凡普实时数据处理架构

东邪

凡普金科 FINUP

让金融有温度

凡普金科是一家金融科技集团，下属公司最早创立于2013 年。集团专注于互联网大数据处理和移动金融科技研发，为小微个人提供更高效的智能金融服务和解决方案，用科技缩短人和金融服务的距离，促进金融服务平等化，践行普惠金融。

以“让金融有温度”为愿景，“数据驱动业务、技术改变金融”为理念，凡普金科集团持续致力于实现“让每个人都享有简单、公平的互联网金融服务”的使命，为有借贷、消费金融、理财知识分享、证券等需求的每一位普通人提供互联网金融服务。目前，凡普金科旗下有爱钱进、钱站、凡普信贷、任买、凡普快车、会牛、秋成等品牌，截至目前，旗下品牌合计用户数已超过2000万。

旗下主要业务

爱钱进：靠谱的互联网金融平台

钱站：借得到的借款信息服务平台

凡普信贷：个人小微借款定制化信息服务平台

任买：都市消费分期平台

会牛：更聪明的练股和选股APP

FinUp云图

自主研发的“FINUP 云图”系统，有效链接内外多元数据，通过机器学习和自然语言处理形成一个用于风控的完整知识体系。与行业内直接对知识图谱的应用不同的是，“FINUP 云图”将知识图谱与深度学习相结合，模仿人类大脑行为，自动发现隐藏在复杂关系里的风险点，挖掘潜在欺诈行为，突破传统方法的局限，是一个完整的“智能”动态风控生态系统。“FINUP 云图”风控全流程的大数据驱动已全面稳固且应用在关键业务上，由此形成了凡普金科及旗下品牌的核心优势，也因此构建出更安全、高效的运作模式。

