FLINK FORWARD # ASIA 2022 ONLINE | 实时即未来

PyFlink 最新进展解读及典型应用场景介绍

付典 | 阿里巴巴高级技术专家、Apache Flink PMC & Committer

01 PyFlink 发展现状介绍

02 PyFlink 最新功能解读

03 PyFlink 典型应用场景介绍

04 PyFlink 下一步发展规划

01 PyFlink 发展现状介绍

PyFlink 的功能及性能情况

功能 & 性能全面生产可用！

PyFlink 的版本数及用户量

1.9 – 1.16 累计 **8** 个大版本，**26** 个小版本

1.9 – 1.11: Python Table API
1.12 – 1.14: Python DataStream API
1.15 – 1.16: Python 运行时优化，引入线程执行模式

JIRA 数 & Commits 数：**1000**+

代码行数：**13W**+

PyPI 下载数：400+ -> 2000+（过去一年日均下载量变化）

02 PyFlink 最新功能解读

PyFlink 最新功能解读

功能全面对齐 Java API

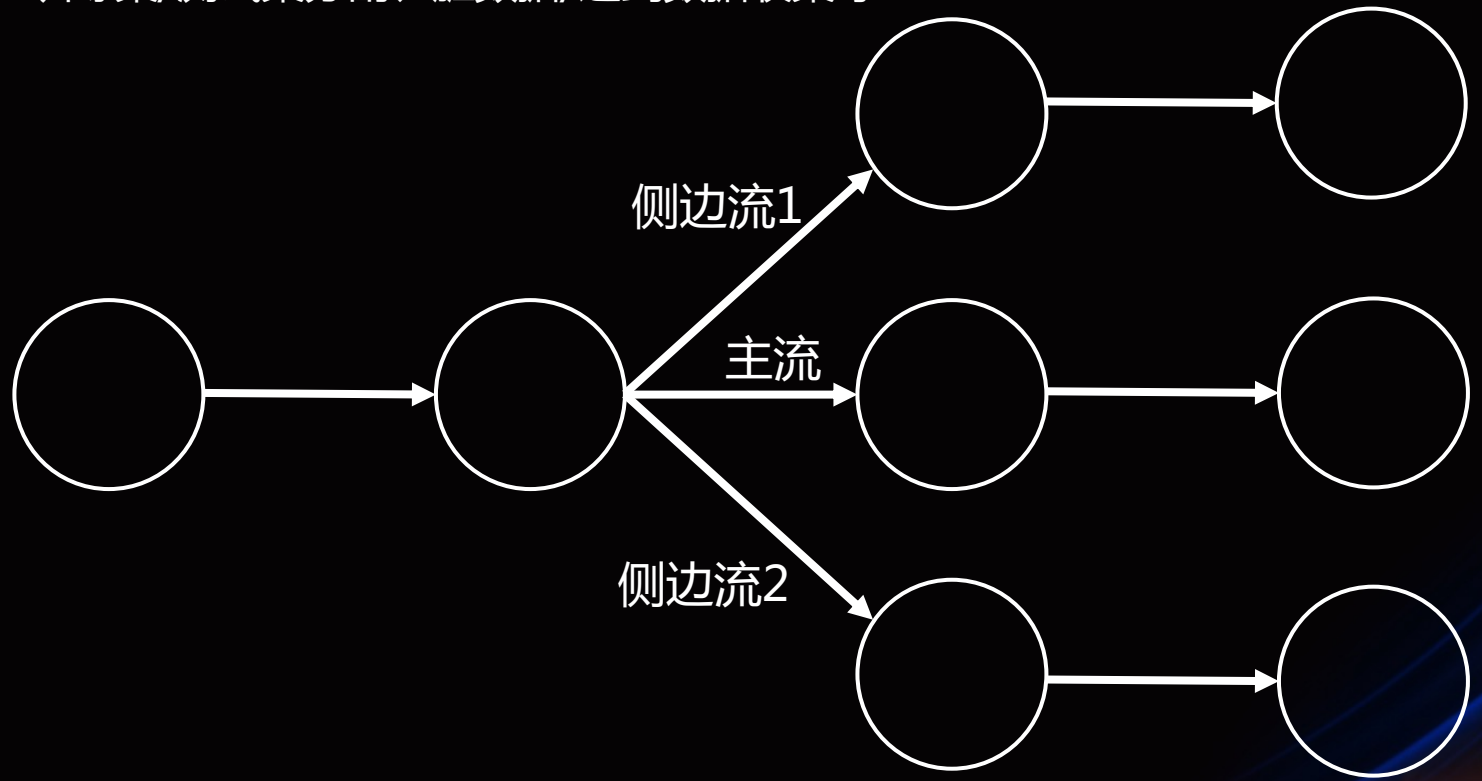
支持所有内置connector

完成了线程模式的支持

补齐最后几处短板，功能 & 性能全面生产可用！

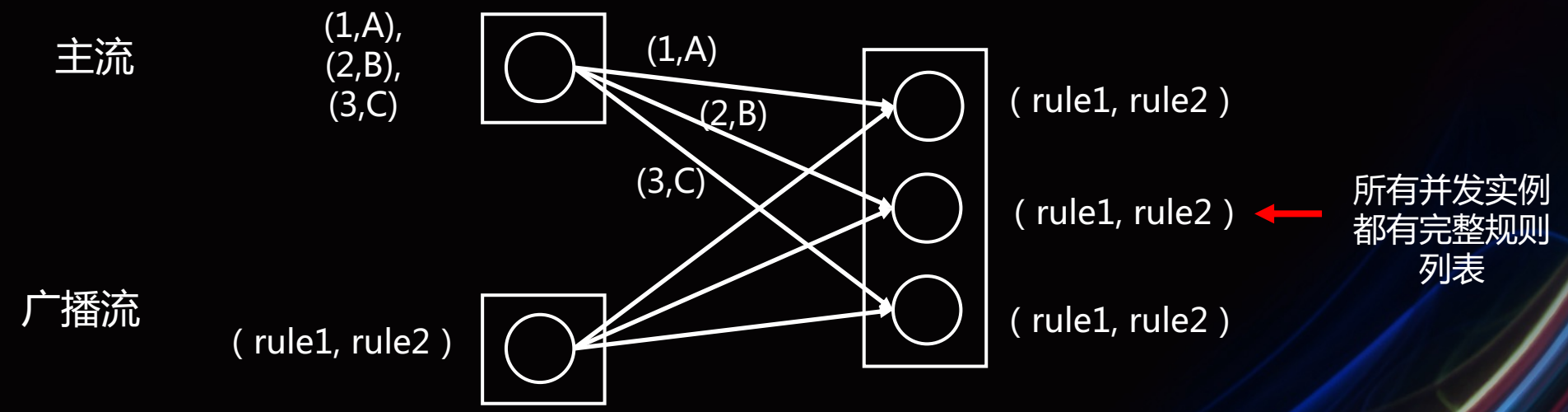
功能全面对齐 Java API

- 支持侧边流(side output)
  - 功能描述：将一条数据流经过一定的计算之后，切分成多条数据流
  - 使用场景：训练集/测试集分割、脏数据/迟到数据收集等



