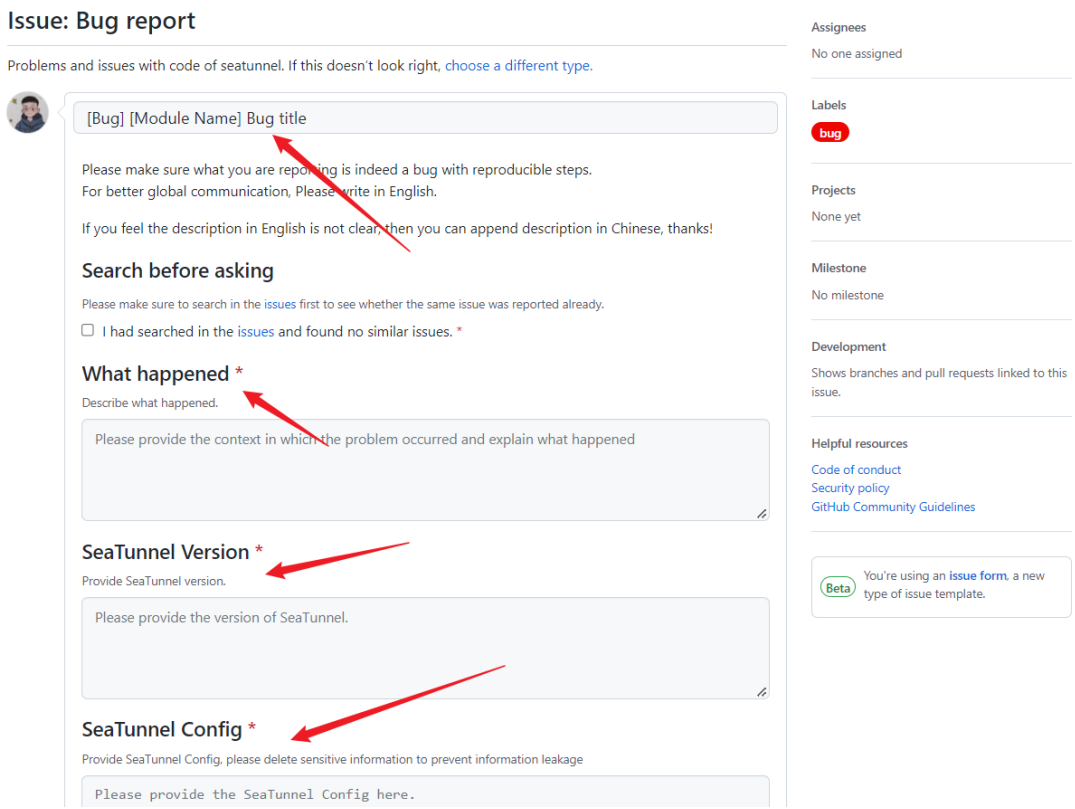
1）点击 new issue 按钮进入下一个页面



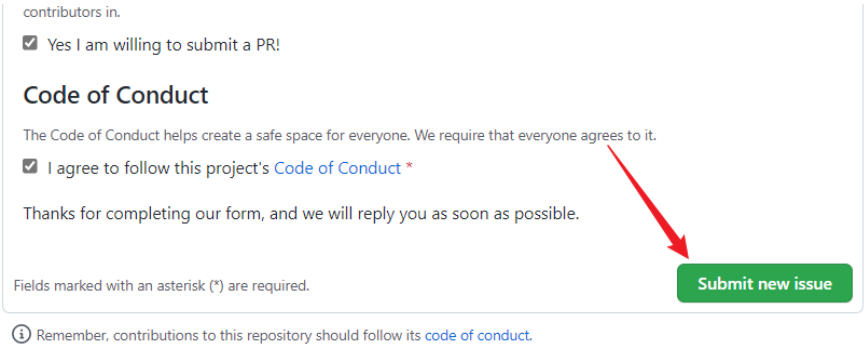
2）选择你要创建的 issue 类型，我们选择 bug report（bug 汇报），进入下一个页面



