



Flink 数据类型和序列化

马庆祥 · 360 / 数据开发

Apache Flink Community China



Apache Flink

CONTENT

目录 >>

01 /

为Flink量身定制的序列化框架

02 /

Flink序列化的最佳实践

03 /

Flink通信层的序列化

04 /

Q/A

01

为Flink量身定制的序列化框架



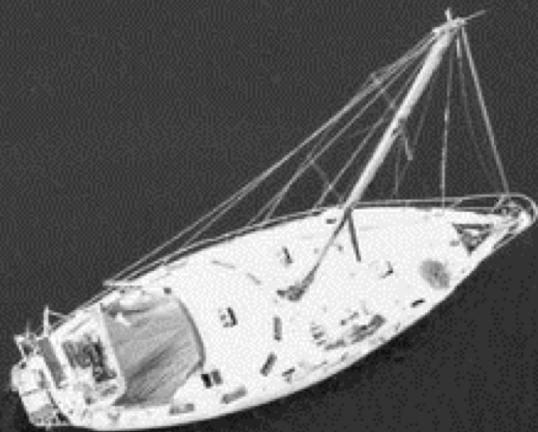
Apache Flink

为什么定制

基于JVM的数据分析引擎

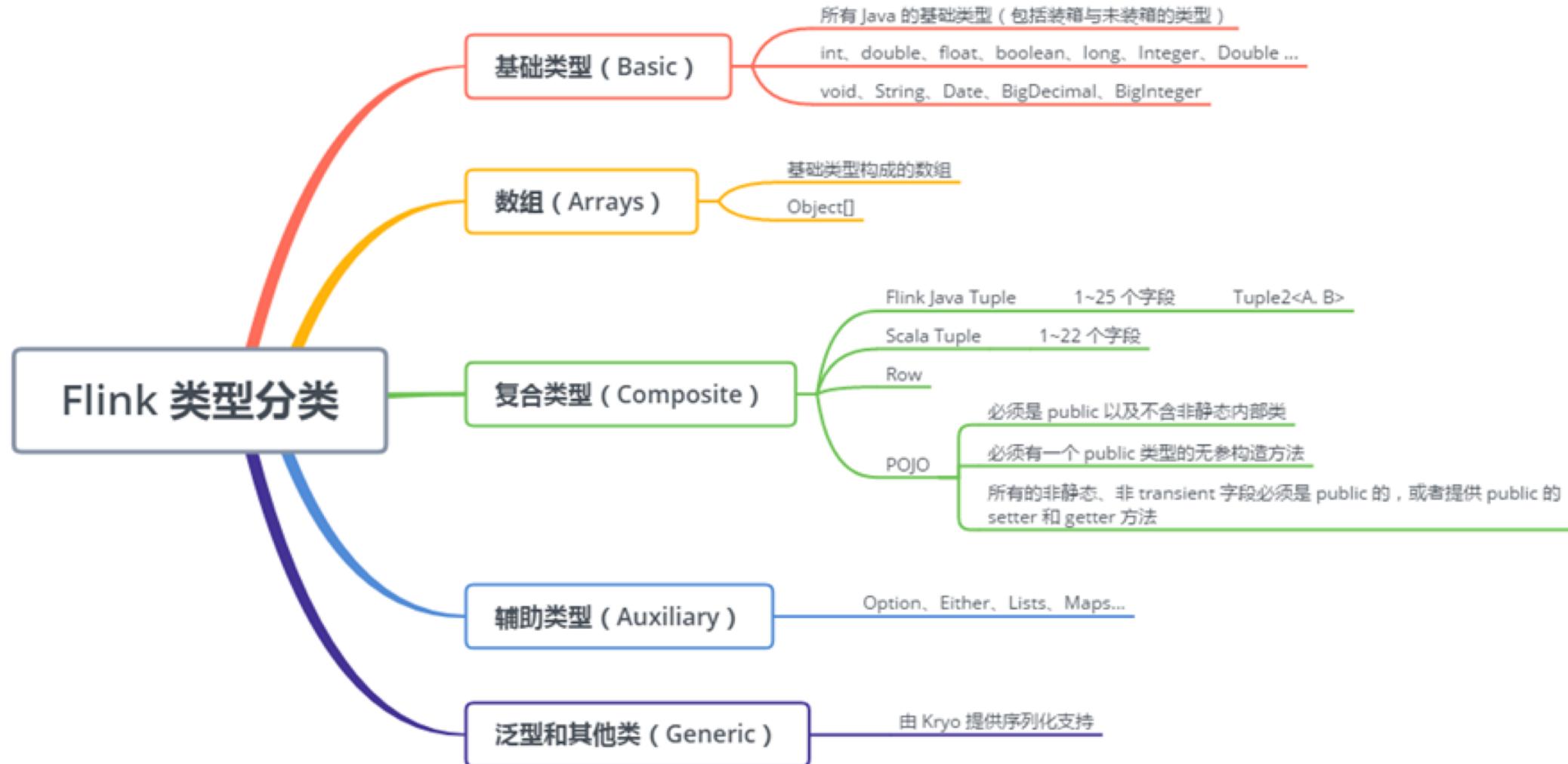
大数据时代的JVM – 显式的内存管理

为Flink量身定制的序列化框架