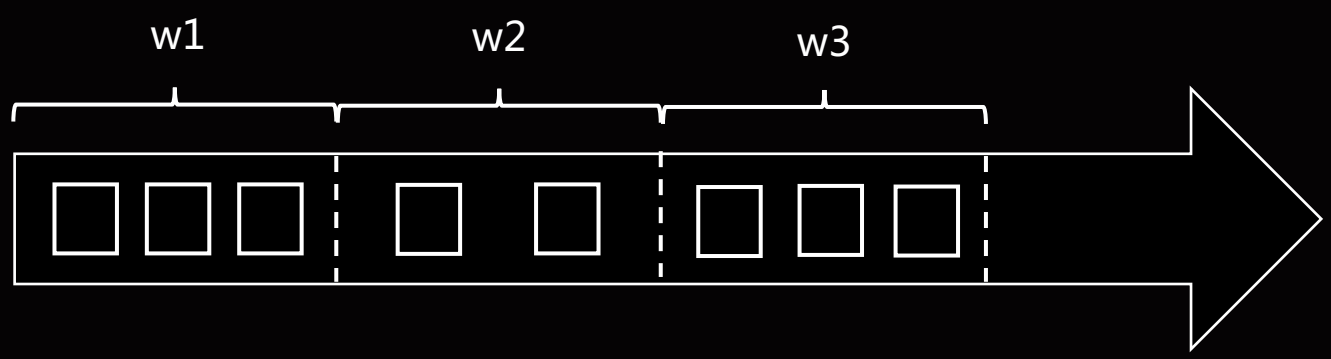
功能全面对齐 Java API

- 支持 broadcast state
  - 功能描述：将一条流中的数据广播发送到另外一条流的算子的所有并发实例，并通过 broadcast state 保存广播流的状态
  - 使用场景：机器学习模型动态更新、规则库动态更新等



功能全面对齐 Java API

- 完成对于 DataStream API 上 window 的支持
  - 功能描述：将无限流中的数据划分成不同的时间窗口进行计算
  - 使用场景：实时特征计算等



特征1:最近 5 分钟用户的视频有效观看列表

特征2:最近 30 分钟某个视频在各种人群中的点击分布

支持所有内置的 connector

- 新增了对于 Elasticsearch、Kinesis、Pulsar、Hybrid source 等的支持
- 新增了对于 Orc、Parquet 等 format 的支持

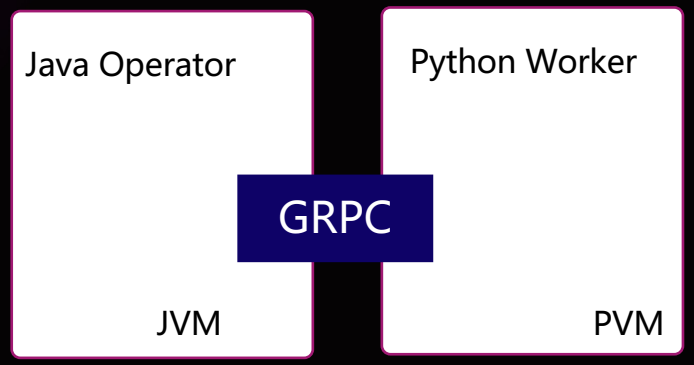
基本完成对于线程执行模式的支持

- 完成了对于 DataStream API 及 SQL 上 Python 表值函数的支持
- 线程执行模式 VS 进程执行模式
  - 性能更好：通过 JNI 调用的方式执行 Python 代码，节省序列化/反序列化开销及通信开销
  - 延迟更低：同步执行，没有攒批延迟，适用于对延迟敏感的作业，比如量化交易等场景