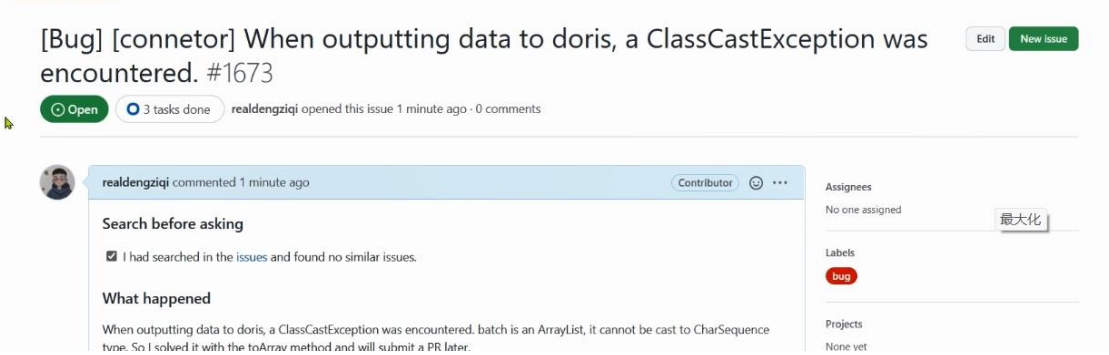
3）按照表单的提示，一步步填写完整。注意，表单提醒你，创建 issue 之前应该先去搜索社区中是否已经有讨论同一问题的 issue。同样的问题，无需重复。



4）按照要求填写表单后，点击下方的 Submit new issue。创建这个 issue。



5）查看我们已经创建好的 issue



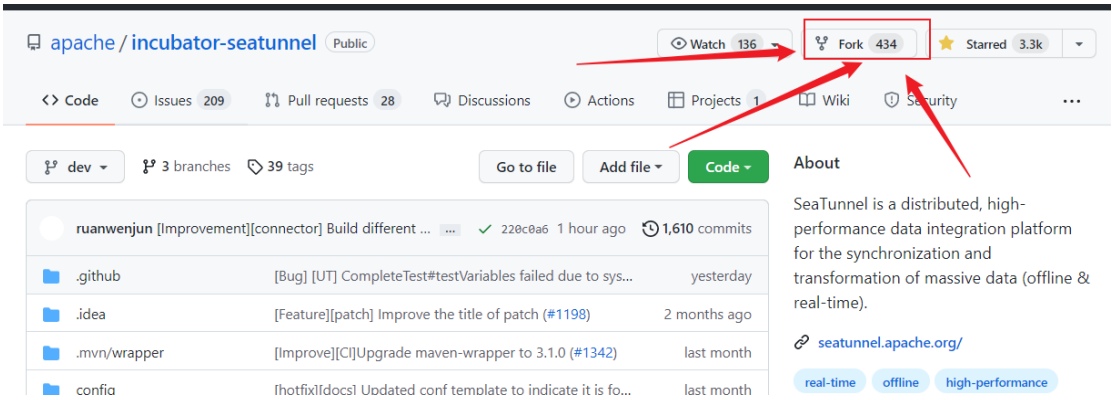
5.4 创建 pull request

pull request 的意思是拉取请求，也就是我这有代码写好了，请你把我的代码拉过去吧。所以，发起拉取请求之前应该要先有自己的代码。这样一来，创建 pull request 并不是一上来就创建，而是要先搞好自己的代码仓库。

