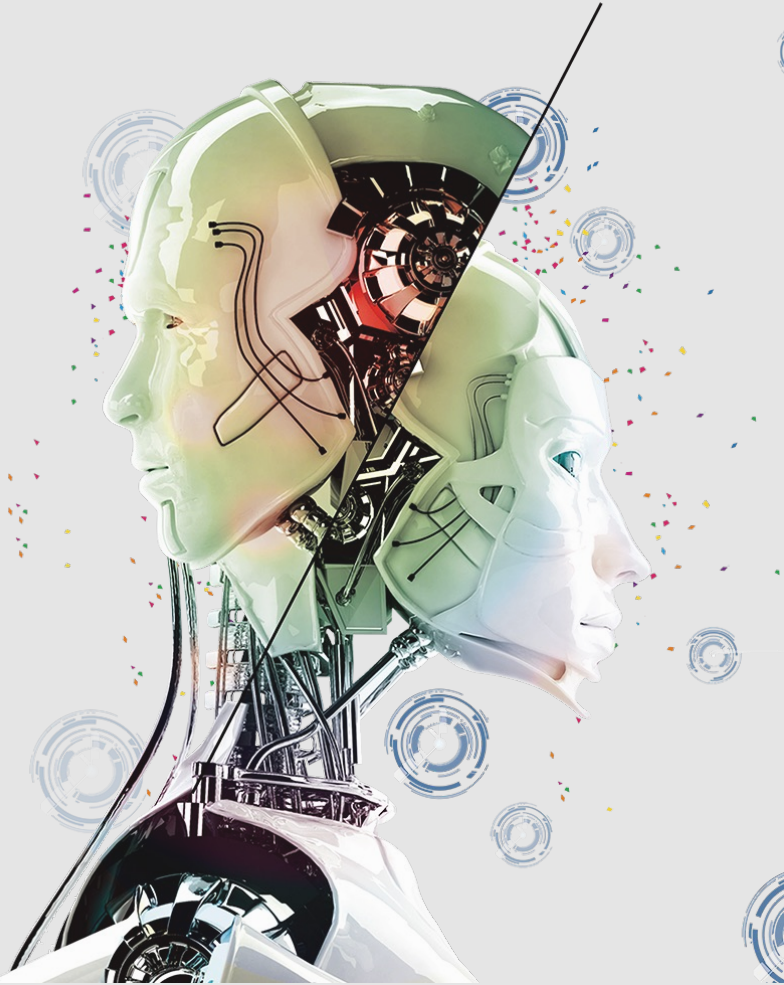
Apache InLong 的 SPI 扩展实践

周康 腾讯大数据 高级开发工程师

Apache InLong PMC 成员

Email: healchow@apache.org

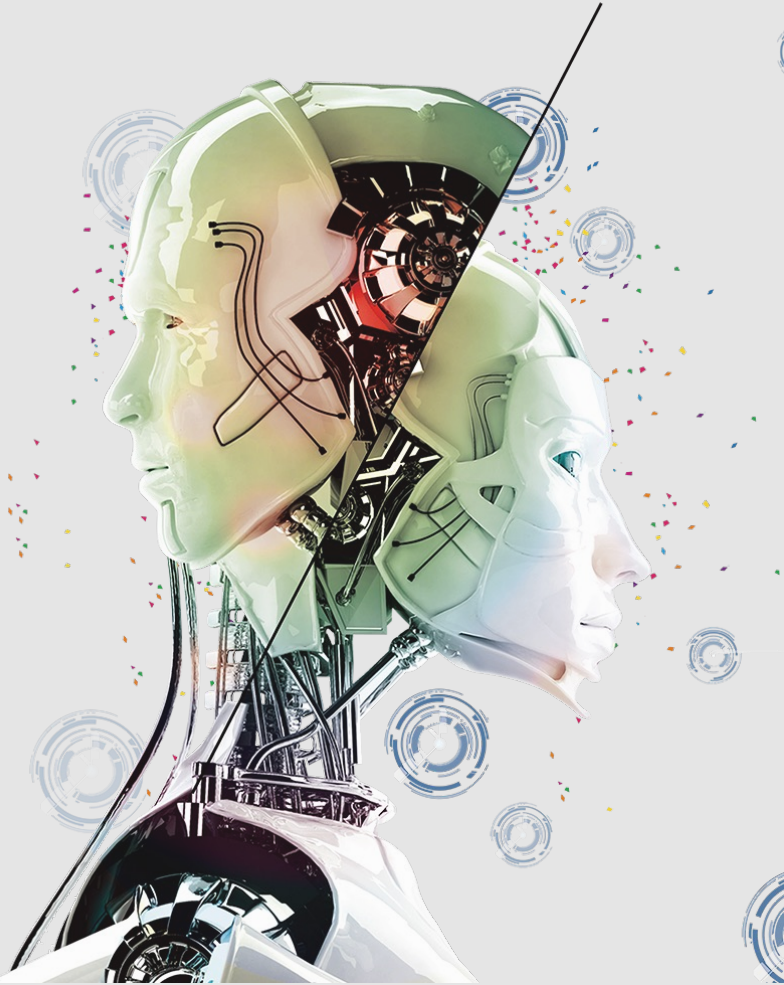


目录 CONTENT

腾讯大数据 TENCENT BIG DATA | Apache INLONG | DataFun.