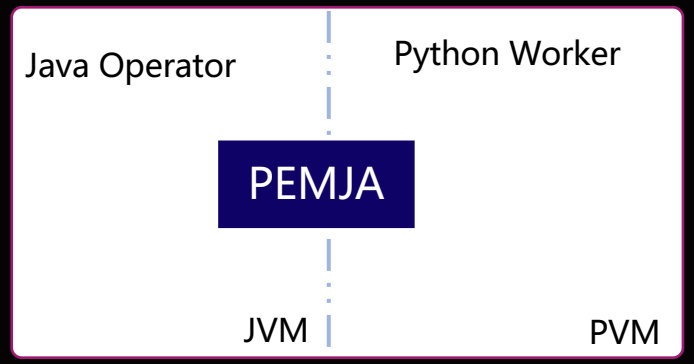
PyFlink 的功能及性能情况

进程模式 VS 线程模式

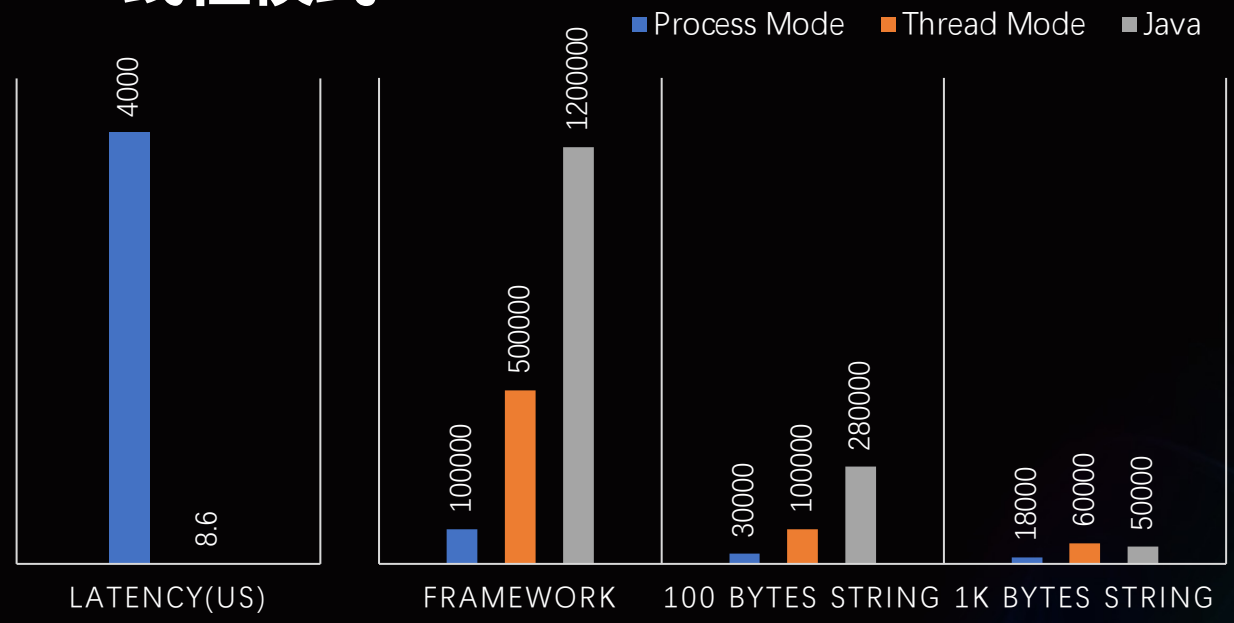
进程模式



线程模式



- 框架开销极小，性能更好
- 单条同步执行，延迟更低



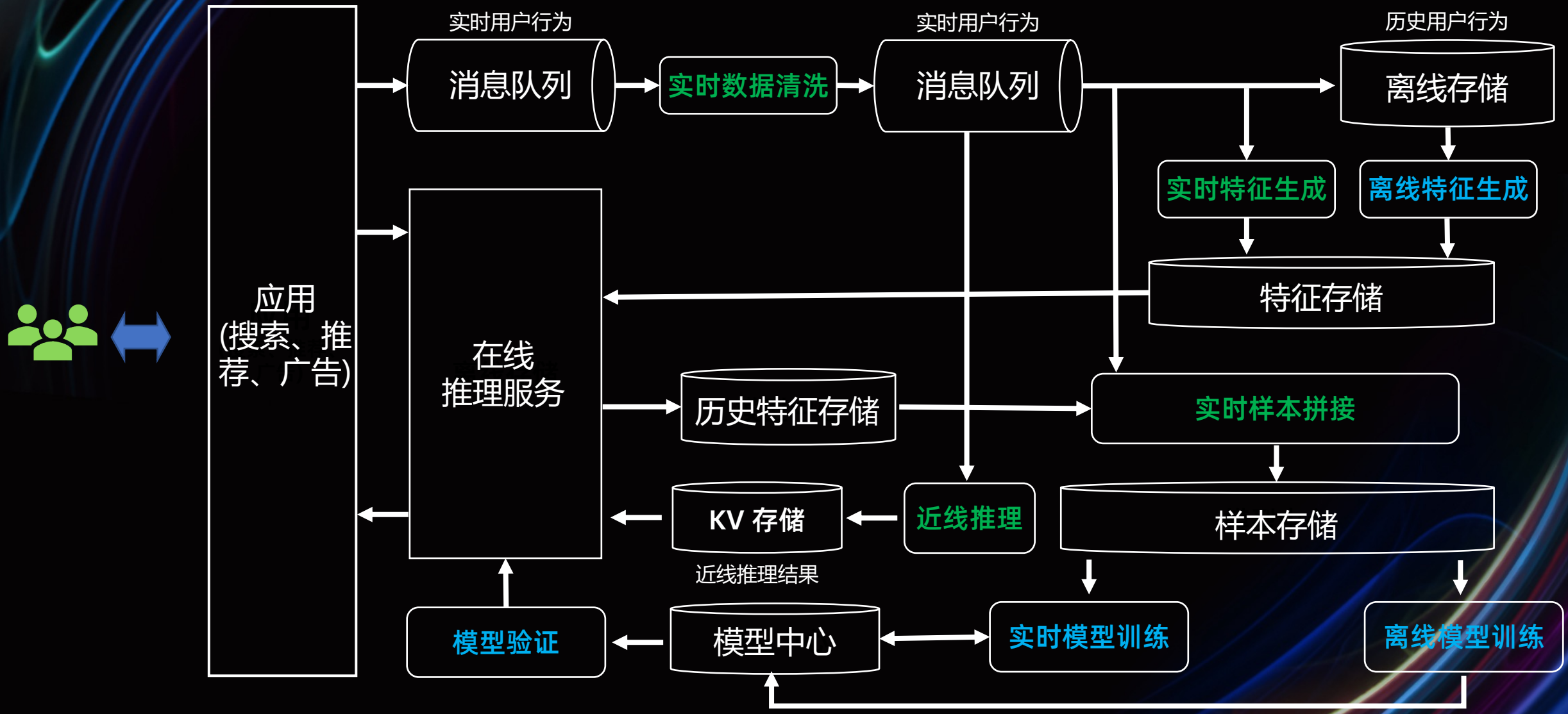
处理时延
（越小越好）

处理性能（JSON计算）
（越大越好）

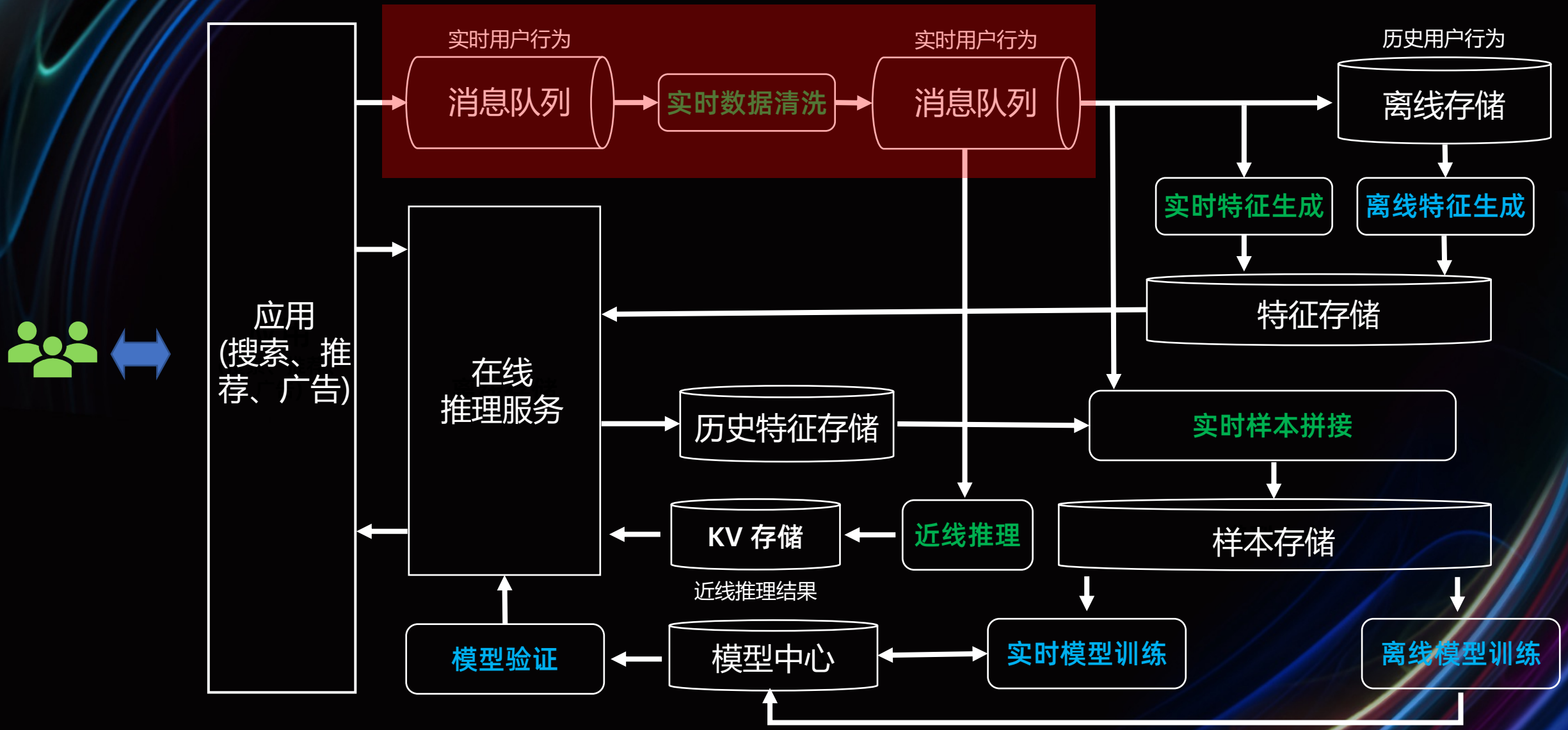
处理延迟降低百倍，吞吐能力提升数倍

03 PyFlink 典型应用场景介绍

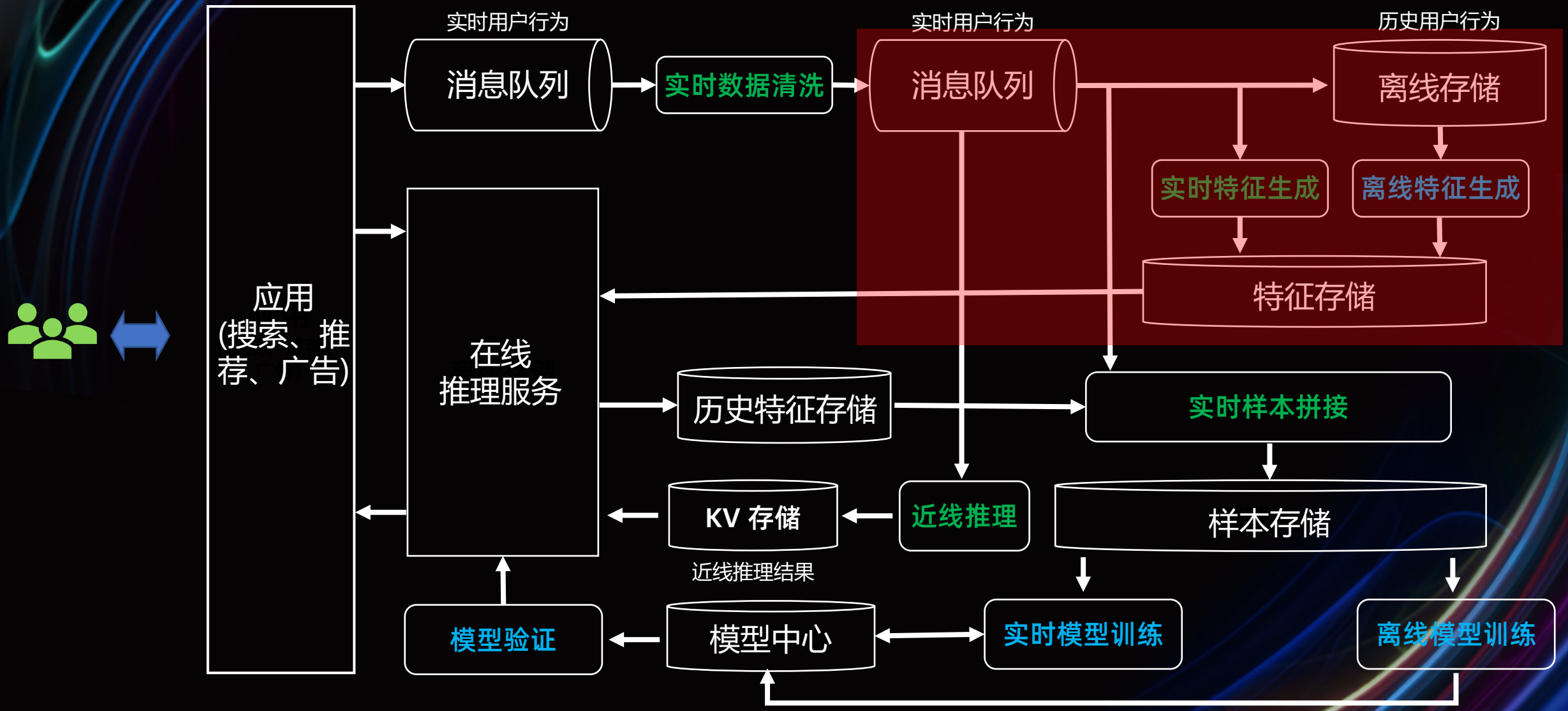
实时机器学习场景


应用