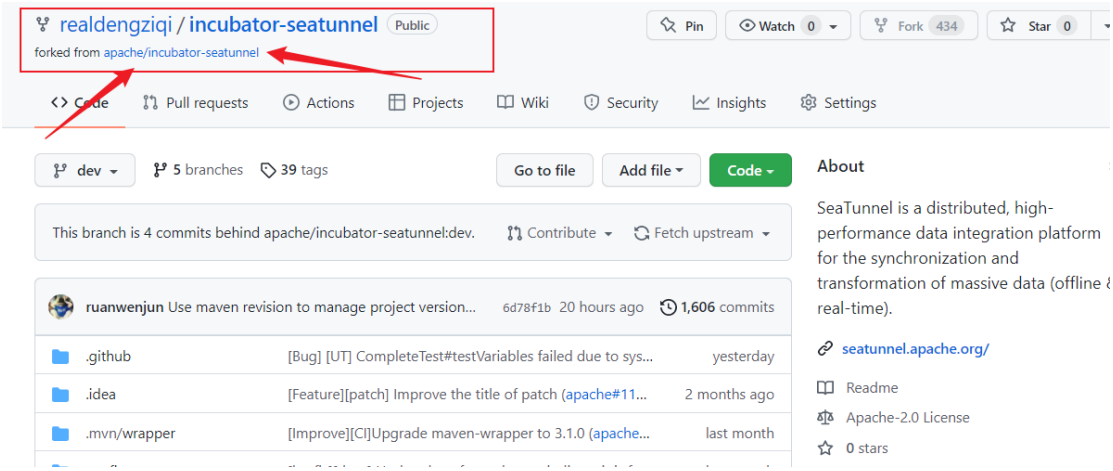
pull request 的简称是 pr。

5.4.1 fork 项目到自己的仓库中

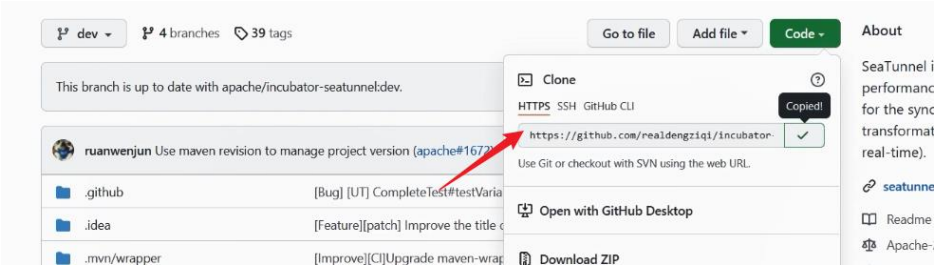
对于第一次对 SeaTunnel 贡献的同学来说，应该先 fork（叉子）官方的仓库。



点击 fork 按钮后，你自己的 github 账号上会出现一个一模一样的仓库。如下图所示。



5.4.2 git clone 自己 fork 的仓库



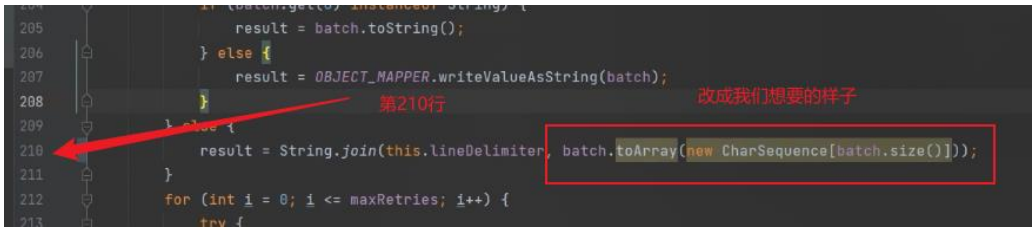
拿到这个 url，在自己电脑上的任意目录上使用下面的 git 命令去 clone 这个仓库。

```
git clone xxxx{你自己的仓库的 url}xxx
```

