前言

flink作为一个分布式的数据处理框架，数据的交换传输是其中比较重要的一块。就这涉及到一个重要的概念。数据的序列化。当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。Java默认的序列化保存了很多元数据信息以及类的继承结构信息,很多二进制序列化方式不需要保存什么元数据的，所以它是不高效的。Flink自己控制了网络传输的内存的使用方式，自然在内存使用上使用的更少更紧凑数据传输的效率才能更高效。Flink拥有自己特制的数据序列化框架，没有采用有Java默认的序列化方式。那么flink是如何自己序列化数据的呢？

抛砖引玉

我们在使用Flink做为数据处理框架的时候，程序中可能存在多种数据格式，例如，各种Java的基本类型，string、int等。java对象等。下面的代码演示下一个基本的流处理代码从kafka读取string类型的数据转换为integer后发送到sink端并打印出来。

```
FlinkKafkaConsumer08 kafkaConsumer08 = new FlinkKafkaConsumer08<>
(topic,SimpleStringSchema(), props);
kafkaConsumer08.setStartFromEarliest();

DataStream<String> input =
env.addSource(kafkaConsumer08).rebalance();
input.map((MapFunction<String, Integer>) value ->
Integer.valueOf(value)).addSink(new SinkFunction<Integer>() {
    @Override
    public void invoke(Integer value, Context context) throws
Exception {
        System.out.println(value);
    }
});
env.execute("test");
```

在flink内部自然就需要对从kafka读取出的string类型的数据序列化，string类型的数据转换为integer的数据也需要序列化。那么flink是怎么知道我job的数据类型是什么呢？它知道了我的数据类型又做了什么呢？ 带着上面的问题，我们开始进入flink内部序列化的世界吧！

类型抽取器TypeExtractor

我们都知道flink作业在执行方法`execute`内部事先要转化为`StreamGraph`的。`为StreamGraph`在经过一系列逻辑转为`StreamTask`去真正执行。在这之前就需要把用户定义的function的输入输出的类型确定下来，这一操作发生在构造`为StreamGraph`的时候。flink使用了一个工具类叫`TypeExtractor`.它可以利用方法签名、子类信息等蛛丝马迹，自动提取和恢复类型信息。下面的代码演示是根据用户自定义的MapFunction获取其输入和输出的类型。这个方法内部调用的又是`getUnaryOperatorReturnType`这个吧方法用于返回一元运算符的返回类型，简述逻辑就是先判断是否使用了`Lambda`表达式。如果是就从`Lambda`提取,如果不是就使用反射的方式获取我们自定义的`function`的返回值的类型。

```
//TypeExtractor line 166
@PublicEvolving
public static <IN, OUT> TypeInformation<OUT>
getMapReturnTypes(MapFunction<IN, OUT> mapInterface,
TypeInformation<IN> inType,
        String functionName, boolean allowMissing)
{
    return getUnaryOperatorReturnType(
        (Function) mapInterface,
        MapFunction.class,
        0,
        1,
        new int[]{0},
        NO_INDEX,
        inType,
        functionName,
        allowMissing);
}
```

`getUnaryOperatorReturnType`这个方法使用反射的方式返回一元运算符的返回类型。我们看下它的参数：

```
 * @param function 从中提取返回类型的函数,一般是一个UDF，它实现或者继承
baseClass
 * @param baseClass 函数的基类，一般是个接口
 * @param inputTypeArgumentIndex 基类中输入泛型类型的索引值(位置)（如果
inType为null，则忽略）
 * @param outputTypeArgumentIndex 基类规范中的输出泛型类型的索引
 * @param lambdaOutputTypeArgumentIndices 指定输入类型的类型参数的索引表
 * @param inType 输入元素的类型（在可迭代的情况下，它是元素类型）或null
 * @param functionName 函数的名称
 * @param allowMissing 可以丢失类型信息（这会生成MissingTypeInfo以推迟异
常）
 * @param <IN> Input type 输入类型
 * @param <OUT> Output type 输出类型
 * @return TypeInformation of the return type of the function
 */
@SuppressWarnings("unchecked")
@PublicEvolving
public static <IN, OUT> TypeInformation<OUT>
getUnaryOperatorReturnType(
```

从上面的参数可以看出提取一个UDF的参数化的类型，需要UDF、基类，参数的位置等。如果入参类型不为空的情况下，需要验证入参类型是否跟UFD定义的类型是否匹配，也会判断UDF是否参数化了基类

```
if (inType != null) {
    validateInputType(baseClass, function.getClass(),
inputTypeArgumentIndex, inType);
}
```

验证输入类型的合法性。第一步从baseClass获取其定义的inType的类型。

```
private static void validateInputType(Class<?> baseClass, Class<?>
clazz, int inputParamPos, TypeInformation<?> inTypeInfo) {
    ArrayList<Type> typeHierarchy = new ArrayList<Type>();

    // try to get generic parameter
    Type inType;
    try {
        // 。第一步从baseClass获取其定义的inType的类型。
        inType = getParameterType(baseClass, typeHierarchy, clazz,
inputParamPos);
    }
    catch (InvalidTypesException e) {
        return; // skip input validation e.g. for raw types
    }

    try {
       //第二步验证从baseClass获取其定义的inType的类型与传入的inType是否匹
配
        validateInfo(typeHierarchy, inType, inTypeInfo);
    }
    catch(InvalidTypesException e) {
        throw new InvalidTypesException("Input mismatch: " +
e.getMessage(), e);
    }
}
```

原来比较原来比较目标类跟基类的关系，然后获取参数的类型；第一步：从目标类的实现的接口入手比较跟基类的关系，，然后获取参数的类型。；第二步：如果从目标类获取不到参数类型，那么用目标类的超类跟基类比较。确认参数的类型。如果也没找到抛出异常。

```
private static Type getParameterType(Class<?> baseClass,
ArrayList<Type> typeHierarchy, Class<?> clazz, int pos) {
    if (typeHierarchy != null) {
        typeHierarchy.add(clazz);
    }
    // 获取UDF类实现的接口
    Type[] interfaceTypes = clazz.getGenericInterfaces();

    // search in interfaces for base class
    for (Type t : interfaceTypes) {
        // 查看每个接口的类型是否是参数化类型的实例以及是目标接口是否跟基本接口
相匹配。
        Type parameter = getParameterTypeFromGenericType(baseClass,
typeHierarchy, t, pos);
        if (parameter != null) {
            return parameter;
        }
    }

    // search in superclass for base class
    // 获取UDF类直接父类的类型
    Type t = clazz.getGenericSuperclass();
    Type parameter = getParameterTypeFromGenericType(baseClass,
typeHierarchy, t, pos);
    if (parameter != null) {
        return parameter;
    }

    throw new InvalidTypesException("The types of the interface " +
baseClass.getName() + " could not be inferred. " +
                    "Support for synthetic interfaces, lambdas, and
generic or raw types is limited at this point");
}
```