(搜索、推荐、广告)
在线
推理服务
实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
历史特征存储
实时样本拼接
KV 存储
近线推理
样本存储
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

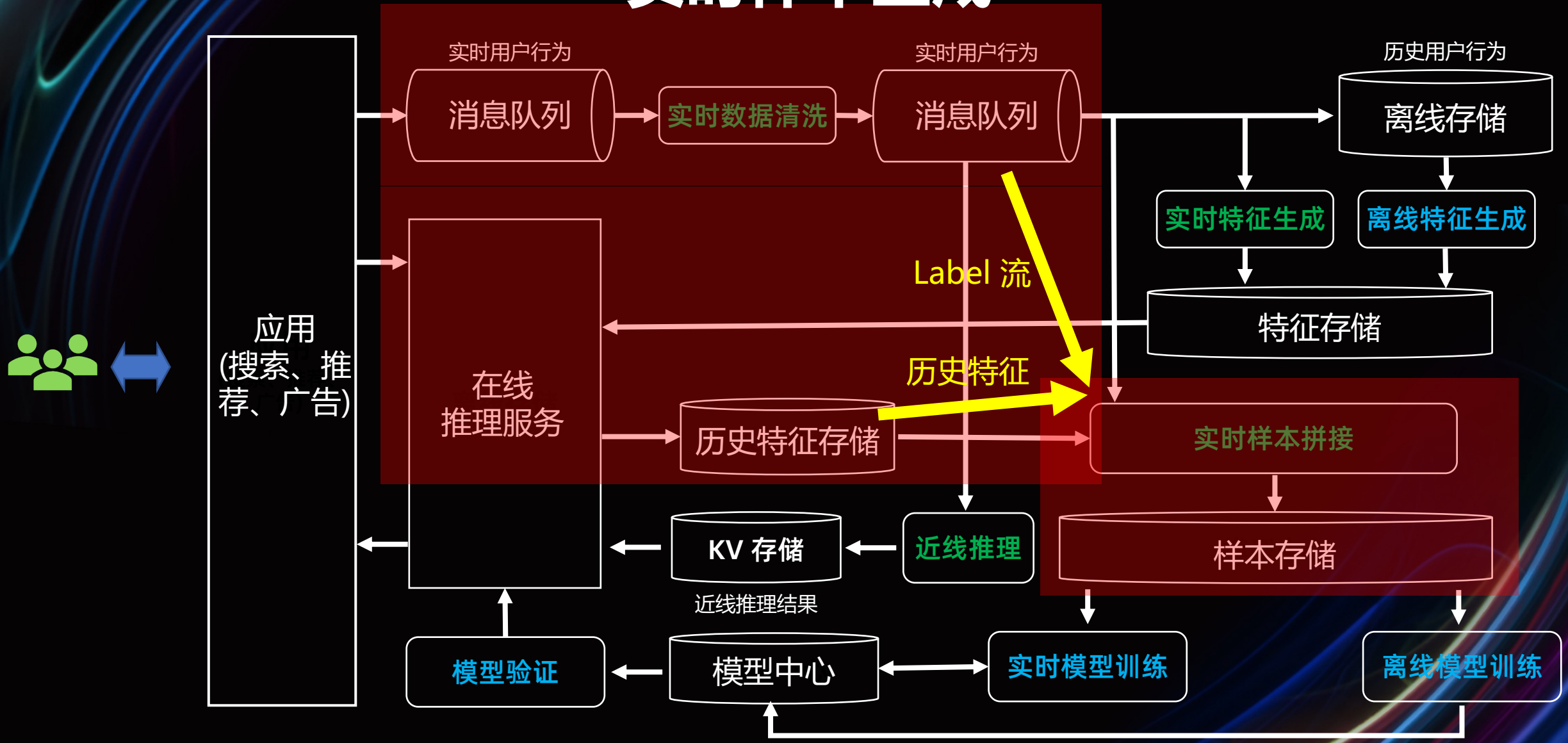
实时数据清洗


实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
应用
(搜索、推荐、广告)
在线
推理服务
历史特征存储
实时样本拼接
KV 存储
近线推理
样本存储
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