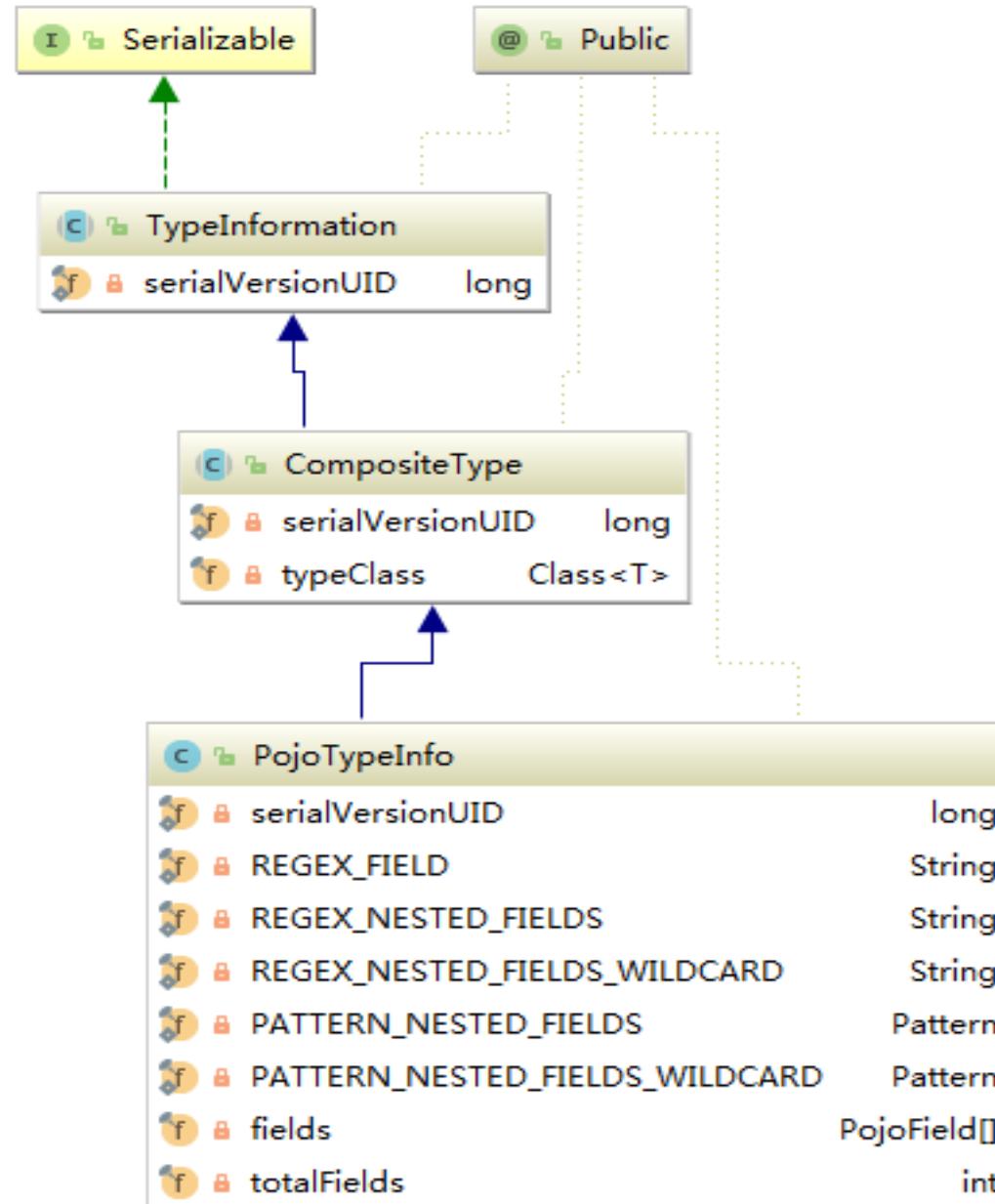
Flink的数据类型





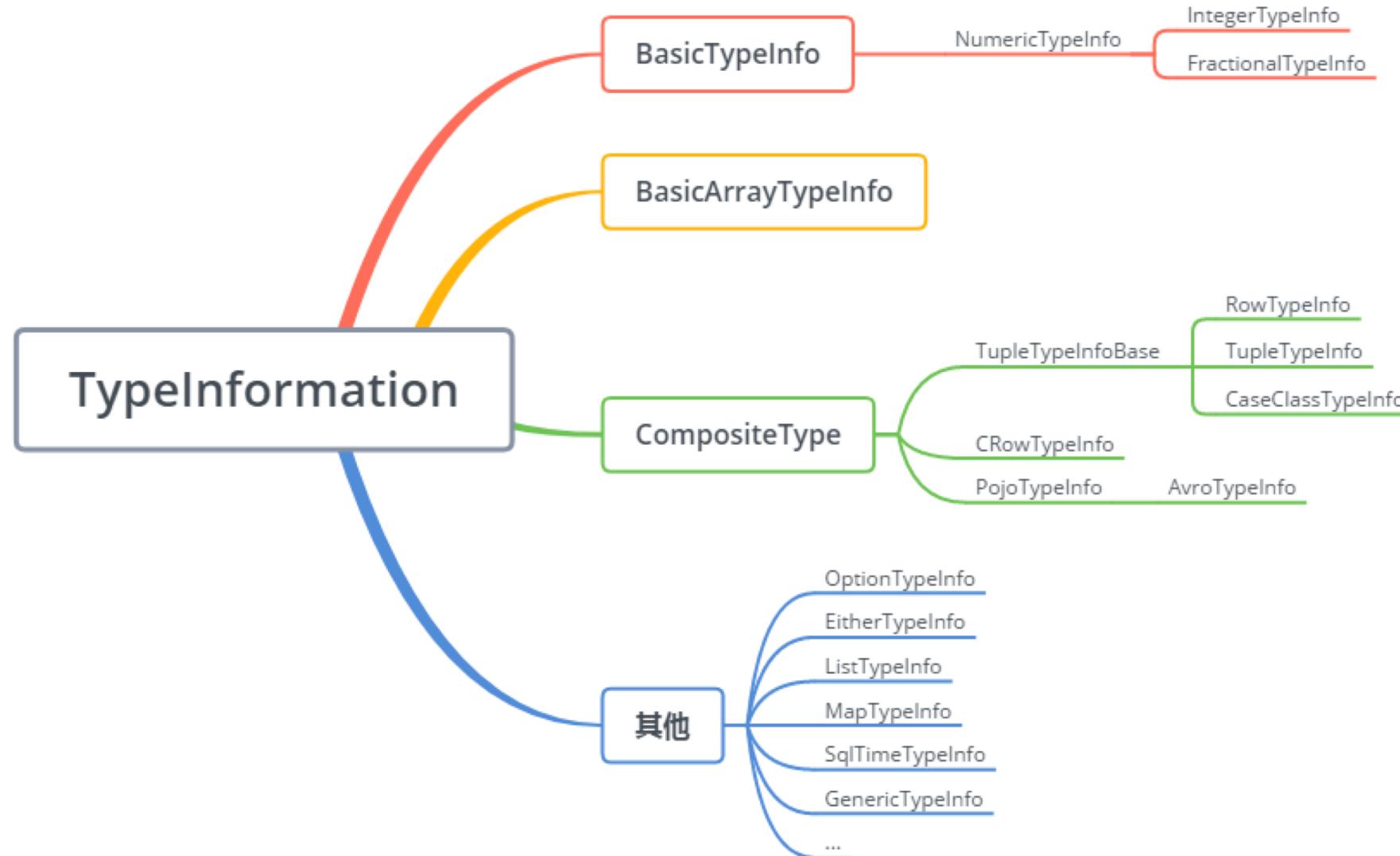
Flink的数据类型

```
public class Person {  
    public int id;  
    public String name;  
}
```