上面的代码涉及的反射的知识点：

知识点	名词解释
getInterfaces	返回直接实现的接口（由于编译擦除，没有显示泛型参数）
getGenericInterfaces	返回表示某些接口的 Type，这些接口由此对象所表示的类或接口直接实现。（包含泛型参数）
getSuperclass	返回直接继承的父类（由于编译擦除，没有显示泛型参数）
getGenericSuperclass	返回直接继承的父类（包含泛型参数）

getParameterTypeFromGenericType 从给定的Type中获取参数的类型，如果条件都不满足会采用递归的方式，一直获取目标类实现的接口是否满足下面的三个条件，某则返回null

```
private static Type getParameterTypeFromGenericType(Class<?>
baseClass, ArrayList<Type> typeHierarchy, Type t, int pos) {
// 查看每个接口的类型是否是参数化类型的实例以及是目标接口是否跟基本接口相匹配。
    if (t instanceof ParameterizedType &&
baseClass.equals(((ParameterizedType) t).getRawType())) {
        if (typeHierarchy != null) {
            typeHierarchy.add(t);
        }
        ParameterizedType baseClassChild = (ParameterizedType) t;
        //
        return baseClassChild.getActualTypeArguments()[pos];
    }
// 查看每个接口的类型是否是参数化类型的实例以及基本接口是目标接口的超级父类。
    else if (t instanceof ParameterizedType &&
baseClass.isAssignableFrom((Class<?>) ((ParameterizedType)
t).getRawType())) {
        if (typeHierarchy != null) {
            typeHierarchy.add(t);
        }
        return getParameterType(baseClass, typeHierarchy, (Class<?
>) ((ParameterizedType) t).getRawType(), pos);
    }
    //基本接口是目标接口的超级父类。
    else if (t instanceof Class<?> &&
baseClass.isAssignableFrom((Class<?>) t)) {
        if (typeHierarchy != null) {
            typeHierarchy.add(t);
        }
        return getParameterType(baseClass, typeHierarchy, (Class<?
>) t, pos);
    }
    return null;
}
```

上面的代码涉及的反射的知识点：

知识点	名词解释
Parameterized	是一个接口，这个类可以用来检验泛型是否被参数化

Type	
getActualTypeArguments	ParameterizedType的方法，用来获取泛型的类型参数数组
getRawType	ParameterizedType的方法，用来获取声明此参数化类型的类的类型(Type)
A instance of B	用来在运行时指出A对象是否是B类的一个实例
A isAssignableFrom B	判定此 A 对象所表示的类或接口与指定的 B 参数所表示的类或接口是否相同，或者是否是B类的超类或超接口。
TypeVariable	类型变量，描述类型，表示泛指任意或相关一类类型，也可以说狭义上的泛型（泛指某一类类型），一般用大写字母作为变量，比如K、V、E等。所谓范型变量就是<E extends List>或者<E>, 也就是TypeVariable<D>这个接口的对应的对象.Java中的Type详解

如果获取到的输入类型不为空，则使用方法`validateInfo`从baseClass获取其定义的inType的类型与传入的inType是否匹配。这个方法的本质是验证TypeInformation只有的Type是否跟目标类型一致。不一致就报异常。下面的代码演示基本类型的校验逻辑。

```
private static void validateInfo(ArrayList<Type> typeHierarchy,
Type type, TypeInformation<?> typeInfo) {
     //.....
    // check for Java Basic Types
    if (typeInfo instanceof BasicTypeInfo) {

        TypeInformation<?> actual;
        // check if basic type at all
        if (!(type instanceof Class<?>) || (actual =
BasicTypeInfo.getInfoFor((Class<?>) type)) == null) {
            throw new InvalidTypesException("Basic type
expected.");
        }
        // check if correct basic type
        if (!typeInfo.equals(actual)) {
            throw new InvalidTypesException("Basic type '" +
typeInfo + "' expected but was '" + actual + "'.");
        }
    }
    //.....
    // check for **POJO**
    if (typeInfo instanceof PojoTypeInfo) {
        Class<?> clazz = null;
        if (!(isClassType(type) && ((PojoTypeInfo<?>)
typeInfo).getTypeClass() == (clazz = typeToClass(type)))) {
            throw new InvalidTypesException("POJO type '"
                    + ((PojoTypeInfo<?>)
typeInfo).getTypeClass().getCanonicalName() + "' expected but was
'"
                    + clazz.getCanonicalName() + "'.");
        }
    }
}
```

如果前两步都没报异常，那么就进入到privateCreateTypeInfo来构造输出类型的TypeInformation了。我们进入到这个方法是怎么实现的。首先判断是否是`TypeVariable`类型的子类，如果是直接获取类型。否则使用方法`createTypeInfoWithTypeHierarchy`进行获取。

```
TypeExtractor().privateCreateTypeInfo(baseClass,
function.getClass(), outputTypeArgumentIndex, inType, null);
```

```
// for (Rich)Functions
private <IN1, IN2, OUT> TypeInformation<OUT>
privateCreateTypeInfo(Class<?> baseClass, Class<?> clazz, int
returnParamPos,
        TypeInformation<IN1> in1Type, TypeInformation<IN2> in2Type)
{
    ArrayList<Type> typeHierarchy = new ArrayList<Type>();
    // 获取输出参数的类型，getParameterType这个方法我们在前面讲过了。
    Type returnType = getParameterType(baseClass, typeHierarchy,
clazz, returnParamPos);

    TypeInformation<OUT> typeInfo;

    // return type is a variable -> try to get the type info from
the input directly
    //首先判断是否是TypeVariable类型的子类，如果是直接获取类型
    if (returnType instanceof TypeVariable<?>) {
        typeInfo = (TypeInformation<OUT>)
createTypeInfoFromInputs((TypeVariable<?>) returnType,
typeHierarchy, in1Type, in2Type);

        if (typeInfo != null) {
            return typeInfo;
        }
    }

    // get info from hierarchy
    return (TypeInformation<OUT>)
createTypeInfoWithTypeHierarchy(typeHierarchy, returnType, in1Type,
in2Type);
}
```