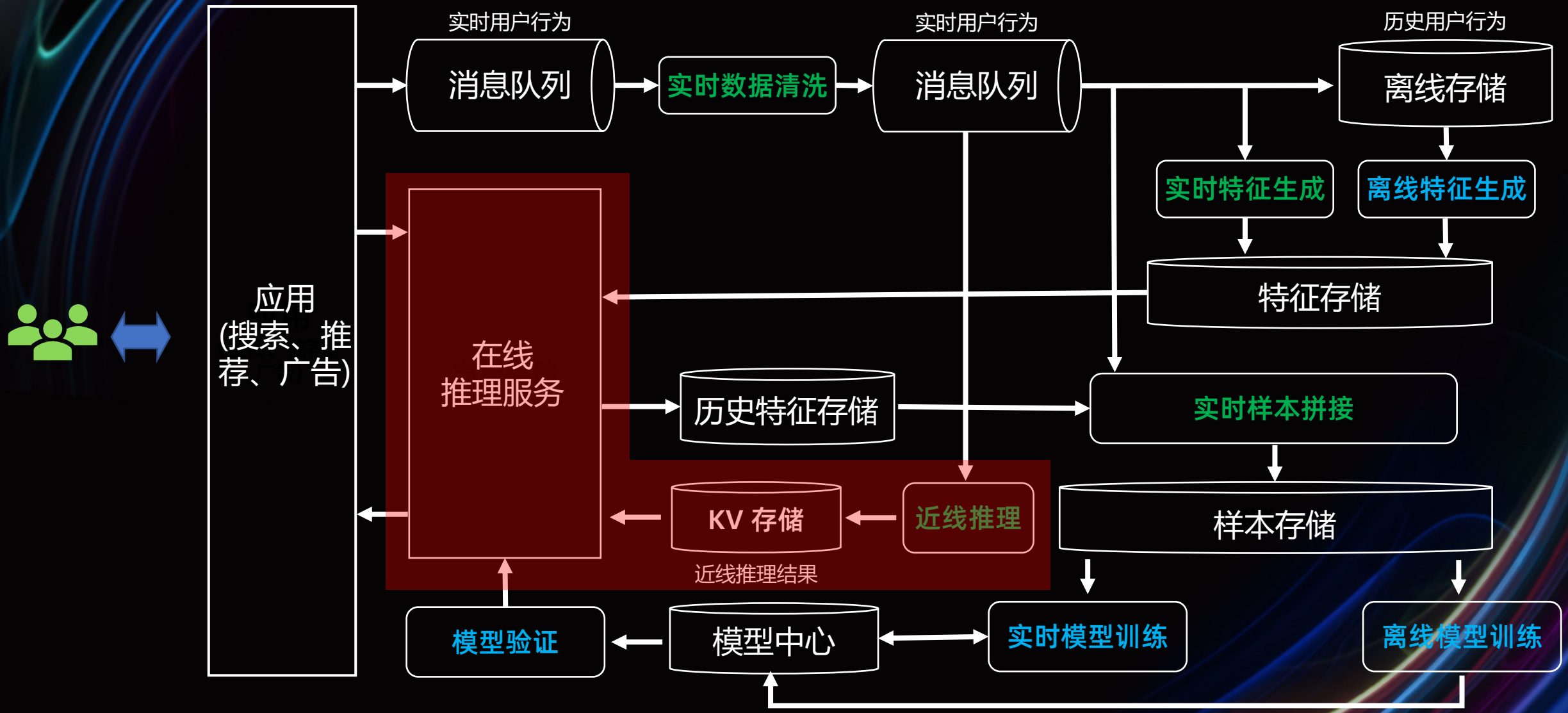
实时特征计算


应用
(搜索、推荐、广告)
在线
推理服务
实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
历史特征存储
实时样本拼接
KV 存储
近线推理
样本存储
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

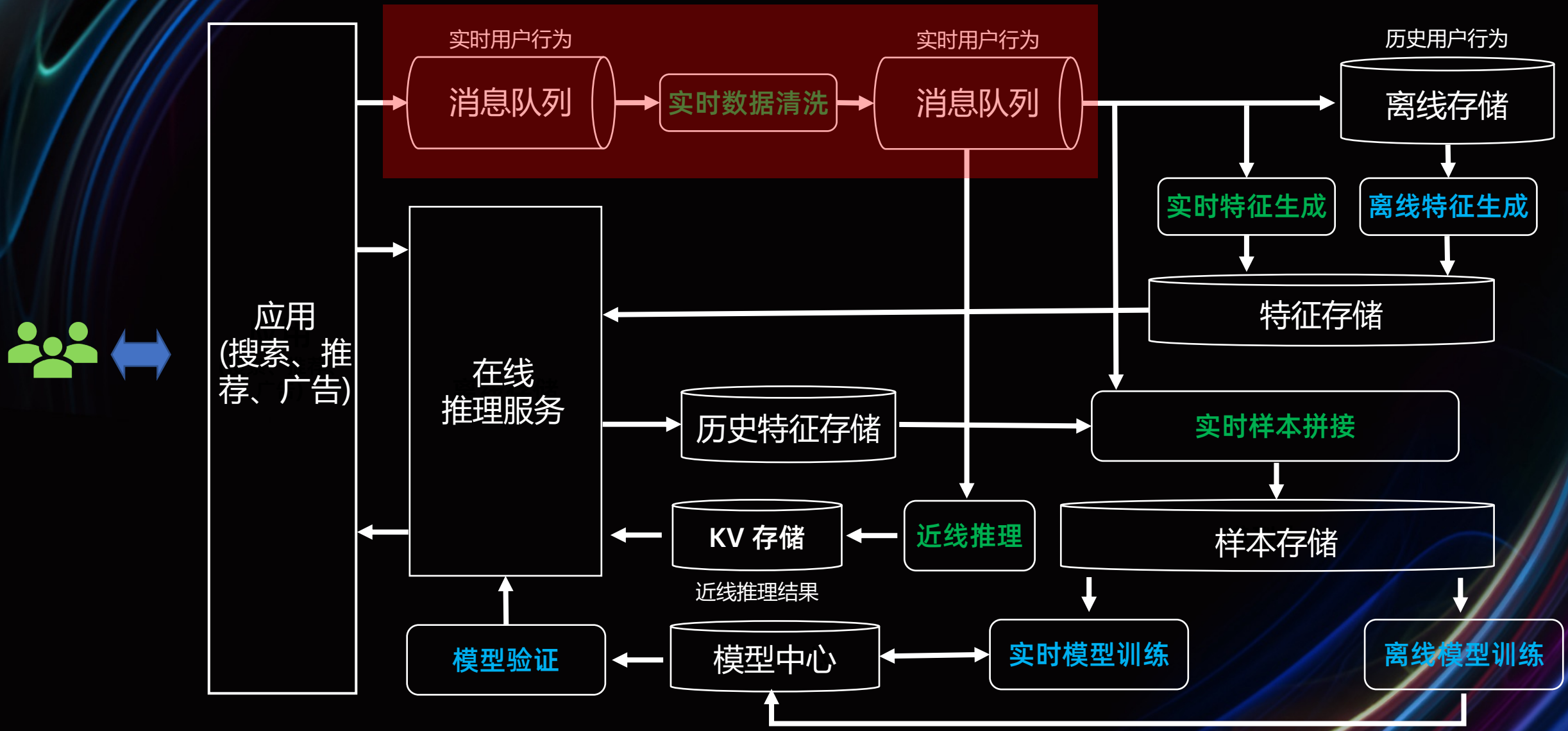
实时样本生成


实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
Label 流
历史特征
应用
(搜索、推荐、广告)
在线
推理服务
历史特征存储
实时样本拼接
样本存储
KV 存储
近线推理
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

近线推理


应用
(搜索、推荐、广告)
在线
推理服务
实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
历史特征存储
实时样本拼接
KV 存储
近线推理
近线推理结果
样本存储
模型验证
模型中心
实时模型训练
离线模型训练

实时数据清洗


实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
应用
(搜索、推荐、广告)
在线
推理服务
历史特征存储
实时样本拼接
KV 存储
近线推理
样本存储
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

实时数据清洗

Why PyFlink?