01 Apache InLong 简介

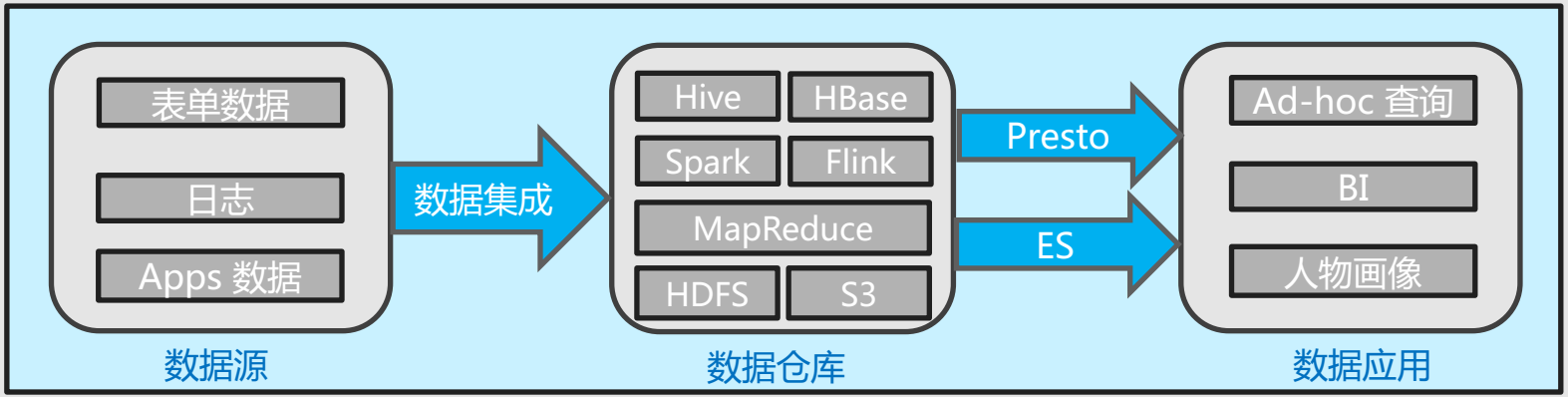
- 项目简介
- 适用场景



01 Apache InLong 简介 / 是什么

官网：https://inlong.apache.org

Apache InLong（应龙）是一个**一站式的海量数据集成框架**，提供自动、安全、可靠和高性能的数据传输能力，方便业务构建基于流式的数据分析、建模和应用。