方法`createTypeInfoWithTypeHierarchy`的逻辑概述如下：

- 首先根据参数t(Type类型)判断是否存在对应的TypeInfoFromFactory,对应方法`createTypeInfoFromFactory`
- 方法`createTypeInfoFromFactory`通过`getClosestFactory`循环获取参数t(Type类型)的，条件必须是class的子类型，必须是参数化的子类ParameterizedType，不能是Object.class;`t instanceof Class<?> || t instanceof ParameterizedType;`

```
private static <OUT> TypeInfoFactory<? super OUT>
getClosestFactory(ArrayList<Type> typeHierarchy, Type t) {
    TypeInfoFactory factory = null;
    while (factory == null && isClassType(t) && !
(typeToClass(t).equals(Object.class))) {
        typeHierarchy.add(t);
        factory = getTypeInfoFactory(t);
        t = typeToClass(t).getGenericSuperclass();

        if (t == null) {
            break;
        }
    }
    return factory;
}
```

- 方法`getTypeInfoFactory`首先通过检查当前已注册的`registeredTypeInfoFactories`集合中是否包含对象的TypeInfoFactory，有直接返回，没有的话看当前Type参数头上是否包含注解@TypeInfo，如果包含则获取注解的值value，判断当前value必须是`TypeInfoFactory`子类。满足条件后使用`InstantiationUtil.instantiate(factoryClass);`实例化并返回。

```
public static <OUT> TypeInfoFactory<OUT> getTypeInfoFactory(Type t)
{
    final Class<?> factoryClass;
    if (registeredTypeInfoFactories.containsKey(t)) {
        factoryClass = registeredTypeInfoFactories.get(t);
    }
    else {
        if (!isClassType(t) ||
!typeToClass(t).isAnnotationPresent(TypeInfo.class)) {
            return null;
        }
        final TypeInfo typeInfoAnnotation =
typeToClass(t).getAnnotation(TypeInfo.class);
        factoryClass = typeInfoAnnotation.value();
        // check for valid factory class
        if (!TypeInfoFactory.class.isAssignableFrom(factoryClass))
{
            throw new InvalidTypesException("TypeInfo annotation
does not specify a valid TypeInfoFactory.");
        }
    }

    // instantiate
    return (TypeInfoFactory<OUT>)
InstantiationUtil.instantiate(factoryClass);
}
```

- 回到方法 `createTypeInfoFromFactory` ,如果能找到参数t(Type类型)的TypeInfoFactory则调用其方法 `createTypeInfo` 返回其对应的 `TypeInformation` ;
- 回到方法 `createTypeInfoWithTypeHierarchy` ,如果没有能找到参数t(Type类型)的TypeInfoFactory。接着判断其是不是 `Tuple` 类型， `TypeVariable` (泛型多继承类型), `GenericArrayType` (泛型数组类型), `ParameterizedType` (参数化类型)， `Class` 的实例。如果都不是其中之一，就该跑异常了。

这里我们以类型为class的实例为例，看看看如何找到class的typeInfoMation;

```
// no tuple, no TypeVariable, no generic type
else if (t instanceof Class) {
    return privateGetForClass((Class<OUT>) t, typeHierarchy);
}
```

简述逻辑就是判断当前参数的type如果是Object.class或者Class.class就直接返回GenericTypeInfo。如果是基本类型`BasicTypeInfo.getInfoFor(clazz);`就直接返回了。如果以上都不是，开始分析是不是POJO类型。通过方法analyzePojo。

> Type直接实现子类 :Class类

> Type直接子接口 : ParameterizedType，GenericArrayType，TypeVariable和WildcardType四种类型的接口

```
@SuppressWarnings("unchecked")
protected <OUT, IN1, IN2> TypeInformation<OUT>
analyzePojo(Class<OUT> clazz, ArrayList<Type> typeHierarchy,
        ParameterizedType parameterizedType, TypeInformation<IN1>
in1Type, TypeInformation<IN2> in2Type) {

    if (!Modifier.isPublic(clazz.getModifiers())) {
        LOG.info("Class " + clazz.getName() + " is not public,
cannot treat it as a POJO type. Will be handled as GenericType");
        return new GenericTypeInfo<OUT>(clazz);
    }

    // add the hierarchy of the POJO itself if it is generic
    if (parameterizedType != null) {
        getTypeHierarchy(typeHierarchy, parameterizedType,
Object.class);
    }
    // create a type hierarchy, if the incoming only contains the
most bottom one or none.
    else if (typeHierarchy.size() <= 1) {
        getTypeHierarchy(typeHierarchy, clazz, Object.class);
    }

    List<Field> fields = getAllDeclaredFields(clazz, false);
    if (fields.size() == 0) {
        LOG.info("No fields detected for " + clazz + ". Cannot be
used as a PojoType. Will be handled as GenericType");
        return new GenericTypeInfo<OUT>(clazz);
    }

    List<PojoField> pojoFields = new ArrayList<PojoField>();
    for (Field field : fields) {
        Type fieldType = field.getGenericType();
        if(!isValidPojoField(field, clazz, typeHierarchy)) {
            LOG.info(clazz + " is not a valid POJO type because not
```

```
all fields are valid POJO fields.");
            return null;
        }
        try {
            ArrayList<Type> fieldTypeHierarchy = new
ArrayList<Type>(typeHierarchy);
            fieldTypeHierarchy.add(fieldType);
            TypeInformation<?> ti =
createTypeInfoWithTypeHierarchy(fieldTypeHierarchy, fieldType,
in1Type, in2Type);
            pojoFields.add(new PojoField(field, ti));
        } catch (InvalidTypesException e) {
            Class<?> genericClass = Object.class;
            if(isClassType(fieldType)) {
                genericClass = typeToClass(fieldType);
            }
            pojoFields.add(new PojoField(field, new
GenericTypeInfo<OUT>((Class<OUT>) genericClass)));
        }
    }

    CompositeType<OUT> pojoType = new PojoTypeInfo<OUT>(clazz,
pojoFields);

    //
    // Validate the correctness of the pojo.
    // returning "null" will result create a generic type
information.
    //
    List<Method> methods = getAllDeclaredMethods(clazz);
    for (Method method : methods) {
        if (method.getName().equals("readObject") ||
method.getName().equals("writeObject")) {
            LOG.info(clazz+" contains custom serialization methods
we do not call.");
            return null;
        }
    }

    // Try retrieving the default constructor, if it does not have
one
    // we cannot use this because the serializer uses it.
    Constructor defaultConstructor = null;
    try {
        defaultConstructor = clazz.getDeclaredConstructor();
    } catch (NoSuchMethodException e) {
        if (clazz.isInterface() ||
Modifier.isAbstract(clazz.getModifiers())) {
```

```
            LOG.info(clazz + " is abstract or an interface, having
a concrete " +
                    "type can increase performance.");
        } else {
            LOG.info(clazz + " must have a default constructor to
be used as a POJO.");
            return null;
        }
    }
    if(defaultConstructor != null &&
!Modifier.isPublic(defaultConstructor.getModifiers())) {
        LOG.info("The default constructor of " + clazz + " should
be Public to be used as a POJO.");
        return null;
    }

    // everything is checked, we return the pojo
    return pojoType;
}
```