- 机器学习应用中，输入数据中往往包含很多列

```
SELECT col1, col2, col3, col4, col5, col6, col7, col8, my_udf(col9, col10), col11, col12
FROM input_table
```

痛点：输入列很多的情况下，SELECT 语句很长

- 机器学习用户，习惯使用 Pandas 库进行数据处理

痛点：默认情况下，Flink SQL 中，UDF 以行为单位对数据进行转换，
而 Pandas 库以列存的方式组织及操作数据

实时数据清洗

行操作 & 列操作

```
src_table = ...

# 过滤无效数据
@udf(result_type=DataTypes.BOOLEAN())
def filter_price(price):
    return price > 0

filtered_table = src_table.filter(filter_price(col('price')))

# 列操作
# 数据归一化处理
@udf(result_type=DataTypes.BIGINT())
def normalize(item_id):
    if item_id is None:
        return -1
    return item_id

normalized_table = filtered_table.add_or_replace_columns(
    normalize(src_table.item_id).alias('item_id'))

# 行操作
# 数据归一化处理
@udf(result_type=DataTypes.ROW(
    [DataTypes.FIELD("price", DataTypes.FLOAT()),
     DataTypes.FIELD("item_id", DataTypes.BIGINT())]))
def row_based_normalize(value: Row) -> Row:
    if value.item_id is None:
        value['item_id'] = -1
    return value

normalized_table = filtered_table.map(row_based_normalize)
```

列操作：

- 功能：增加列、删除列、替换列等
- 操作的对象是列
- 适用于输入数据的列很多，且只有个别列发生变化的场景

行操作：

- 功能：对数据以行为单位进行变换
- 操作的对象是行，可以直接在 UDF 实现中通过列名引用对应列
- 适用于输入数据的列很多，且需要对多个列进行处理的场景

实时数据清洗

Pandas UDF

```
src_table = ...

# 数据归一化处理
@udf(result_type=DataTypes.ROW(
        [DataTypes.FIELD("price", DataTypes.FLOAT()),
         DataTypes.FIELD("item_id", DataTypes.BIGINT())]),
     func_type="pandas")
def normalize(data: pd.DataFrame) -> pd.DataFrame:
    data['item_id'].fillna(-1, inplace=True)
    return data

filtered_table = src_table.map(normalize)
```

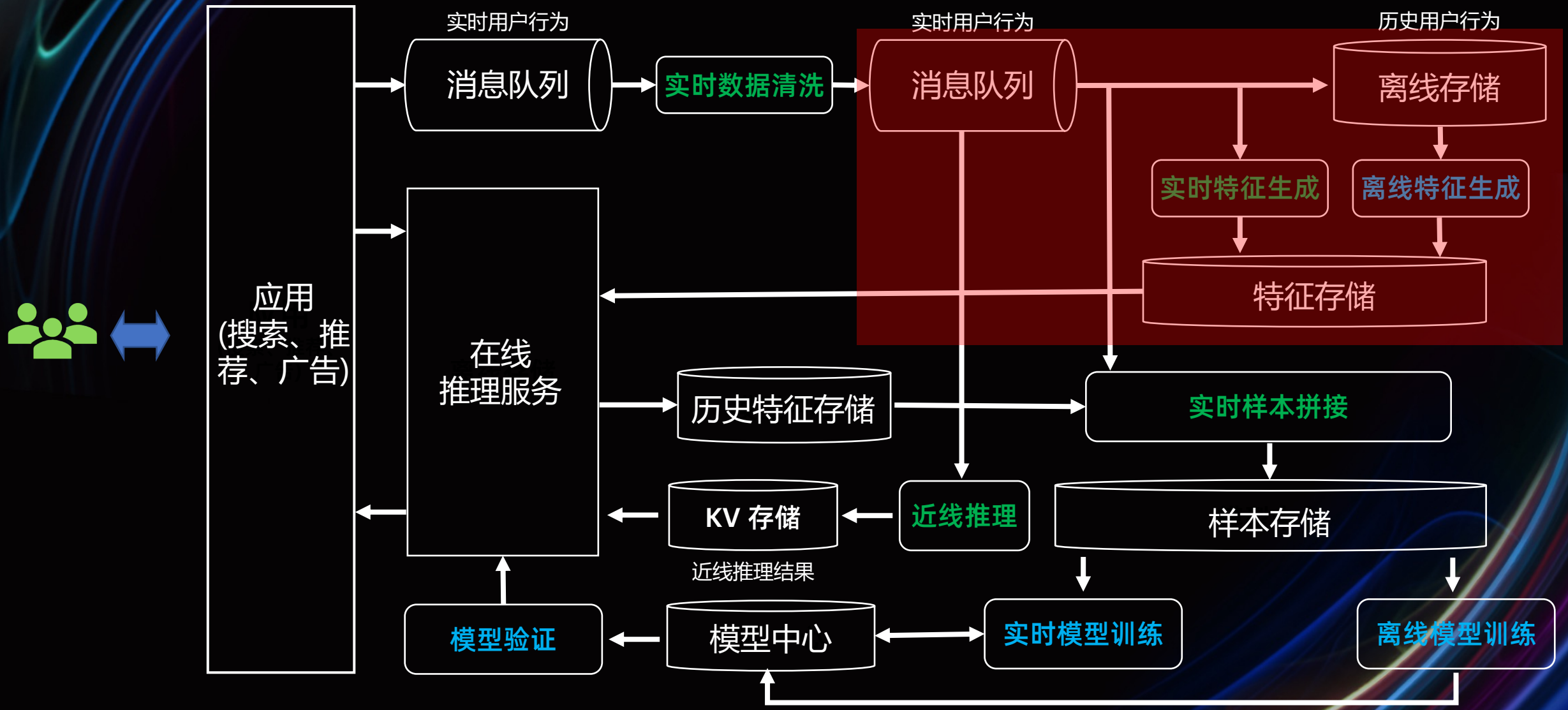
输入

输出

Python UDF 的输入/输出都是 pandas 数据结构

UDF 实现可以使用高性能的 Pandas 库

实时特征计算


应用
(搜索、推荐、广告)
在线
推理服务
实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
历史特征存储
实时样本拼接
KV 存储
近线推理
样本存储
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

实时特征计算

Why PyFlink?

痛点：沟通成本高、不灵活、特征上线周期长

- 降低实时特征任务的开发门槛、缩短实时特征的上线周期

  以前：提需求（算法团队） -> 需求沟通 -> 特征任务开发（数据团队） -> 特征验证（算法团队） -> 特征任务上线/废弃

  现在：特征任务开发（算法团队） -> 特征校验（算法团队） -> 特征任务上线/废弃

- 特征计算的过程中，经常需要使用各种 Python 库，比如 jieba、OpenCV、 sklearn 等

实时特征计算

示例：电商推荐场景中，计算用户最近
5分钟的访问物品列表（序列特征）

实时特征计算

通过 SQL + Pandas UDAF 计算序列特征

Pandas UDAF（除此之外，也支持普通 Python UDAF）

```
SELECT user, accessed_item_seq(accessed_time, accessed_item)
FROM src_table
GROUP BY user, HOP(event_time, INTERVAL '30' SECOND, INTERVAL '5' MINUTE)
```

滑动窗口（除此之外，也支持滚动窗口、会话窗口等）

Pandas UDAF 的输入是 pandas 数据结构，输出是普通 Python 对象

```
@udaf(result_type=DataTypes.STRING(), func_type="pandas")
def accessed_item_seq(accessed_time: pd.Series, accessed_item: pd.Series) -> str:
    import pandas as pd
    df = pd.DataFrame({'ts': accessed_time, 'items': accessed_item})
    latest_5_df = df.sort_values(by="ts" , ascending=True).tail(5)
    return '|'.join(latest_5_df['items'])
```