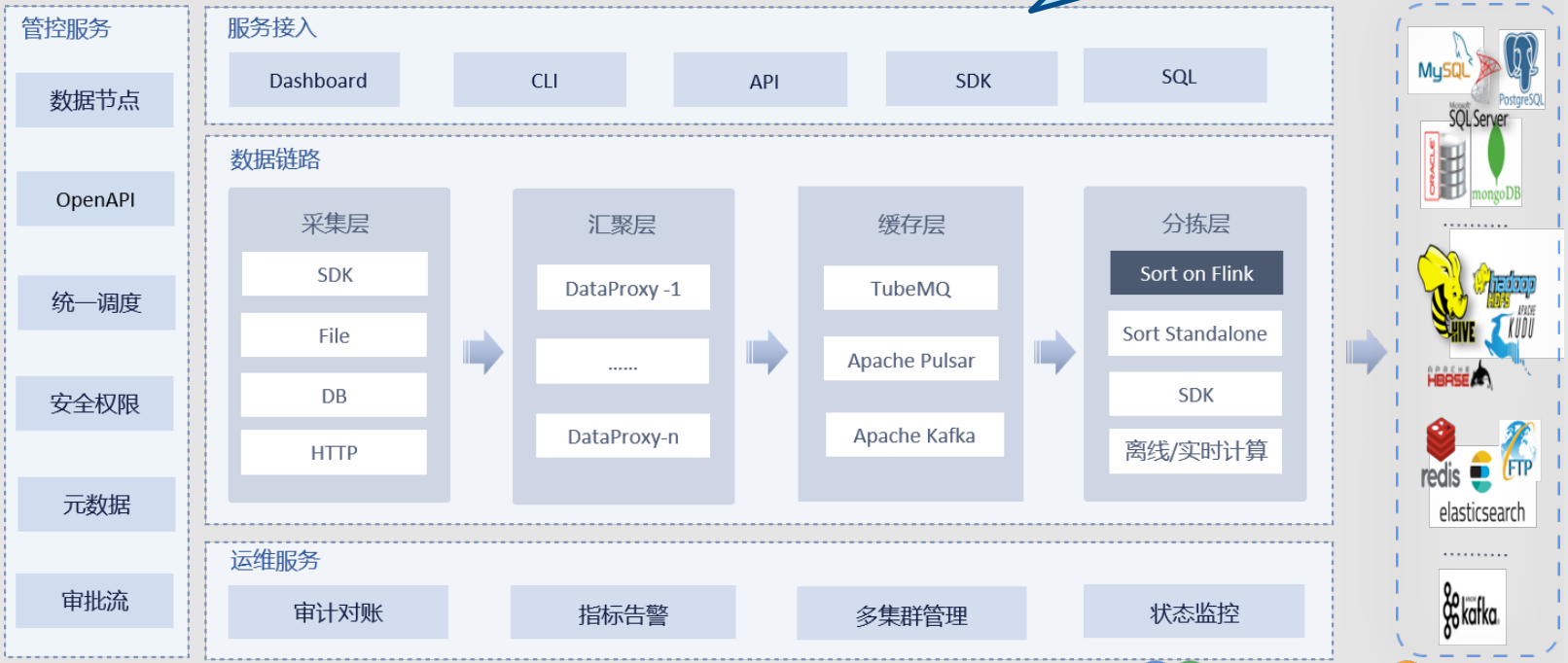


最初于 2019 年 11 月由腾讯大数据团队捐献到 Apache 孵化器，2022 年 6 月正式孵化毕业，成为 **Apache 顶级项目（TLP）**。

01 Apache InLong 简介 / 适用场景

依托腾讯百万亿级别的数据接入和处理能力，
整合了**数据采集、汇聚、缓存、分拣**全流程，
具有简单易用、稳定可靠、灵活扩展等特性。

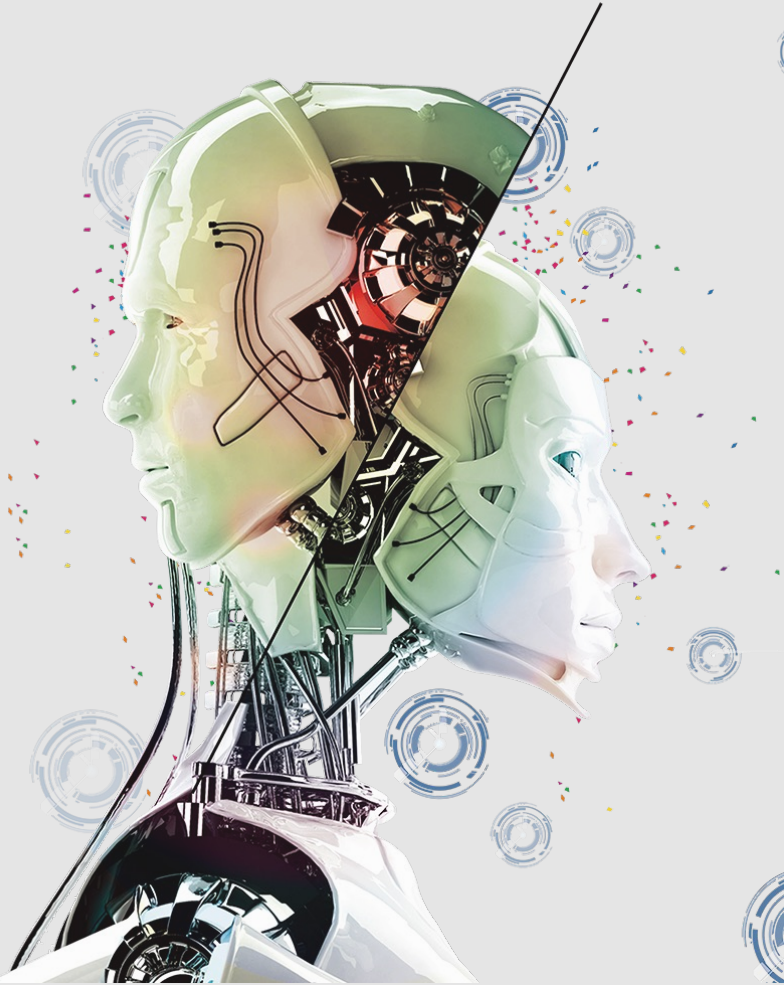
广泛应用于广告、支付、社交、游戏、运营商、人工智能等领域。



腾讯大数据 TENCENT BIG DATA | Apache INLONG | DataFun.