5.4.3 修改代码

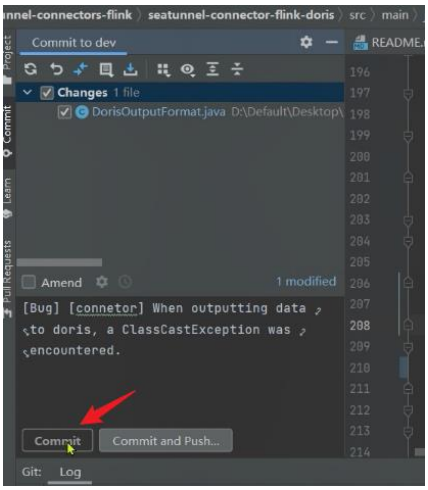
1）在项目的跟目录右键，用 idea 打开我们 clone 的项目

2）在我们之前确定的位置，改代码

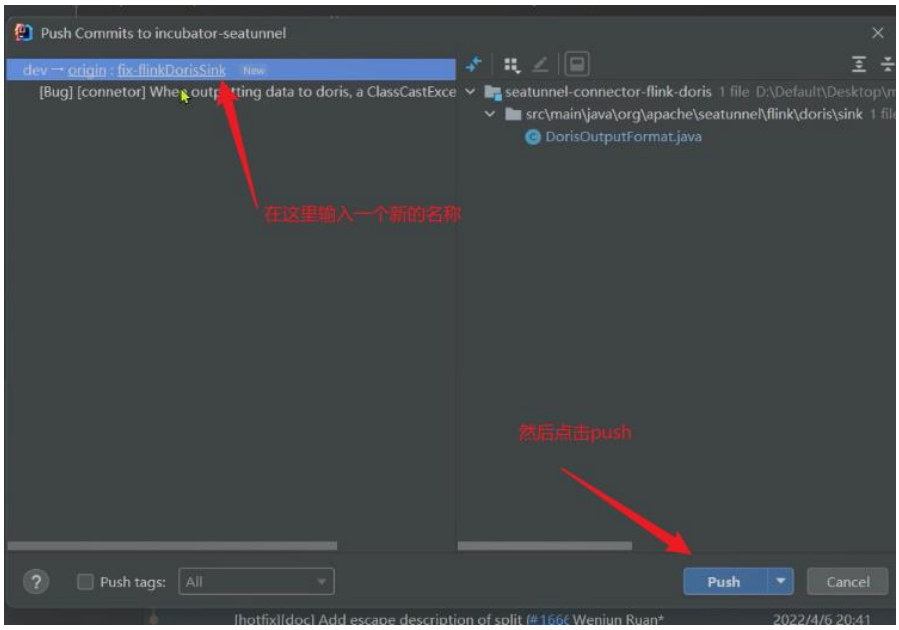


3）commit 提交

（这个地方应该先建一个分支，从 dev 上分出来，在新建分支的基础上 commit。这里成反面教材了^_^）

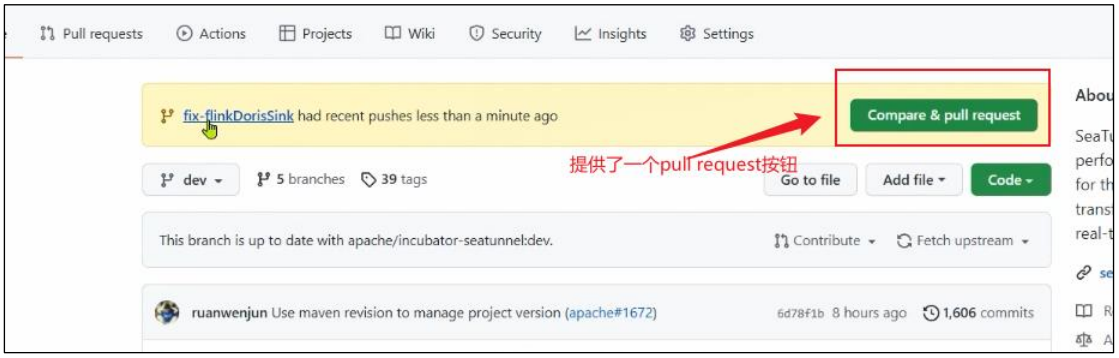


4）push 到我们 fork 的仓库里去，这个时候在远端的目标分支上，我们写一个新的分支名