这里在多说一句`StreamGraph`的转化是使用了工具类`StreamGraphGenerator`的方法generate进行转化的。方法的入参是`List<StreamTransformation<?>>`;它属于`StreamExecutionEnvironment`属性。

```
public static StreamGraph generate(StreamExecutionEnvironment env,
List<StreamTransformation<?>> transformations) {
    return new
StreamGraphGenerator(env).generateInternal(transformations);
}
```

`StreamTransformation`描述了构建`DataStream`的信息。上面的获取`MapFunction`的输出类型的操作发生在构建`StreamOperator`时候。注意：这里的`outType`的类型是个`TypeInformation`.这也是一个重要的累。我们后面会详细讲解这个类。

```
//DataStream line 1084
public <R> SingleOutputStreamOperator<R> map(MapFunction<T, R>
mapper) {
    TypeInformation<R> outType =
TypeExtractor.getMapReturnTypes(clean(mapper), getType(),
            Utils.getCallLocationName(), true);
    return transform("Map", outType, new StreamMap<>
(clean(mapper)));
}
```

transform 会调用 `StreamExecutionEnvironment` 的方法 `addOperator` 填充其属性 `List<StreamTransformation<?>>`;

```
public <R> SingleOutputStreamOperator<R> transform (String
operatorName, TypeInformation<R> outTypeInfo,
OneInputStreamOperator<T, R> operator) {
// read the output type of the input Transform to coax out errors
about MissingTypeInfo
transformation.getOutputType();

OneInputTransformation<T, R> resultTransform = new
OneInputTransformation<>(
        this.transformation,
        operatorName,
        operator,
        outTypeInfo,
        environment.getParallelism());

@SuppressWarnings({ "unchecked", "rawtypes" })
SingleOutputStreamOperator<R> returnStream = new
SingleOutputStreamOperator(environment, resultTransform);

getxecutionEnvironment().addOperator(resultTransform);
return returnStream;
}
```

这样 `StreamTransformation` 包含的 `TypeInformation(OutputType)` 就会在构建 `StreamGraph` 的时候被使用到。详见 `StreamGraph` 的方法 `addOperator` .实例代码如下:

```
TypeSerializer<OUT> outSerializer = outTypeInfo != null && !

(outTypeInfo instanceof MissingTypeInfo) ?
outTypeInfo.createSerializer(executionConfig) : null;
setSerializers(vertexID, inSerializer, null, outSerializer);
```

上面的代码是为其包含的streamNode设置其的出入的序列化器。

```
public void setSerializers(Integer vertexID, TypeSerializer<?> in1,
TypeSerializer<?> in2, TypeSerializer<?> out) {
    StreamNode vertex = getStreamNode(vertexID);
    vertex.setSerializerIn1(in1);
    vertex.setSerializerIn2(in2);
    vertex.setSerializerOut(out);
}
```

最终`StreamGraph`在转化为`StreamTask`其本身的各种Config信息都会转为`StreamTask`配置，也包括序列化的配置信息。 在讲解上述flink是如何抽取用户自定义的Function的输入输出的类型的时候，大家肯定注意到了一个类`TypeInformation`,那么它是做什么呢?貌似它在这里起到了一个封装类型信息的作用。下面我们讲注意力转移到这个类上。因为我们想知道既然知道了数据类型，那么flink是如何序列化用户的数据的?

TypeInformation

TypeInformation是所有类型描述类的基类。它包括了一些类型的基本属性，并可以动过它来生成序列化器（serializer）.它是一个抽象类，需要根据类型信息的不同去实现对应的方法。例如：下面的创建序列化器的方法。

```
public abstract TypeSerializer<T> createSerializer(ExecutionConfig
config);
```

TypeInformation 支持以下几种类型：

实现类	作用
`BasicTypeInfo`	任意Java 基本类型（装箱的）或 String 类型。
`BasicArrayTypeInfo`	任意 `Java` 基本类型数组（装箱的）或 `String` 数组。
`WritableTypeInfo`	任意 `Hadoop Writable` 接口的实现类。
`TupleTypeInfo`	任意的 Flink Tuple 类型(支持 `Tuple1 to Tuple25`)。 `Flink tuples` 是固定长度固定类型的Java Tuple实现。
`CaseClassTypeInfo`	任意的 `Scala CaseClass` (包括 `Scala tuples`)。
`PojoTypeInfo`	任意的 `POJO (Java or Scala)` ，例如， `Java` 对象的所有成员变量，要么是 `public` 修饰符定义，要么有 `getter/setter` 方法。
`GenericTypeInfo`	任意无法匹配之前几种类型的类。