02 InLong Manager 简介

- Manager 的作用

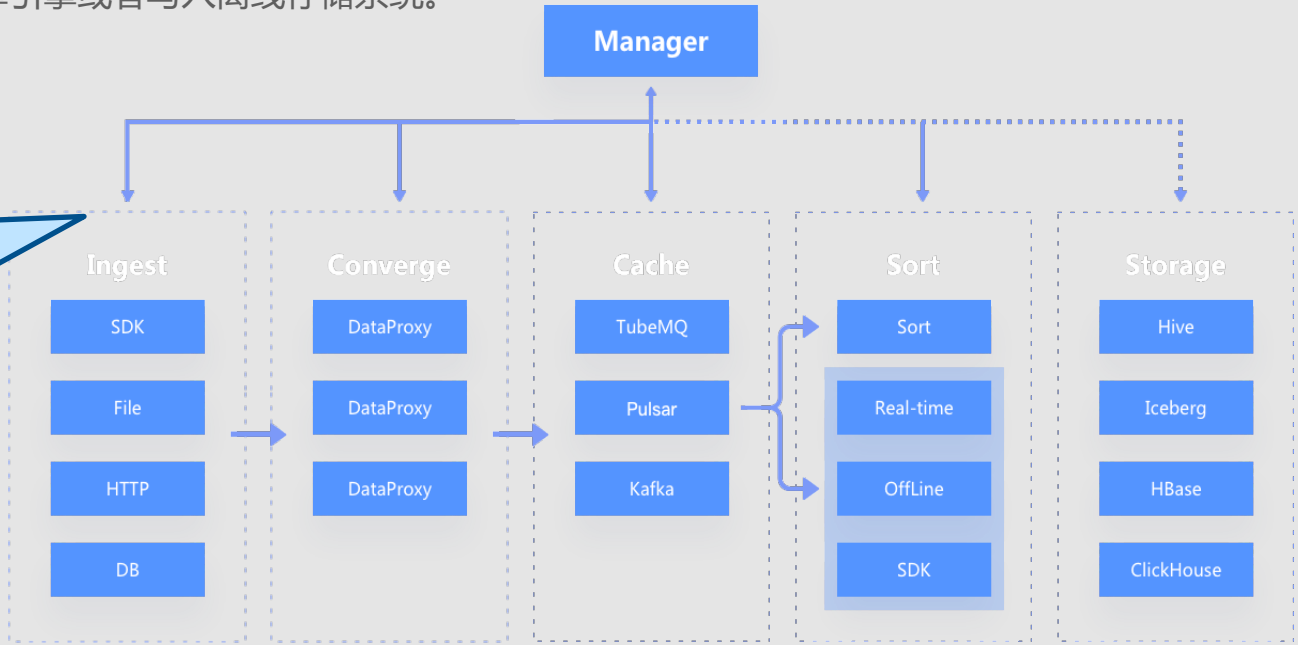


02 InLong Manager 简介

InLong 支持数据的**采集、汇聚、缓存和分拣**，只需一些基础配置，就可把数据从源端导入到实时计算引擎或者写入离线存储系统。

系统如何将这些流程串联起来？

通过 InLong Manager 来管理系统和任务的元数据，串联任务的全流程。



元数据主要有：

- 审批信息
- 集群配置信息
- 数据 schema 配置（源、目标...）

02 InLong Manager 简介

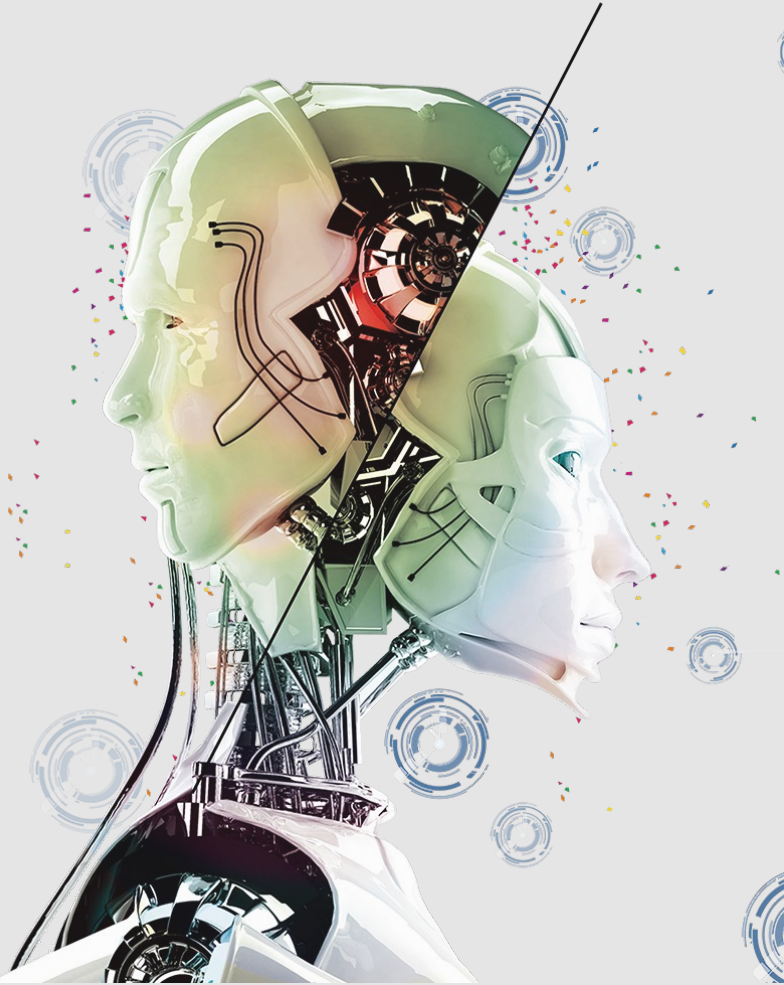
通过 InLong Dashboard 提供的 Web UI（或 Manager Client 提供的命令行工具）创建数据流任务，任务审批通过后，即可串联起全部流程，主要包括：