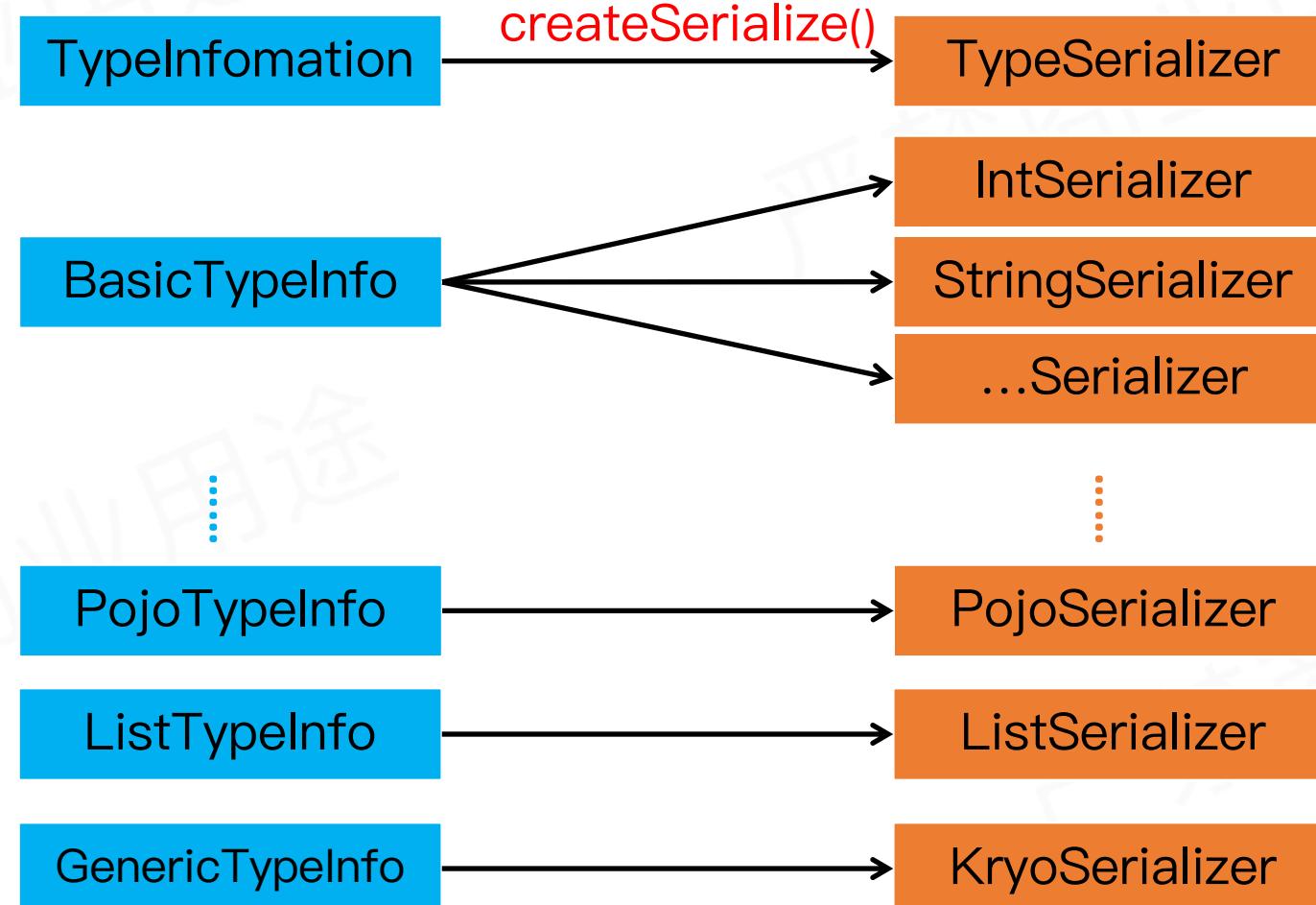
TypeInformation





Flink的序列化过程

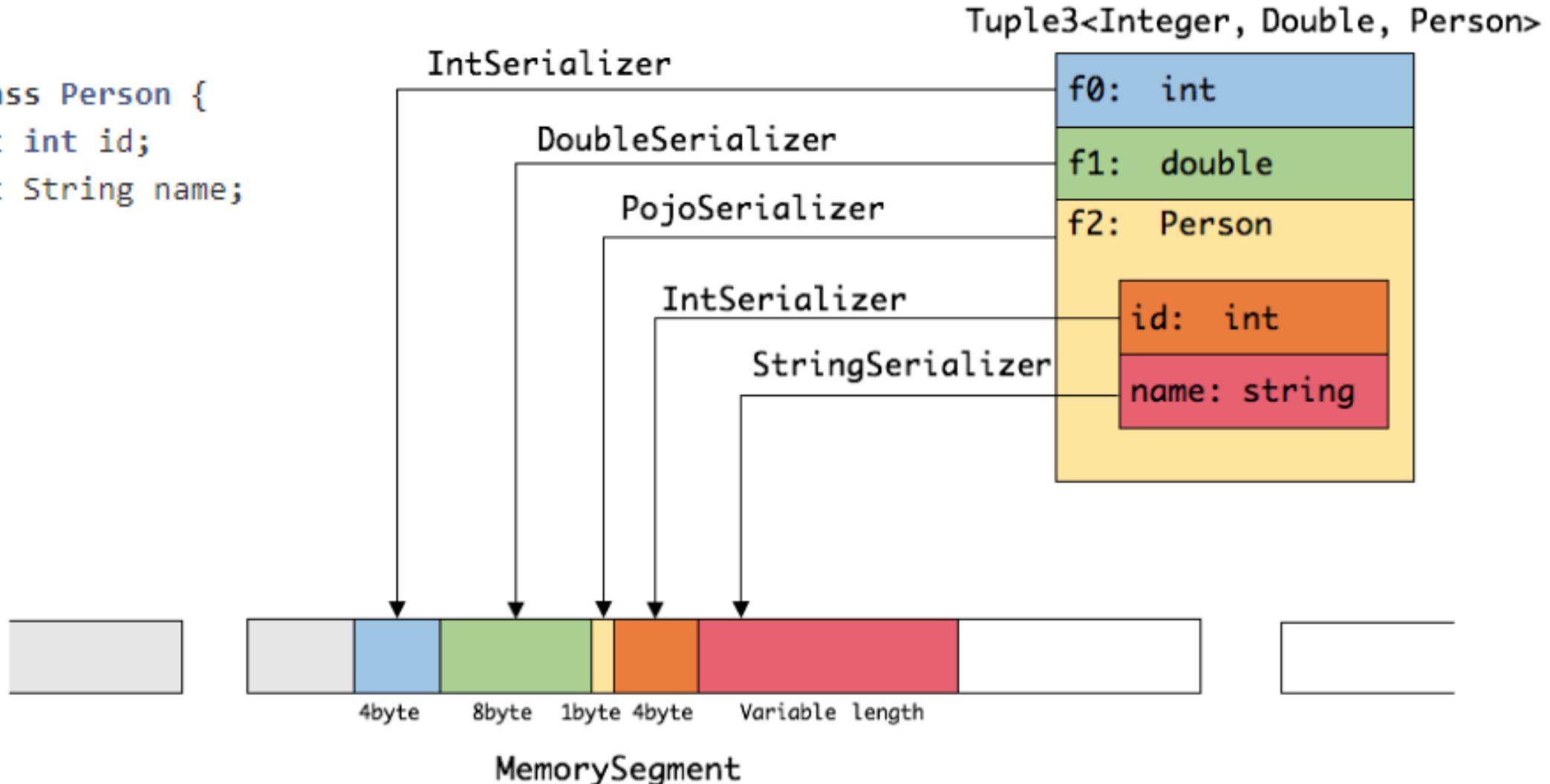
获取序列化器：





Flink的序列化过程

```
public class Person {  
    public int id;  
    public String name;  
}
```



02

Flink序列化的最佳实践

最常见的场景

- 注册子类型
- 注册自定义序列化器
- 添加类型提示
- 手动创建TypeInformation



实践 – 类型声明 (1)

通过TypeInformation.of()方法可以很方便的创建类型信息对象：

1. 对于非泛型类，直接传入class对象即可

- PojoTypeInfo<Person> typeInfo = (PojoTypeInfo<Person>) TypeInformation.of(Person.class);

2. 对于泛型类，需要通过TypeHint来保存泛型类型信息

- final TypeInformation<Tuple2<Integer, Integer>> resultType =
TypeInformation.of(new TypeHint<Tuple2<Integer, Integer>>(){});

```
this.unionOffsetStates = stateStore.getUnionListState<new ListStateDescriptor<>(  
    OFFSETS_STATE_NAME,  
    TypeInformation.of<new TypeHint<Tuple2<KafkaTopicPartition, Long>>() {}));
```



实践 – 类型声明 (2)

3. 预定义常量

- 如BasicTypeInfo中定义了一系列常用原生类型的类型信息实例，也可以使用更简单的Types类；
- Types.STRING、Types.INT等等