上述的类型都是 `TypeInformation` 的实现类，基本上可以满足绝大部分的 `Flink` 程序，Flink皆可以自动生成对应的 `TypeSerializer` ，能非常高效地对数据集进行序列化和反序列化。

注意： `Flink` 自身不会序列化泛型，而是借助Kryo进行序列化. 另外 `POJO` 类也主要着重注意下其格式，否则它不会被flink自己序列化，而是使用 `kryo` 进行序列化。那就有点惨了。因为 `POJO` 类可以支持复杂类型的创建。 `POJO` 类型的规则 在满足如下条件时，Flink会将这种数据类型识别成POJO类型,并使用 `PojoTypeInfo` 定义的序列化器进行正反序列化。

- 该类是`public`的并且是独立的（即没有非静态的内部类）
- 该类有一个`public`的无参构造方法
- 该类（及该类的父类）的所有成员要么是public的，要么是拥有按照标准java bean命名规则命名的`public getter`和`public setter`方法。

TypeInformation的使用

一般我们在程序中不会主动使用这个类，因为`flink`在内部已经帮我们做了。唯一使用的地方就是在自定义`state`的时候。需要显示的指定`state`里的数据类型信息，以便flink进行序列化后进行存储。下面展示下`FlinkKafkaConsumerBase`中使用stat在存储topic消费信息的代码：

```
// FlinkKafkaConsumerBase line 696
this.unionOffsetStates = stateStore.getUnionListState(new
ListStateDescriptor<>(
        OFFSETS_STATE_NAME,
        TypeInformation.of(new TypeHint<Tuple2<KafkaTopicPartition,
Long>>() {})));
```

// 1.先检查是不是Lambda表达式类型 2.不是先验证数据类型

泛型类型的序列化

注意上面的代码使用了`TypeHint`包装了`Tuple2`.因为`Tuple2`是泛型类型。Java通常会擦除类型信息。所以这里需要使用`TypeHint`以下。在内部，这个创建了TypeHint的匿名子类，来捕获类型的泛型信息并保持知道运行时。对于非泛型数据类型，你可以传递Class：

```
TypeInformation<String> info = TypeInformation.of(String.class);
```

POJO类型的序列化

`PojoTypeInfomation`类用于创建针对`POJO`对象中所有成员的序列化器。`Flink`自带了针对诸如`int，long，String`等标准类型的序列化器，对于所有其他的类型，我们交给Kryo处理。

如果Kryo不能处理某种类型，我们可以通过`PojoTypeInfo`去调用`Avro`来序列化`POJO`，为了这样处理，我们必须调用如下接口：

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().enableForceAvro();
```

> 注意：Flink会用Avro序列化器自动序列化由Avro产生POJO对象。

如果你想使得整个`POJO`类型都交给Kryo序列化器处理，则你需要配置：

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().enableForceKryo();
```

如果Kryo不能序列化该POJO对象，你可以添加一个自定义序列化器到Kryo，使用代码如下：

```
env.getConfig().addDefaultKryoSerializer(Class<?> type, Class<?
extends Serializer<?>> serializerClass)
```

自定义 TypeInfo 和 TypeInfoFactory

类型信息工厂允许以插件的形式自定义类型信息到 Flink 类型系统中，你需要实现`org.apache.flink.api.common.typeinfo.TypeInfoFactory`用于返回你自定义类型信息。 当对应的类型已经注解了`@org.apache.flink.api.common.typeinfo.TypeInfo`，那么在类型提取阶段 Flink 就会调用你实现的工厂。

> 类型信息工厂在Java和Scala API中均可使用。

通过自定义`TypeInfo`为任意类提供`Flink`原生内存管理（而非`Kryo`），可令存储更紧凑，运行时也更高效。开发者在自定义类上使用`@TypeInfo`注解，随后创建相应的`TypeInfoFactory`并覆盖`createTypeInfo`方法。 下面的例子展示了在Java中如何注解一个自定义类型，并且通过工厂创建自定义的类型信息对象。

自定义类型`MyTuple`：

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

利用工厂创建自定义类型信息对象:

```
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"));
  }
}
```

`createTypeInfo(Type, Map<String, TypeInformation<?>>)` 方法用于创建该工厂针对类型的类型信息，参数和类型的泛型参数可以提供关于该类型的附加信息(如果有参数)。方法的返回值是我们自定义的 `TypeInformation` 。

注意自定义 `TypeInformation` 这里是 `MyTupleTypeInfo` 需要继承 `TypeInformation` 类，为每个字段定义类型，并覆盖元数据方法，例如是否是基本类型（isBasicType）、是否是 `Tuple` （isTupleType）、元数（对于一维的 `Row` 类型，等于字段的个数）等等，从而为 `TypeExtractor` 提供决策依据。

```
public static class MyTupleTypeInfo<T0, T1> extends
TypeInformation<MyTuple<T0, T1>> {
        private TypeInformation field0;
        private TypeInformation field1;
        // .....
        @Override
        public Map<String, TypeInformation<?>>
getGenericParameters() {
            Map<String, TypeInformation<?>> map = new HashMap<>(2);
            map.put("T0", field0);
            map.put("T1", field1);
            return map;
        }
    }
```

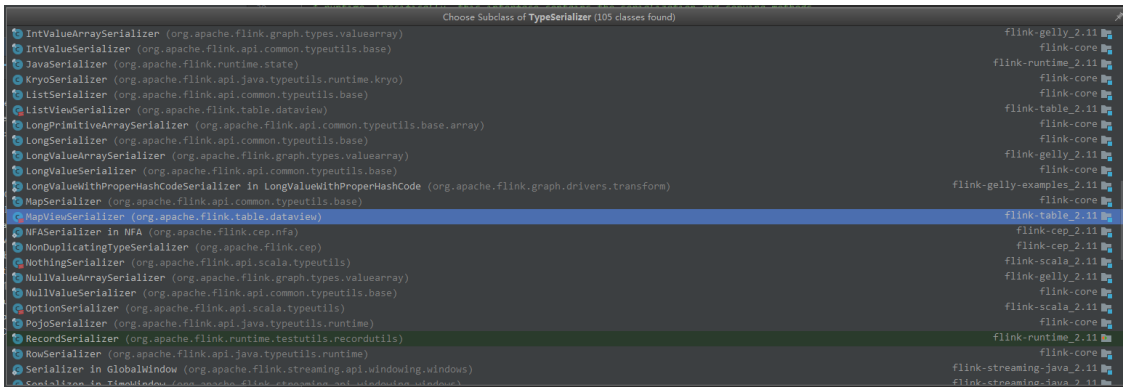
如果你的类型包含的泛型参数可能源自Flink方法的输入类型，请确保实现了`org.apache.flink.api.common.typeinfo.TypeInformation#getGenericParameters`这个方法用于建立从泛型参数到类型信息的双向映射。 更多示例，请参考 Flink 源码的 org/apache/flink/api/java/typeutils/TypeInfoFactoryTest.java