1. 创建目标端的库表结构
2. 创建 MQ 的 Topic 和消费者
3. 启动 Flink 任务，开始从 MQ 消费数据，写入目标端
4. 下发采集任务，向 MQ 生产数据

腾讯大数据 TENCENT BIG DATA | Apache INLONG | DataFun.

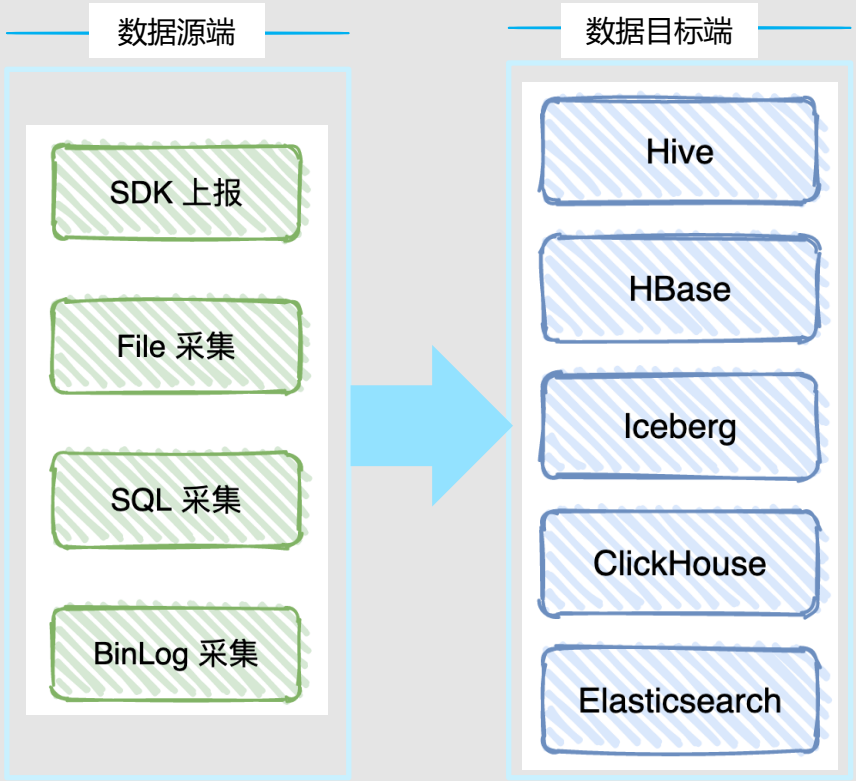
03 Manager 的 SPI 改造实践

- 什么是 SPI
- 改造过程
- 改造后的收益



03 Manager 的 SPI 改造实践 / 存在的问题

InLong 源于腾讯内网业务，在近10年的发展中，主要支持的数据源和数据存储如下：



03 Manager 的 SPI 改造实践 / 存在的问题

以数据存储端为例，由于用到的存储类型有限，且考虑到不同的存储类型的参数差异较大，
我们的配置表是这样设计的：

```
CREATE TABLE `ck_sink` (
    `id`              int(11),
    `inlong_group_id` varchar(256),
    `inlong_stream_id` varchar(256),
    `sink_name`       varchar(128),
    `jdbc_url`        varchar(1024),
    `username`        varchar(128),
    `password`        varchar(512),
    `db_name`         varchar(256),
    `table_name`      varchar(256),
    `is_distributed`  tinyint(1),
    `key_field_names` varchar(1024),
    `engine`          varchar(50),
    `partition_by`    varchar(128),
    `order_by`        varchar(128),
    `ck_version`      varchar(10)
    -- status, creator, create_time, e
);
```

```
CREATE TABLE `hive_sink` (
    `id`              int(11),
    `inlong_group_id` varchar(256),
    `inlong_stream_id` varchar(256),
    `sink_name`       varchar(128),
    `jdbc_url`        varchar(1024),
    `username`        varchar(128),
    `password`        var
    `db_name`
    `table_name`
    `data_path`
    `file_format`     va
    `data_encoding`   varchar(20),
    `data_separator`  varchar(10),
    `hive_version`    varchar(10)
    -- status, creator, create_time, e
);
```

```
CREATE TABLE `es_sink` (
    `id`              int(11),
    `inlong_group_id` varchar(256),
    `inlong_stream_id` varchar(256),
    `sink_name`       varchar(128),
    `host`            varchar(1024),
    `port`            varchar(20),
    `username`        varchar(128),
    `password`        varchar(512),
    `index_name`      varchar(256),
    `flush_interval`  varchar(11),
    `flush_record`    int(11),
    `partition_by`    varchar(128),
    `document_type`   varchar(128),
    `es_version`      varchar(10)
    -- status, creator, create_time, etc.
);
```