4. 自定义TypeInfo和TypeInfoFactory

```
@TypeInfo(MyTupleTypeInfoFactory.class)
public class MyTuple<T0, T1> {
    public T0 myfield0;
    public T1 myfield1;
}
```

```
public class MyTupleTypeInfoFactory extends TypeInfoFactory<MyTuple> {

    @Override
    public TypeInformation<MyTuple> createTypeInfo(Type t, Map<String, TypeInformation<?>> genericParameters) {
        return new MyTupleTypeInfo(genericParameters.get("T0"), genericParameters.get("T1"));
    }
}
```

实践 – 注册子类型

```
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
Env. registerType(typeClass);
```

在registerType方法内部，会使用TypeExtractor来提取类型信息，如图：

```
public void registerType(Class<?> type) {
    if (type == null) {
        throw new NullPointerException("Cannot register null type class.");
    }

    TypeInformation<?> typeInfo = TypeExtractor.createTypeInfo(type);

    if (typeInfo instanceof PojoTypeInfo) {
        config.registerPojoType(type);
    } else {
        config.registerKryoType(type);
    }
}
```

获取到的类型信息属于PojoTypeInfo及其子类，那么将其注册到一起；否则统一交给Kryo去处理，Flink并不过问(这种情况下性能会变差)。

实践 – Kryo序列化

对于Flink无法序列化的类型， 默认会交给Kryo处理， 如果Kryo仍然无法处理， 有两种解决方案：

1. 强制使用Avro来代替Kryo
 - `env.getConfig().enableForceAvro();`
2. 为Kryo增加自定义的Serializer以增强Kryo的功能
 - `env.getConfig().addDefaultKryoSerializer(clazz, serializer);`

注：禁用Kryo – `env.getConfig().disableGenericType();`

03

Flink通信层的序列化

Flink通信层的序列化

Flink的Task之间如果需要跨网络传输数据记录，那么就需要将数据序列化之后写入 NetworkBufferPool，然后下层的Task读出之后再进行反序列化操作，最后进行逻辑处理。

为了使得记录以及事件能够被写入Buffer随后在消费时再从Buffer中读出，Flink提供了数据记录序列化器（RecordSerializer）与反序列化器（RecordDeserializer）以及事件序列化器（EventSerializer）。

我们的Function发送的数据被封装成SerializationDelegate，它将任意元素公开为 IReadableWritable以进行序列化，通过setInstance()来传入要序列化的数据。



Flink通信层的序列化

那么问题来了~

- ① 何时确定Function的输入输出类型?
- ② 何时确定Function的序列化/反序列化器?
- ③ 何时进行真正的序列化/反序列化操作?
- ④ 这些与我们上面提到的TypeSerializer又是怎么联系在一起的呢?