TypeSerializer

TypeSerializer是个接口，此接口描述Flink运行时处理数据类型所需的方法。 具体来说，此接口包含序列化和复制方法。此类中的方法假定为无状态，因此它是有效的线程安全的。 这些方法的有状态实现可能导致不可预测的副作用，并且会损害程序的稳定性和正确性。之类需要着重实现下面的方法。

```
public abstract T copy(T from, T reuse);
public abstract void serialize(T record, DataOutputView target)
throws IOException;
public abstract T deserialize(DataInputView source) throws
IOException;
```

Flink 自带了很多 TypeSerializer子类，大多数情况下各种自定义类型都是常用类型的排列组合，因而可以直接复用：



如果不能满足，那么可以继承 TypeSerializer 及其子类以实现自己的序列化器。 下面以`PojoTypeInfo`为示例，讲解下它是如何序列化一个`POJO`的。

```
public PojoTypeInfo(Class<T> typeClass, List<PojoField> fields) {
}
```

`PojoTypeInfo`的构造函数需要`Class`类型的参数，`List<PojoField>`对应的字段集合。`PojoTypeInfo`的初始化是通过类型抽取器TypeExtractor初始化的。它集成`TypeInformation`。它在实现`createSerializer`时候根据`ExecutionConfig`中是否配置了强制使用`Kryo`或者`Avro`。如果都没有就使用自己实现的序列化方式。构造一个`PojoSerializer`。

```
public TypeSerializer<T> createSerializer(ExecutionConfig config) {
    if (config.isForceKryoEnabled()) {
        return new KryoSerializer<>(getTypeClass(), config);
    }

    if (config.isForceAvroEnabled()) {
        return
AvroUtils.getAvroUtils().createAvroSerializer(getTypeClass());
    }
    return createPojoSerializer(config);
}
```

`PojoSerializer`是`POJO`对象的序列化器。它需要记录当前`POJO`的class类型以及他们的字段的序列化器。有了这些信息就可以进行序列化跟反序列化了。

```
public PojoSerializer<T> createPojoSerializer(ExecutionConfig
config) {
    TypeSerializer<?>[] fieldSerializers = new TypeSerializer<?>
[fields.length];
    Field[] reflectiveFields = new Field[fields.length];

    for (int i = 0; i < fields.length; i++) {
        fieldSerializers[i] =
fields[i].getTypeInformation().createSerializer(config);
        reflectiveFields[i] = fields[i].getField();
    }

    return new PojoSerializer<T>(getTypeClass(), fieldSerializers,
reflectiveFields, config);
}
```

我们下面分别看下它的`serialize`和`deserialize`方法。探究flink是如何高效的操作的。 下面的序列化的代码逻辑简述如下，首先判断写入的对象是否为null，为null的化就写入一个字节的标志`IS_NULL`.不为null，首先判读当前写入的value的class跟当前序列化持有的class是否匹配。不匹配，看下有没有注册这个子类型。如果注册了就使用这个子类序列化器。然后将标识是子类的表示`IS_SUBCLASS`和子类的class的全限定名写入。否则只写入一个字节的标识`NO_SUBCLASS`；为啥只写入一个标识，因为当前学历化器已经记住了当前要序列化的数据的class。之后调用每个字段的序列化器顺序写入即可。

```
public void serialize(T value, DataOutputView target) throws
IOException {
        int flags = 0;
        // handle null values
        if (value == null) {
            flags |= IS_NULL;
            target.writeByte(flags);
            return;
        }

        Integer subclassTag = -1;
        Class<?> actualClass = value.getClass();
        TypeSerializer subclassSerializer = null;
        if (clazz != actualClass) {
            subclassTag = registeredClasses.get(actualClass);
            if (subclassTag != null) {
                flags |= IS_TAGGED_SUBCLASS;
                subclassSerializer =
registeredSerializers[subclassTag];
            } else {
                flags |= IS_SUBCLASS;
                subclassSerializer =
getSubclassSerializer(actualClass);
            }
        } else {
            flags |= NO_SUBCLASS; //举例说明：a|=b的意思就是把a和b按位或
然后赋值给a  按位或的意思就是先把a和b都换成2进制，然后用或操作，相当于a=a|b
a!=b的意思a不等于b；
        }

        target.writeByte(flags);

        // if its a registered subclass, write the class tag id,
otherwise write the full classname
        if ((flags & IS_SUBCLASS) != 0) {
            target.writeUTF(actualClass.getName());
        } else if ((flags & IS_TAGGED_SUBCLASS) != 0) {
```

```
            target.writeByte(subclassTag);
        }

        // if its a subclass, use the corresponding subclass
serializer,
        // otherwise serialize each field with our field
serializers
        if ((flags & NO_SUBCLASS) != 0) {
            try {
                for (int i = 0; i < numFields; i++) {
                    Object o = (fields[i] != null) ?
fields[i].get(value) : null;
                    if (o == null) {
                        target.writeBoolean(true); // null field
handling
                    } else {
                        target.writeBoolean(false);
                        fieldSerializers[i].serialize(o, target);
                    }
                }
            } catch (IllegalAccessException e) {
                throw new RuntimeException("Error during POJO copy,
this should not happen since we check the fields before.", e);
            }
        } else {
            // subclass
            if (subclassSerializer != null) {
                subclassSerializer.serialize(value, target);
            }
        }
    }
```