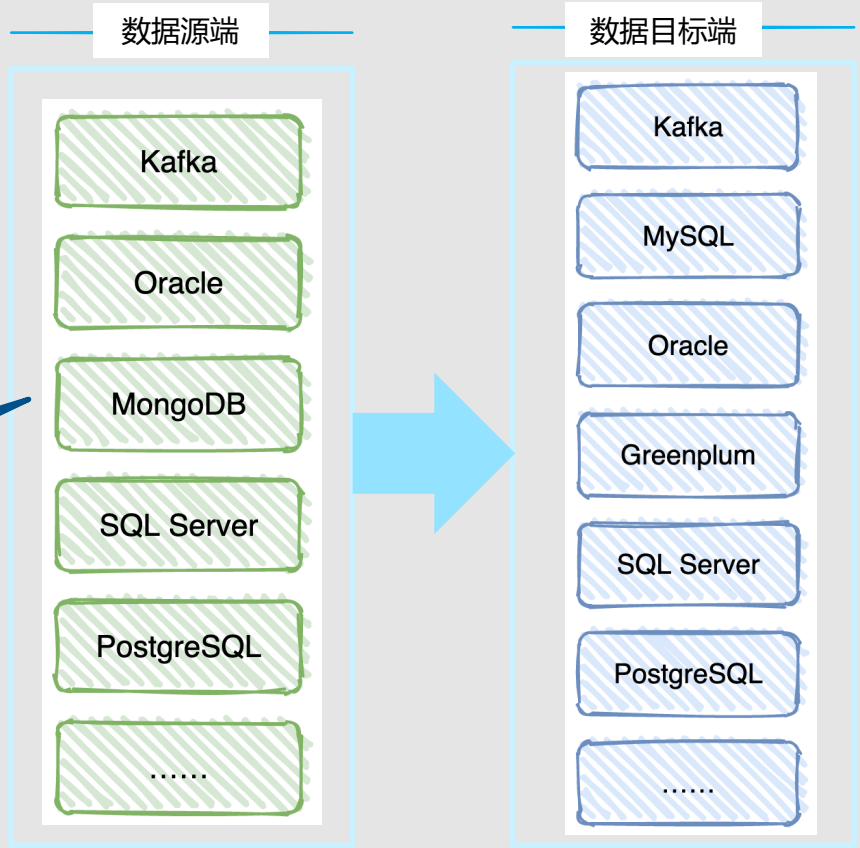
重复字段

03 Manager 的 SPI 改造实践 / 存在的问题

在 InLong 上云的过程中，
数据源端和目标端的类型急剧增多。

可以预见，随着云上客户规模的增加，
还会继续扩展更多类型的数据源和目标端。

在前面已有类型的基础上，新增的类型



03 Manager 的 SPI 改造实践 / 存在的问题

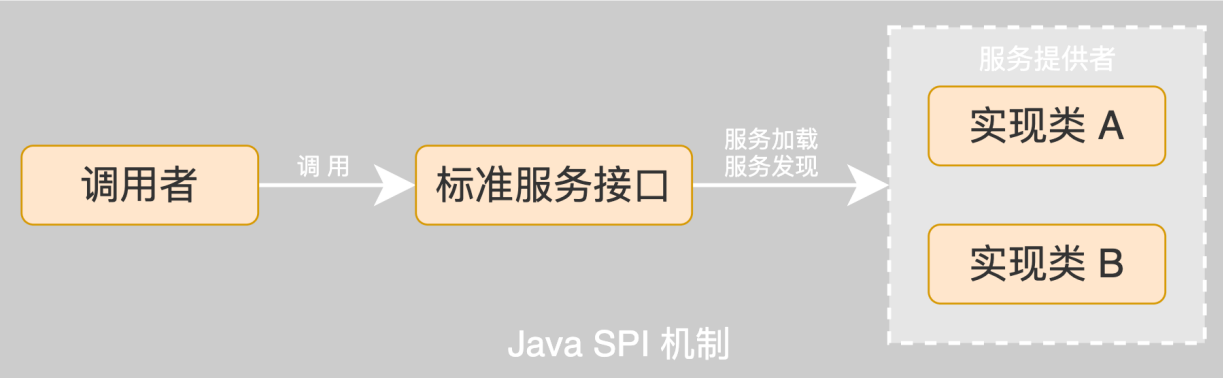
扩展的过程中发现的痛点：

- 维护成本高：表多，重复字段多
- 大量相似代码（if-else / switch-case 处理相似逻辑）
- 难扩展：要扩展新的存储类型，不仅要添加一张表，还要修改接口中的代码，添加 else / case 语句（不符合开闭原则）

```
switch (sinkType.toUpperCase(Locale.ROOT)) {
    case SinkType.HIVE:
        id = storageHiveOperation.saveSink(sinkInfo, operator);
        break;
    case SinkType.ICEBERG:
        id = icebergOperation.saveSink(sinkInfo, operator);
        break;
    case SinkType.CLICK_HOUSE:
        id = clickHouseOperation.saveSink(sinkInfo, operator);
        break;
    case SinkType.ELASTICSEARCH:
        id = esOperation.saveSink(sinkInfo, operator);
        break;
    default:
        throw new BusinessException(String.format("Unsupported type %s", sinkType));
}
```

03 Manager 的 SPI 改造实践 / 什么是 SPI

SPI，全称 Service Provider Interface，是 Java 提供的一套用来被第三方实现或者扩展的 API，它可以用来启用框架扩展和替换组件。
翻译后是“服务提供者接口”，顾名思义，这个接口是给“服务提供者”使用的。



常见示例：

- 加载 数据库驱动 load 接口的实现类
- SLF4J 加载不同提供商的日志实现类 日志门面接口的实现类
- Spring 中自动类型转换 Type Conversion SPI (Converter SPI、Formatter SPI) 等

03 Manager 的 SPI 改造实践 / 什么是 SPI

以 Flink JDBC Connector 中对不同 JDBC 方言的处理为例：

1）首先定义一个开放给外部去实现的 JdbcDialectFactory 接口，由不同的 DB Dialect 去实现：

```java
@PublicEvolving
public interface JdbcDialectFactory {

    /**
     * Retrieves whether the dialect thinks that it can open a connection to the
     * given URL. Typically, dialects will return true if they understand the sub-
     * protocol specified in the URL and false if they do not.
     * Params: url – the URL of the database
     * Returns: true if this dialect understands the given URL; false otherwise.
     */
    boolean acceptsURL(String url);

    /**
     * Returns: Creates a new instance of the JdbcDialect.
     */
    JdbcDialect create();
}
```

03 Manager 的 SPI 改造实践 / 什么是 SPI

2）在 *classpath* 的 *META-INF/*services 目录下创建一个名为此接口全限定名的文件，内容是各个实现类的全限定名：


```
16  org.apache.flink.connector.jdbc.dialect.derby.DerbyDialectFactory