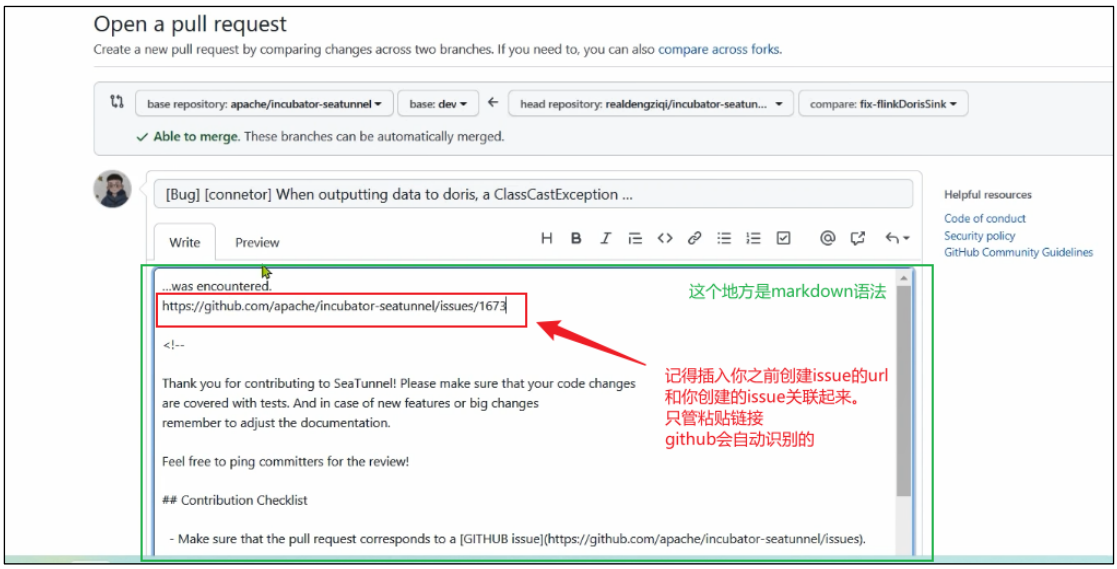


5.4.4 创建 PR

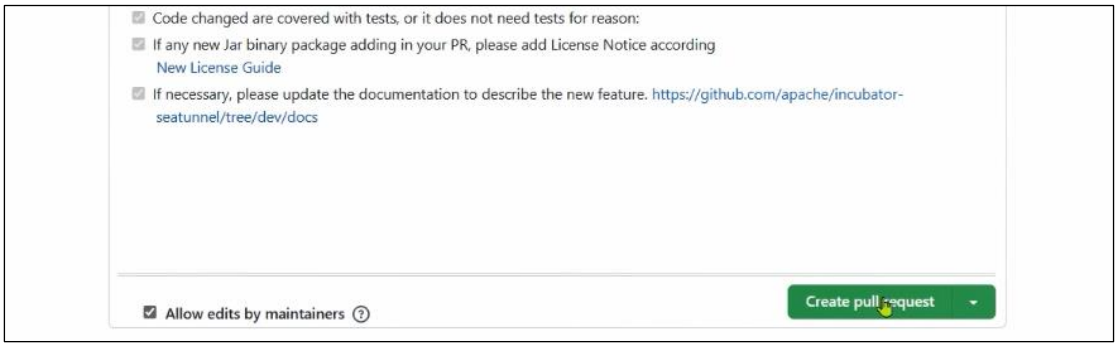
1）去我们的 github 上，看一下自己的仓库，发现它会提示我们可以创建一个 pr 了。点击这个按钮，进入下一个页面



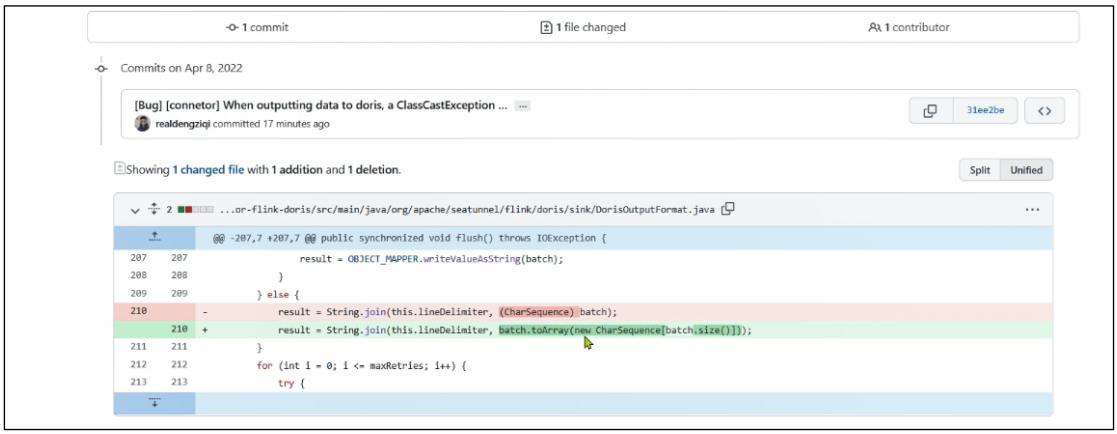
2）在新的页面中，按照对话框里给出的模板，说明我们这个 pr 的目的。最终，不要忘了和你之前的 issue 关联起来，关联的方式就是直接粘贴你创建的 issue 的链接。