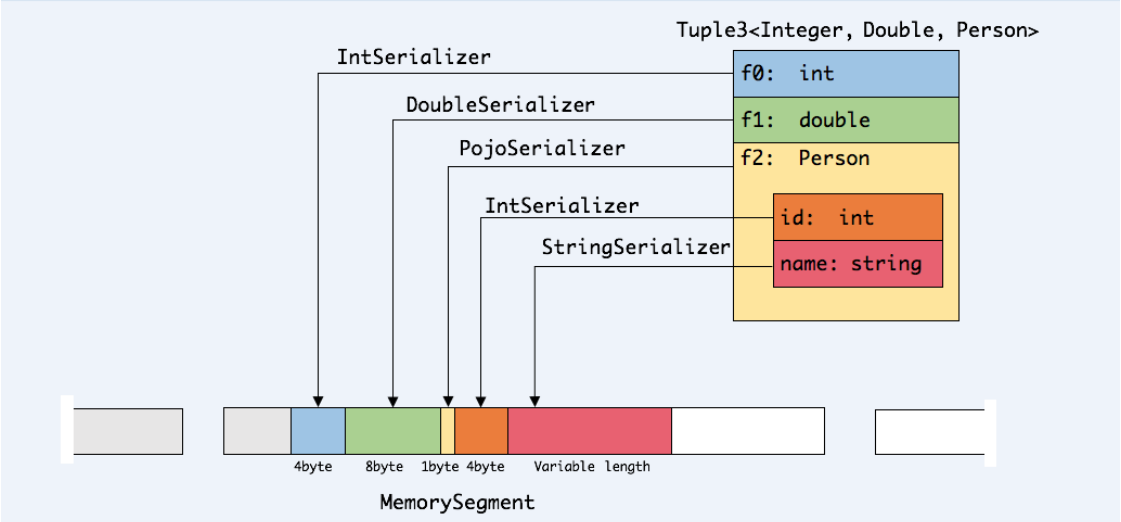
从上面的代码可以看出在在编码阶段声明父类类型，真正数据传输的使用是其子类型。学历化的开销比直接声明使用子类型开销大多了。体现在上面代码的`subclassSerializer.serialize(value, target);`反序列化就不赘叙了，跟是上面的过程相反。唯一注意还是子类型的问题。

```
public T deserialize(DataInputView source) throws IOException {
        int flags = source.readByte();
        if((flags & IS_NULL) != 0) {
            return null;
        }

        T target;
```

```
        Class<?> actualSubclass = null;
        TypeSerializer subclassSerializer = null;

        if ((flags & IS_SUBCLASS) != 0) {
            String subclassName = source.readUTF();
            try {
                actualSubclass = Class.forName(subclassName, true,
cl);
            } catch (ClassNotFoundException e) {
                throw new RuntimeException("Cannot instantiate
class.", e);
            }
            subclassSerializer =
getSubclassSerializer(actualSubclass);
            target = (T) subclassSerializer.createInstance();
            // also initialize fields for which the subclass
serializer is not responsible
            initializeFields(target);
        } else if ((flags & IS_TAGGED_SUBCLASS) != 0) {

            int subclassTag = source.readByte();
            subclassSerializer =
registeredSerializers[subclassTag];
            target = (T) subclassSerializer.createInstance();
            // also initialize fields for which the subclass
serializer is not responsible
            initializeFields(target);
        } else {
            target = createInstance();
        }

        if ((flags & NO_SUBCLASS) != 0) {
            try {
                for (int i = 0; i < numFields; i++) {
                    boolean isNull = source.readBoolean();

                    if (fields[i] != null) {
                        if (isNull) {
                            fields[i].set(target, null);
                        } else {
                            Object field =
fieldSerializers[i].deserialize(source);
                            fields[i].set(target, field);
                        }
                    } else if (!isNull) {
                        // read and dump a pre-existing field value
                        fieldSerializers[i].deserialize(source);
```

```
                    }
                }
            } catch (IllegalAccessException e) {
                throw new RuntimeException("Error during POJO copy,
this should not happen since we check the fields before.", e);
            }
        } else {
            if (subclassSerializer != null) {
                target = (T) subclassSerializer.deserialize(target,
source);
            }
        }
        return target;
    }
```

可以看到，Flink 对于内存管理是非常细致的，层次分明，代码也容易理解。对象序列化后会写到 DataOutput (由 `MemorySegement` 所支持)，并高效地自动通过 Java 的 Unsafe API 操作写到内存。对于基本类型是这样的方式写入，那么对于符合类型是如何高效的序列化的呢？复合类型的对象可能包含内嵌的数据类型，其TypeSerializer和TypeComparator也是组合的，序列化和比较时会委托给对应的serializers和comparators。下图描述了一个典型的复合类型对象 `Tuple<Integer, Double, Person>` 是如何序列化和存储的:



可以看出这种序列化方式存储密度是相当紧凑的。其中 `int` 占4字节， `double` 占8字节， `POJO` 多个一个字节的 `header` ， `Pojo` 对应的 `PojoTypeInfo` 内部持有的 `Serializer` 只负责将 `header` 序列化进去，并委托每个字段对应的 `serializer` 对字段进行序列化。 更多关于Flink是如何高效的玩弄内存的请参考这个博文Juggling with Bits and Bytes

常见问题

用户如果需要介入 Flink 类型框架的话，一般是在遇到下面这些情况的时候：

1. 注册子类型： 如果方法签名只描述了父类型，但实际执行中用到了该类型的子类型，让Flink知道这些子类型能够提升不少性能。因此，`在StreamExecutionEnvironment`或`ExecutionEnvironment`中应当为每个子类型调用`.registerType(clazz)`方法。
2. 注册自定义序列化器： Flink会将自己不能处理的类型转交给Kryo,但并不是所有的类型都能被Kryo完美处理（也就是说：不是所有类型都能被Flink处理），比如，许多 Google Guava 的集合类型默认情况下是不能正常工作的。针对存在这个问题的这些类型，其解决方案 是通过在`StreamExecutionEnvironment`或`ExecutionEnvironment`中调用`.getConfig().addDefaultKryoSerializer(clazz, serializer)`方法，注册辅助的序列化器。许多库都提供了`Kryo`序列化器,关于自定义序列化器的更多细节请参考Custom Serializers。
3. 添加Type Hints： 有时，Flink尝试了各种办法仍不能推断出泛型，这时用户就必须通过借助type hint来推断泛型。
4. 手动创建一个 TypeInformation类： 在某些API中，手动创建一个TypeInformation类可能是必须的，因为Java泛型的类型擦除特性会使得Flink无法推断数据类型。更多详细信息可以参考Creating a TypeInformation or TypeSerializer