计算序列特征

实时特征计算

```
ds = ...  # (user_id, ts, item_id)
ds.key_by(lambda x: x.user_id) \
  .window(SlidingEventTimeWindows.of(Time.minutes(5), Time.seconds(30))) \
  .aggregate(AccessedItemSeqAggregate(), output_type=Types.STRING()) \
  .sink_to(xxx)
```

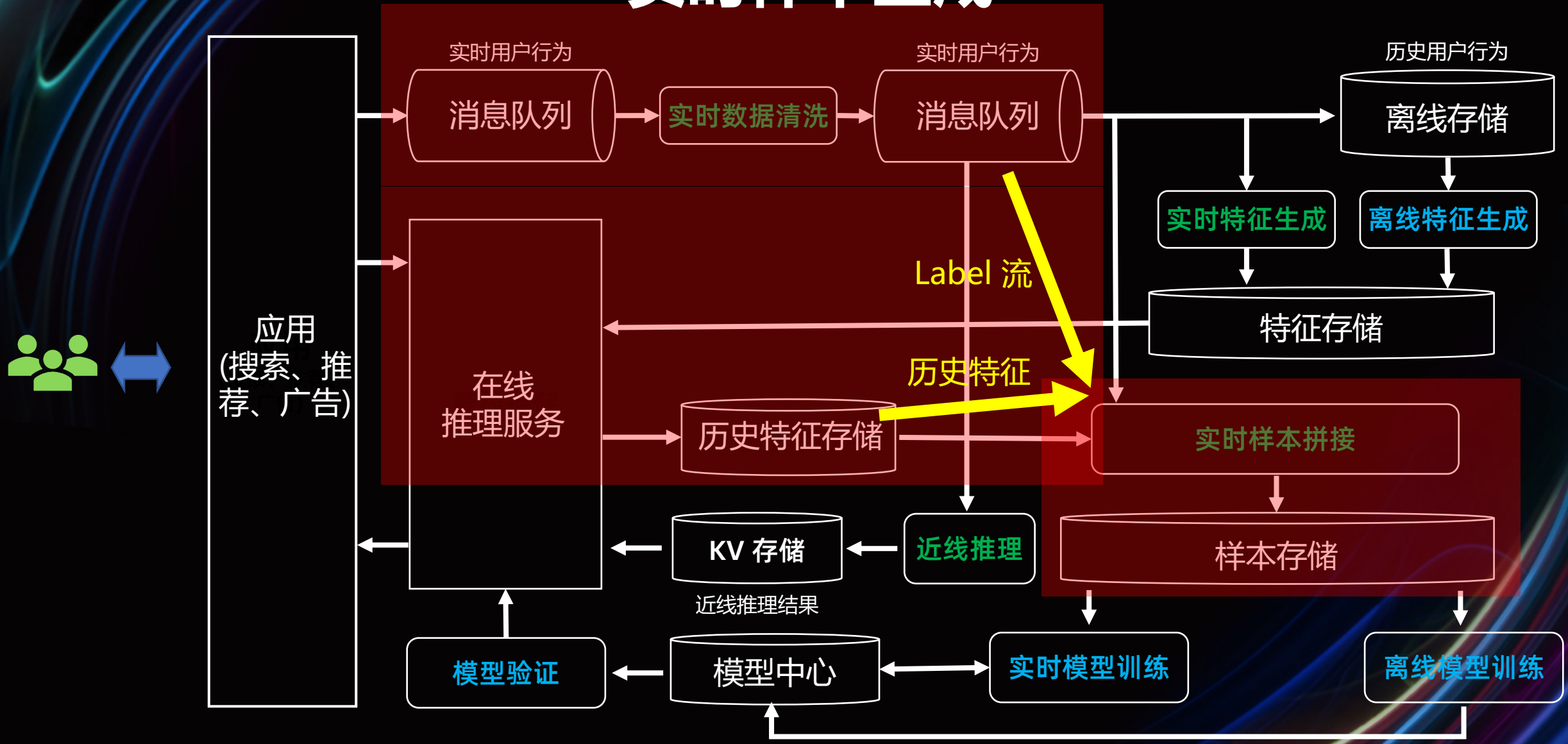
通过 DataStream API 计算序列特征

滑动窗口（除此之外，还支持滚动窗口、会话窗口）

```
class AccessedItemSeqAggregate(AggregateFunction):
    def create_accumulator(self) -> pd.DataFrame:
        return pd.DataFrame(columns=['ts', 'items'])

    def add(self, value: Tuple[str, int, str], accumulator: pd.DataFrame) -> pd.DataFrame:
        return df.append({'ts': value[1], 'items': value[2]})

    def get_result(self, accumulator: pd.DataFrame) -> str:
        latest_5_df = accumulator.sort_values(by="ts" , ascending=True).tail(5)
        return '|'.join(latest_5_df['items'])

    def merge(self, a: pd.DataFrame, b: pd.DataFrame) -> pd.DataFrame:
        return a.append(b, ignore_index=True)
```

计算序列特征

实时样本生成


实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
Label 流
历史特征
应用
(搜索、推荐、广告)
在线
推理服务
历史特征存储
实时样本拼接
样本存储
KV 存储
近线推理
近线推理结果
模型验证
模型中心
实时模型训练
离线模型训练

实时样本生成

Label 生成

```
src_datastream = ...
rst_datastream = src_datastream.key_by(lambda x: (x.user_id, x.item_id).process(LabelGenerator())
rst_datastream.sink_to(result_sink)
```

```
class LabelGenerator(KeyedProcessFunction):
    def open(self, runtime_context):
        self.input_state = runtime_context.get_state(
            ValueStateDescriptor("input_element", input_type))
        self.click_timer_state = runtime_context.get_state(
            ValueStateDescriptor("click_timer", Types.LONG()))

    def process_element(self, input_value, ctx: 'KeyedProcessFunction.Context'):
        if input_value.type == 'expose':
            self.input_state.update(input_value)
            # register the click timer
            click_timer = ctx.timestamp() + 600000
            self.click_timer_state.update(click_timer)
            ctx.timer_service().register_event_time_timer(click_timer)

        if input_value.type == 'click':
            # cancel the click timer
            click_timer = self.click_timer_state.value()
            if click_timer is not None:
                ctx.timer_service().delete_event_time_timer(click_timer)
            yield Row(*input_value, label=1)

    def on_timer(self, timestamp: int, ctx: 'KeyedProcessFunction.OnTimerContext'):
        input_value = self.input_state.value()
        yield Row(*input_value, label=0)
```