17  org.apache.flink.connector.jdbc.dialect.mysql.MySqlDialectFactory
18  org.apache.flink.connector.jdbc.dialect.psql.PostgresDialectFactory
19  org.apache.flink.connector.jdbc.dialect.oracle.OracleDialectFactory
```

3）程序中通过 *java.util.ServiceLoder* 扫描 *META-INF/services* 目录下的配置文件，根据实现类的全限定名来动态装载具体的实现类：

JdbcDialectLoader.java | JdbcDialectFactory.java

```
93  @   private static List<JdbcDialectFactory> discoverFactories(ClassLoader classLoader) {
94          try {
95              final List<JdbcDialectFactory> result = new LinkedList<>();
96              ServiceLoader.load(JdbcDialectFactory.class, classLoader)
97                      .iterator()
98                      .forEachRemaining(result::add);
99              return result;
100         } catch (ServiceConfigurationError e) {
101             LOG.error("Could not load service provider for jdbc dialects factory.", e);
102             throw new RuntimeException(
103                     "Could not load service provider for jdbc dialects factory.", e);
104         }
105     }
```

只需要调用 3）中 *JdbcDialectLoader* 的加载方法，即可根据传入的参数动态选中符合条件的实现类，

比如传入的 JDBC URL 以 jdbc:mysql: 开头，在 *JdbcDialectLoader* 中就会返回 MySqlDialectFactory：

```
public static JdbcDialect load(String url) {
    ClassLoader cl = Thread.currentThread().getContextClassLoader();
    List<JdbcDialectFactory> foundFactories = discoverFactories(cl);
    // ...
    final List<JdbcDialectFactory> matchingFactories = foundFactories.stream()
            .filter(f -> f.acceptsURL(url)).collect(Collectors.toList());
    // ...
    return matchingFactories.get(0).create();
}
```

→→ Java SPI 实际上是“**基于接口的编程 + 策略模式 + 配置文件**”组合实现的动态加载机制。

03 Manager 的 SPI 改造实践 / 改造过程

1）精简服务层代码，删除繁琐的 if-else / switch-case。

收敛 Service 层的接口，同一领域模型的请求都在同一个 Service 接口中处理，
以保存操作为例：

```
public Integer save(SinkRequest request, String operator) {
    // ...
    // According to the sink type, save sink information
    StreamSinkOperator sinkOperator = operatorFactory.getInstance(request.getSinkType());
    return sinkOperator.saveOpt(request, operator);
}
```