通信层序列化器

Flink的Task之间如果需要跨网络传输数据记录，那么就需要讲数据序列化之后写入`NetworkBufferPool`下层的Task读出之后在进行反序列化进行逻辑处理。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发送的数据是被封装成一个类叫`SerializationDelegate`.它将任意元素公开为`IOReadableWritable`以进行序列化。通过setInstance的方式传入要序列化的数据来设置其值。

```
private T instance;

private final TypeSerializer<T> serializer;

public SerializationDelegate(TypeSerializer<T> serializer) {
    this.serializer = serializer;
}

public void setInstance(T instance) {
    this.instance = instance;
}
```

发送数据的接口叫Collector，例如它的一个实现类叫OutputCollector,它的构造函数中的参数`serializer`，是数据出口序列化器它从Task的Config获取的。

```
public OutputCollector(List<RecordWriter<SerializationDelegate<T>>>
writers, TypeSerializer<T> serializer) {
        this.delegate = new SerializationDelegate<T>(serializer);
        this.writers = (RecordWriter<SerializationDelegate<T>>[])
writers.toArray(new RecordWriter[writers.size()]);
    }
```

```
// get the factory for the serializer
final TypeSerializerFactory<T> serializerFactory =
config.getOutputSerializer(cl);
//初始化OutputCollector
new OutputCollector<T>(writers, serializerFactory.getSerializer());
```

有了序列化器响应的Task的`RecordSerializer`以及`RecordDeserializer`就能正确序列化以及反序列化数据，并发送到入`NetworkBufferPool`。

参考文档

- http://doc.flink-china.org/1.2.0/dev/types_serialization.html
- http://doc.flink-china.org/1.2.0/dev/custom_serializers.html
- https://cloud.tencent.com/developer/article/1240444
- https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html
- http://wuchong.me/blog/2016/04/29/flink-internals-memory-manage/

Flink与其他序列化框架比较

这里主要想颜值下flink的序列化器的高效性。验证的点是，Java对象序列化后的大小。

准备数据。采用的Java对象TestSerial：

```
public class TestSerial implements Serializable {
    public byte version = 100;
    public byte count = 0;
}
```

分别使用不同的序列化方式。序列化TestSerial的实例，并生成序列化后的byte文件。

- 使用flink自定义的POJO的序列化方式

```
public static void useFlinkSerializer() throws Exception {
    TestSerial demo = new TestSerial();
    TypeInformation<TestSerial> type =
TypeExtractor.getForClass(TestSerial.class);
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputViewStreamWrapper target = new
DataOutputViewStreamWrapper(out);
    TypeSerializer<TestSerial> serializer =
type.createSerializer(new ExecutionConfig());
    serializer.serialize(demo, target);
    try (OutputStream outputStream = new
FileOutputStream("flink.out")) {
        out.writeTo(outputStream);
    }
}
```

- 使用Kyro的POJO的序列化方式

```
public static void useKryoSerializer() throws Exception {
    TestSerial demo = new TestSerial();
    KryoSerializer<TestSerial> serializer = new KryoSerializer<>
(TestSerial.class, (new ExecutionConfig()));
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputViewStreamWrapper target = new
DataOutputViewStreamWrapper(out);
    serializer.serialize(demo, target);
    try (OutputStream outputStream = new
FileOutputStream("kryo.out")) {
        out.writeTo(outputStream);
    }
}
```

- 使用Avro的序列化方式

```
private static void useAvroSerializer() throws Exception {
    TestSerial demo = new TestSerial();
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputViewStreamWrapper target = new
DataOutputViewStreamWrapper(out);
    TypeSerializer<TestSerial> serializer =
AvroUtils.getAvroUtils().createAvroSerializer(TestSerial.class);
    serializer.serialize(demo, target);
    try (OutputStream outputStream = new
FileOutputStream("avro.out")) {
        out.writeTo(outputStream);
    }
}
```

- 使用Java的POJO的序列化方式

```
public static void useJavaSerializer() throws Exception {
    TestSerial demo = new TestSerial();
    FileOutputStream fos = new FileOutputStream("java.out");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(demo);
    oos.flush();
    oos.close();
}
```

- 结果

文件名	大小
flink.out	5 字节
kryo.out	4 字节
avro.out	8 字节
java.out	63 字节

- 分析

我们注意到`TestSerial`类中只有两个属性, 且都是 byte 型，理论上存储这两个域只需要2个`byte`，但是实际上`java.out`占据空间为`63bytes`，也就是说除了数据以外，还包括了对序列化对象的其他描述。

> Java 的序列化算法 序列化算法一般会按步骤做如下事情：

- 将对象实例相关的类元数据输出。
- 递归地输出类的超类描述直到不再有超类。
- 类元数据完了以后，开始从最顶层的超类开始 输出对象实例的实际数据值。
- 从上至下递归输出实例的数据

所以说采用java默认的序列化方式是非常耗费存储空间的。

flink的序列化后的文件大小比kryo多出两个字节是怎么回事呢？这是因为PojoSerializer这个类的`serialize`方法在针对对象的属性序列化的时候会使用一个字节记录当前属性的值是否为null.代码如下：

```
for(int i = 0; i < this.numFields; ++i) {
    Object o = this.fields[i] != null ? this.fields[i].get(value) :
null;
    if (o == null) {
        target.writeBoolean(true); // 这里
    } else {
        target.writeBoolean(false); // 这里
        this.fieldSerializers[i].serialize(o, target);
    }
}
```

由于我们的测试对象`TestSerial`类的两个属性都不为`null`，所以增加了两个字节。我这个测试好像不太全面，这么比较起来貌似kryo的序列化方式好像更好些，感兴趣的同学可以在深入的测试下。

参考文档

- http://longdick.iteye.com/blog/458557
- https://blog.csdn.net/xlgen157387/article/details/79840134
- https://www.cnblogs.com/hntyzgn/p/7122709.html