带着这些问题，我们继续往下看...

Flink通信层的序列化

Flink作业的提交过程：

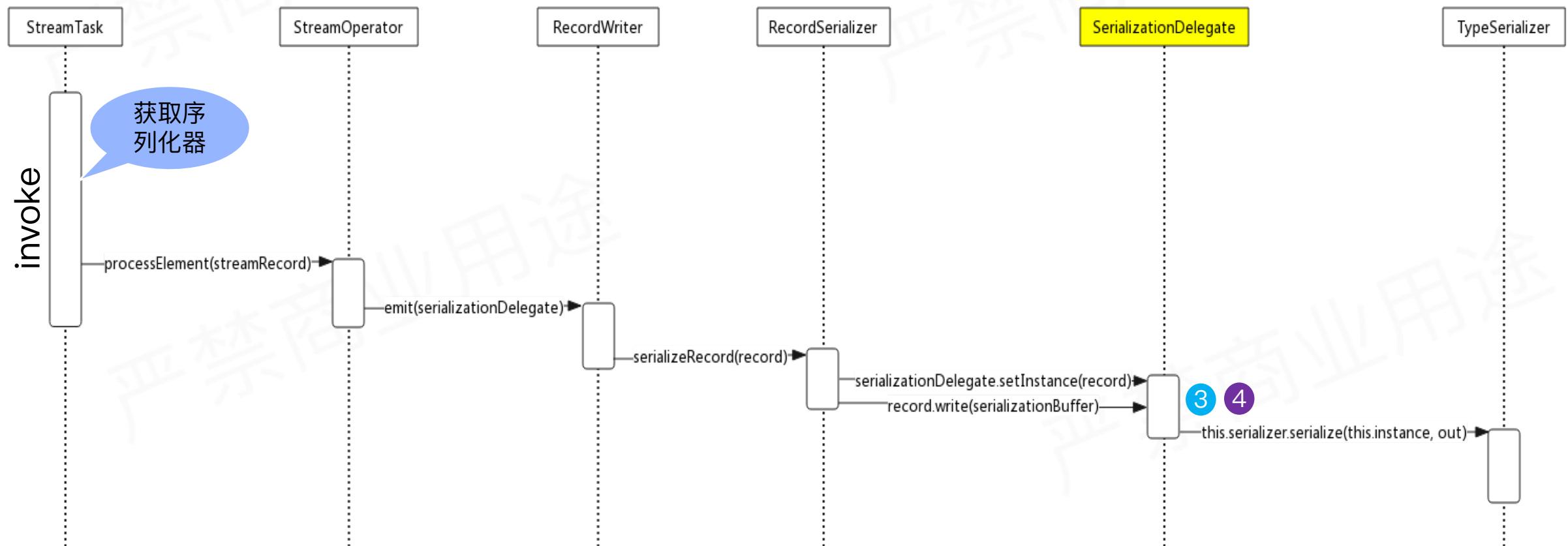


- 1 在构建StreamTransformation的时候通过TypeExtractor工具确定Function的输入输出类型。
TypeExtractor 类可以根据方法签名、子类信息等蛛丝马迹自动提取或恢复类型信息。

- 2 构造StreamGraph的时候，通过TypeInformation的createSerializer()方法获取对应类型的序列化器
TypeSerializer，并在addOperator()的过程中执行setSerializers()操作，设置StreamConfig的
TYPE_SERIALIZER_IN_1、TYPE_SERIALIZER_IN_2、TYPE_SERIALIZER_OUT_1属性。

Flink通信层的序列化

Flink task的调用过程：



04

Q/A



Apache Flink

THANKS