3）全部搞定之后，点击 create pull request 按钮，创建一个 pr



4）我们还可以看到 github 会判断我们做了哪些修改。红色的地方表示我们删除的代码，绿色的地方表示我们新增的代码。因为 github 的差异是按行进行标记的。所以如果你就改了一个字母。也是一个删除行和新增行的效果。



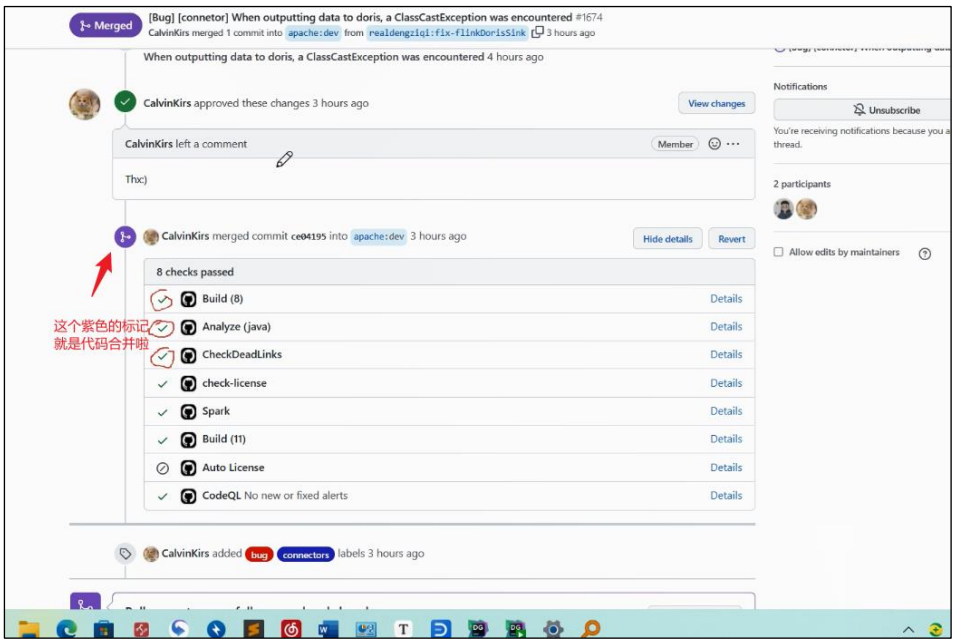
5）我们的 PR 已经提交完毕，我们可以看到 github 会启动一个自动的检查。这个叫做 CI/CD。持续交付/持续部署的意思。简单来说，你上传的代码，云端会自动拉取，然后自动地跑一边编译，然后进行单元测试，代码格式等一系列检查。这些测试都通过后，你的代码才有

被合并的可能。

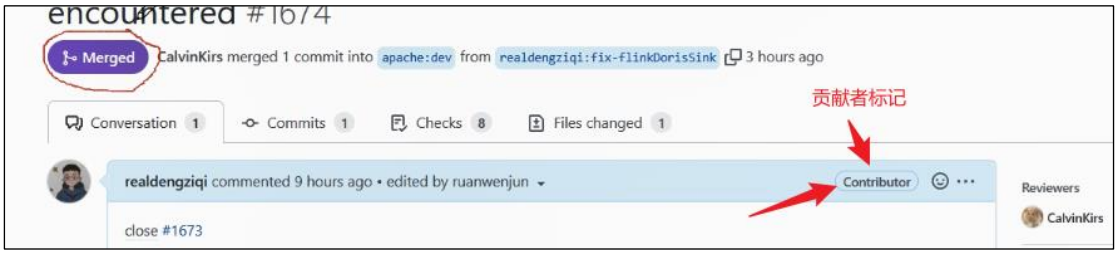
6）接下来你可以去干点别的，自动测试的时间会比较久，而且你需要等待社区人员注意到你的 pull request。

5.5 成功成为源码贡献者

过一段时间就可以回来看一下你的 pr 了。我们看到有一个 apache member 审核了我们的代码，并将我们的代码合并到了项目中。以后，大家使用 seatunnel 将数据从 flink 写入 doris，就有你的一份功劳了。



你的发言记录上，会出现 contributor 的标记。



弄完这些，就算是 SeaTunnel 的源码贡献者啦。

5.6 寻找贡献机会

Apache 的开源项目中，社区成员们通常会维护一个待办列表，里面是一些好做的任务。适合新手上路。