代码路径：org.apache.inlong.manager.service.sink.StreamSinkServiceImpl

03 Manager 的 SPI 改造实践 / 改造过程

根据不同的类型，通过执行器的工厂找到具体的配置执行器，该执行器去执行真正的保存方法：

```
@Service
public class SinkOperatorFactory {

    // Spring 自动加载 StreamSinkOperator 接口的所有实现者，与 SPI 要达到的效果相同
    @Autowired
    private List<StreamSinkOperator> sinkOperatorList;

    /**
     * Get a sink operator instance via the given sinkType
     */
    public StreamSinkOperator getInstance(String sinkType) {
        return sinkOperatorList.stream()
                .filter(inst -> inst.accept(sinkType))
                .findFirst()
                .orElseThrow(() -> new BusinessException(ErrorCodeEnum.SINK_TYPE_NOT_SUPPORT,
                        String.format(ErrorCodeEnum.SINK_TYPE_NOT_SUPPORT.getMessage(), sinkType)));
    }

}
```

代码路径：org.apache.inlong.manager.service.sink.SinkOperatorFactory

03 Manager 的 SPI 改造实践 / 改造过程

2）重构数据库实体模型，一张表支持任意类型 Sink 的配置。
表中记录通用字段，对于不同类型 Sink 的特有字段，通过一个扩展字段来存储（KV，JSON）：

```
CREATE TABLE IF NOT EXISTS `stream_sink`
(
    `id`                    int(11)      NOT NULL AUTO_INCREMENT COMMENT 'Incremental primary key',
    `inlong_group_id`       varchar(256) NOT NULL COMMENT 'Owning inlong group id',
    `inlong_stream_id`      varchar(256) NOT NULL COMMENT 'Owning inlong stream id',
    `sink_type`             varchar(15)           DEFAULT 'HIVE' COMMENT 'Sink type, including: HIVE, ES, etc'
    `sink_name`             varchar(128) NOT NULL DEFAULT '' COMMENT 'Sink name',
    `description`           varchar(500) NULL COMMENT 'Sink description',
    `enable_create_resource` tinyint(1)           DEFAULT '1' COMMENT 'Whether to enable create sink resource?
    `data_node_name`        varchar(128)          DEFAULT NULL COMMENT 'Node name, which links to data_node ta
    `ext_params`            text         NULL COMMENT 'Another fields, will be saved as JSON type',
    `operate_log`           text                  DEFAULT NULL COMMENT 'Background operate log',
    `status`                int(11)               DEFAULT '100' COMMENT 'Status',
    `previous_status`       int(11)               DEFAULT '100' COMMENT 'Previous status',
    `is_deleted`            int(11)               DEFAULT '0' COMMENT 'Whether to delete, 0: not deleted, > 0:
    `creator`               varchar(64)  NOT NULL COMMENT 'Creator name',
    `modifier`              varchar(64)           DEFAULT NULL COMMENT 'Modifier name',
    `create_time`           timestamp    NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT 'Create time',
```

03 Manager 的 SPI 改造实践 / 改造过程

不同类型 Sink 的特有参数，转换成 JSON 格式存储到 *ext_params* 字段中，查询时再将其解析成其特有的 DTO。相关代码可以参考：

- `org.apache.inlong.manager.service.sink.AbstractSinkOperator` 中 `saveOpt` 方法调用的 `setTargetEntity` 方法
- `org.apache.inlong.manager.service.sink.StreamSinkOperator` 接口中 `getFromEntity` 方法的各个实现

其他类似操作，感兴趣的朋友请检视：

- `org.apache.inlong.manager.service.group.InlongGroupOperator`
- `org.apache.inlong.manager.service.sink.StreamSinkOperator`
- `org.apache.inlong.manager.service.source.StreamSourceOperator`
- `org.apache.inlong.manager.service.resource.sink.SinkResourceOperator`
- `org.apache.inlong.manager.service.resource.queue.QueueResourceOperator`

03 Manager 的 SPI 改造实践 / 改造后的收益

在经过上述改造后，至少带来了如下收益：

1）代码复用性提高，减少了大量重复/相似逻辑的代码，降低了维护成本

2）代码扩展性极大增强，增加不同类型的配置，只需要依葫芦画瓢，去实现其特殊逻辑即可，也无需改动已有接口

3）表的 DDL 不用频繁变动，降低了维护成本，避免修改 DDL / 增加新表 引发线上问题

4）可以在不侵入修改开源代码的情况下，扩展腾讯内部的配置类型，加入内网特有的业务逻辑。

感谢观看

腾讯大数据 TENCENT BIG DATA | Apache INLONG | DataFun.

官网：https://inlong.apache.org

代码：https://github.com/apache/inlong

Apache InLong 公众号

Apache InLong 交流群 5

该二维码 7 天内 (9月2日前) 有效，重新进入将更新