通过定时器解决正负样本问题

若定时器触发前，收到点击事件，生成正样本

若定时器触发，生成负样本

实时样本生成

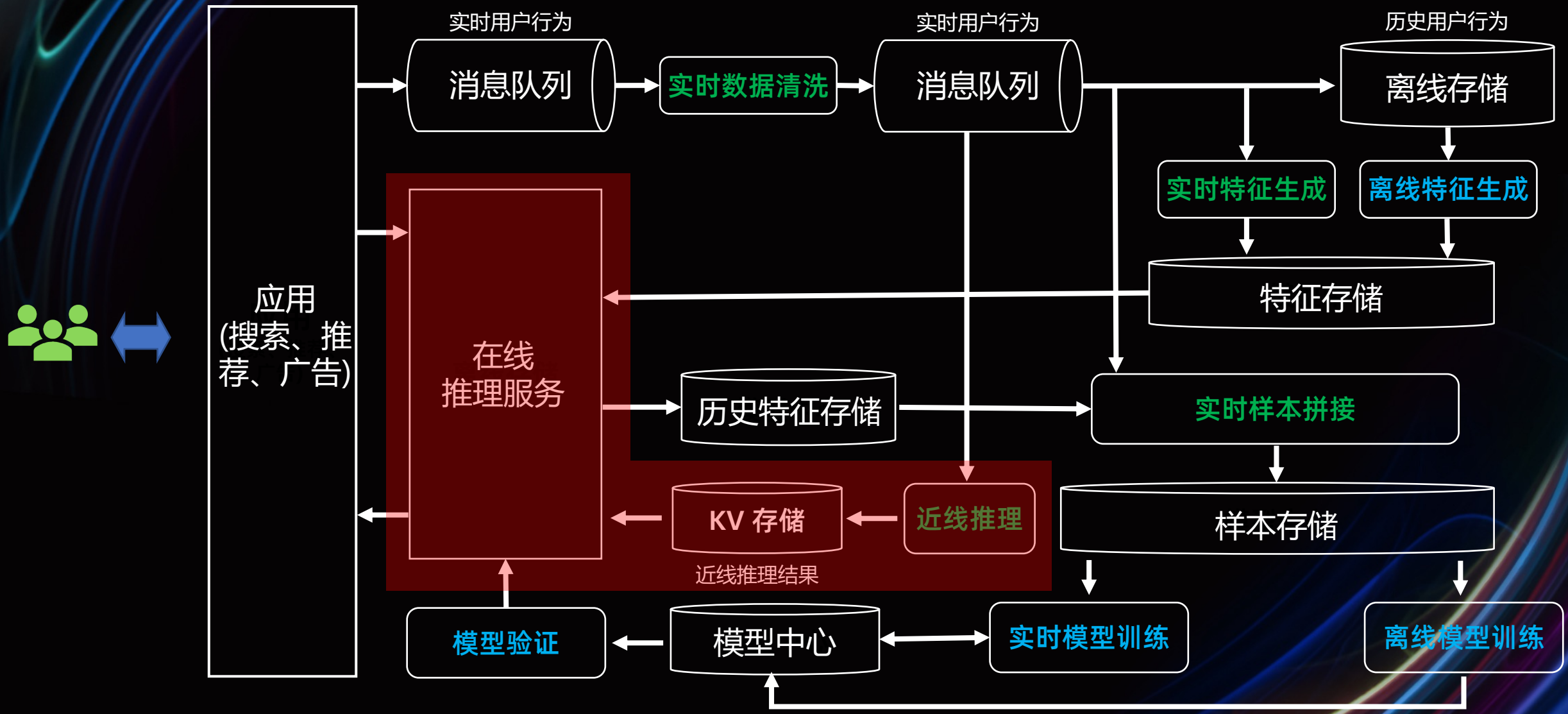
样本拼接

```
table_env = ...
label_table = ...
feature_table = ...

table_env.execute_sql("""
    SELECT *
    FROM %s AS label_table
    JOIN %s FOR SYSTEM_TIME AS OF label_table.proc_time AS feature_table
    ON label_table.user_id == feature_table.user_id
      && label_table.item_id == feature_table.item_id
      && label_table.trace_id == feature_table.trace_id
""" % (label_table, feature_table))
```

维表 Join：label 流 Join 特征库（KV 存储）

通过 trace_id 解决特征穿越问题

近线推理


应用
(搜索、推荐、广告)
在线
推理服务
实时用户行为
消息队列
实时数据清洗
实时用户行为
消息队列
历史用户行为
离线存储
实时特征生成
离线特征生成
特征存储
历史特征存储
实时样本拼接
KV 存储
近线推理
近线推理结果
样本存储
模型验证
模型中心
实时模型训练
离线模型训练

近线推理

```
src_table = ...
rst_table = src_table.select(predict(col('input_a'), col('input_b')))
rst_table.execute_insert("result_sink")
```

通过 Table API 进行近线推理

```
class Predict(ScalarFunction):
    def open(self, function_context):
        self.model = load_model("/path/to/model")

    def eval(self, input_a, input_b):
        return self.model.predict(input_a, input_b)

predict = udf(Predict(), result_type=DataTypes.DOUBLE())
```

模型加载

预测

近线推理

```
src_datastream = ...
rst_datastream = src_datastream.map(Predict(), output_type=Types.STRING())
rst_datastream.sink_to(result_sink)
```

通过 DataStream API 进行近线推理

```
class Predict(MapFunction):
    def open(self, runtime_context):
        self.model = load_model("/path/to/model")

    def map(self, input_value):
        return self.model.predict(input_value)
```

← 模型加载

← 预测

近线推理

通过 timer 提升推理的时效性

```
src_datastream = ...
rst_datastream = \
    src_datastream.key_by(lambda x: x.user_id) \
                  .process(PeriodicPredict(), output_type=Types.STRING())
rst_datastream.sink_to(result_sink)

class PeriodicPredict(KeyedProcessFunction):
    def open(self, runtime_context):
        self.model = load_model("/path/to/model")
        self.input_state = runtime_context.get_state(
                    ValueStateDescriptor("input_element", input_type))
        self.timer_state = runtime_context.get_state(
                    ValueStateDescriptor("timer", Types.LONG()))

    def process_element(self, input_value, ctx: 'KeyedProcessFunction.Context'):
        # cancel the current timer
        cur_timer = self.timer_state.value()
        if cur_timer is not None:
            ctx.timer_service().delete_event_time_timer(cur_timer)

        # register the next timer
        next_timer = ctx.timestamp() + 300000
        self.timer_state.update(next_timer)
        ctx.timer_service().register_event_time_timer(next_timer)

        self.input_state.update(input_value)
        yield self.model.predict(input_value)

    def on_timer(self, timestamp: int, ctx: 'KeyedProcessFunction.OnTimerContext'):
        ctx.timer_service().register_event_time_timer(ctx.timestamp() + 300000)
        yield self.model.predict(self.input_state.value())
```

通过定时器进行周期性预测

预测

近线推理

通过 Flink Java 作业做近线推理

```
public class Predict extends RichMapFunction {
        private final String modelPath;
        private transient PythonInterpreter interpreter;

        public Predict(String modelPath) {
            this.modelPath = modelPath;
        }

        public void open(Configuration parameters) throws Exception {
                String pythonPath = ...;
        PythonInterpreterConfig config = PythonInterpreterConfig
            .newBuilder()
            .setPythonExec("python3") // specify python exec
            .addPythonPaths(pythonPath) // add path to search path
            .build();

        interpreter = new PythonInterpreter(config);
        interpreter.exec("model=load_model(" + this.modelPath + ")");
        }

    public String map(String input) {
        return interpreter.invokeMethod("model", "predict", input);
    }
}
```

← 模型加载

← 预测

Powered By PEMJA (https://github.com/alibaba/pemja)

04 PyFlink 下一步发展规划

PyFlink 下一步发展规划

独立的 PyFlink 网站 https://pyflink.readthedocs.io/

- 环境安装、可在线运行的QuickStart、端到端示例、常见问题
- 降低用户上手门槛

提高易用性

- 优化报错提示
- 优化 API 文档、添加 API 示例

提升稳定性

- 降低进程模式下 checkpoint 的耗时

FLINK FORWARD # ASIA 2022 ONLINE | 实时即未来

THANK YOU

谢 谢 观 看