贷前审核阶段

数据获取
反欺诈引擎
数据处理与分析

贷中管理阶段

借款人数据动态
更新实时不良状态预警

贷后管理阶段

催收评分
添加事后标签
反馈至3R 引擎处理

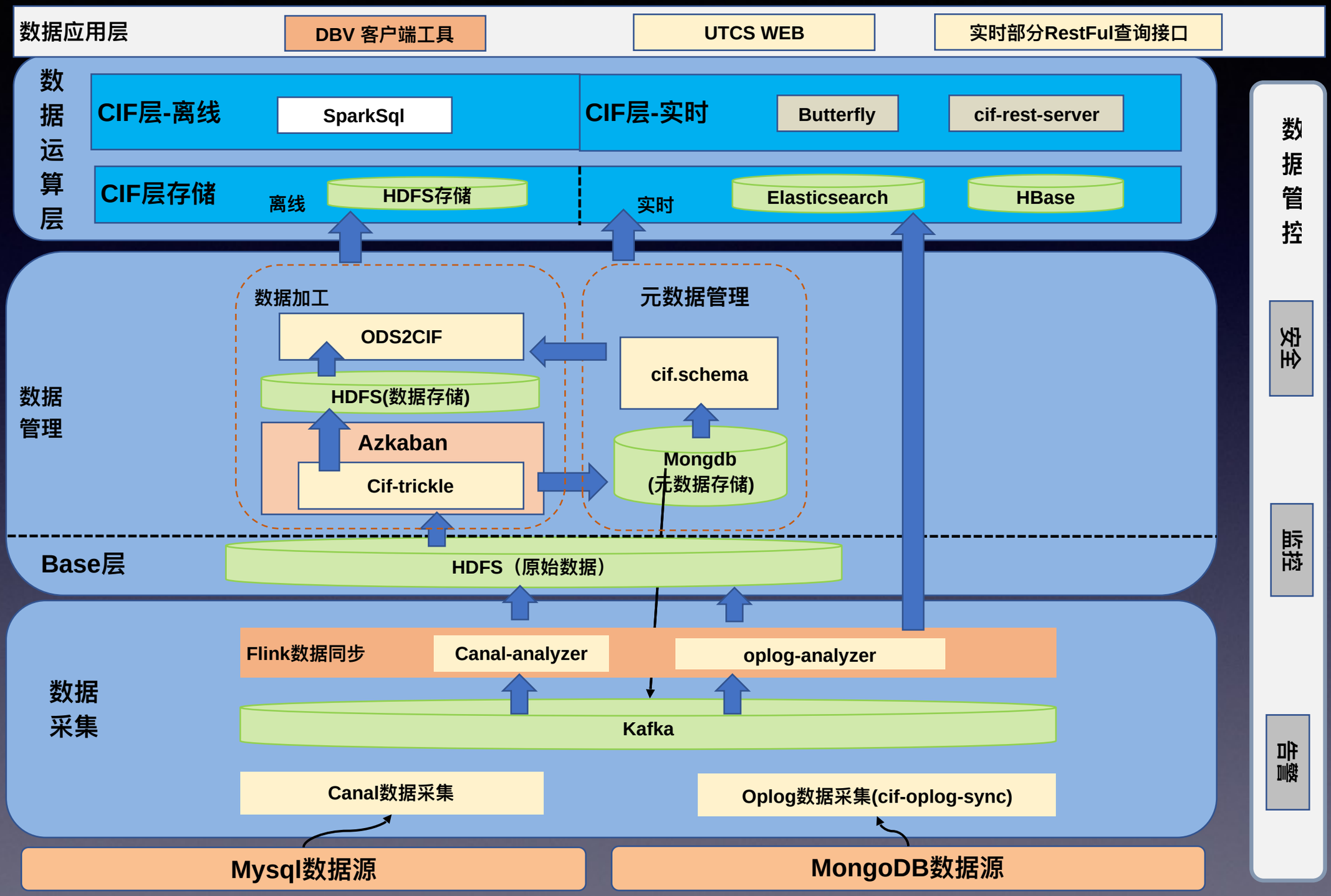
1.总体架构

2.实时数据同步-Flink

3.元数据管理

4.Butterfly-Sql计算引擎

技术架构


数据应用层
DBV 客户端工具
UTCS WEB
实时部分RestFul查询接口
数据运算层
CIF层-离线
SparkSql
CIF层-实时
Butterfly
cif-rest-server
CIF层存储
离线
HDFS存储
实时
Elasticsearch
HBase
数据管理
数据加工
ODS2CIF
HDFS(数据存储)
Azkaban
Cif-trickle
元数据管理
cif.schema
Mongdb
(元数据存储)
Base层
HDFS（原始数据）
数据采集
Flink数据同步
Canal-analyzer
oplog-analyzer
Kafka
Canal数据采集
Oplog数据采集(cif-oplog-sync)
Mysql数据源
MongoDB数据源
数据管控
安全
监控
告警

1.总体架构

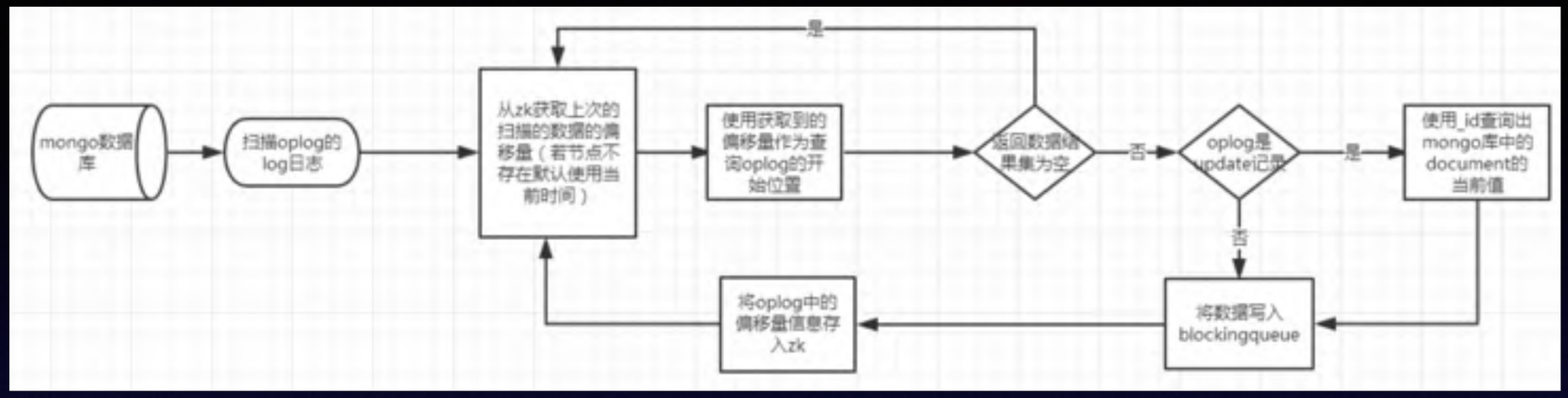
2.实时数据同步-Flink

3.元数据管理

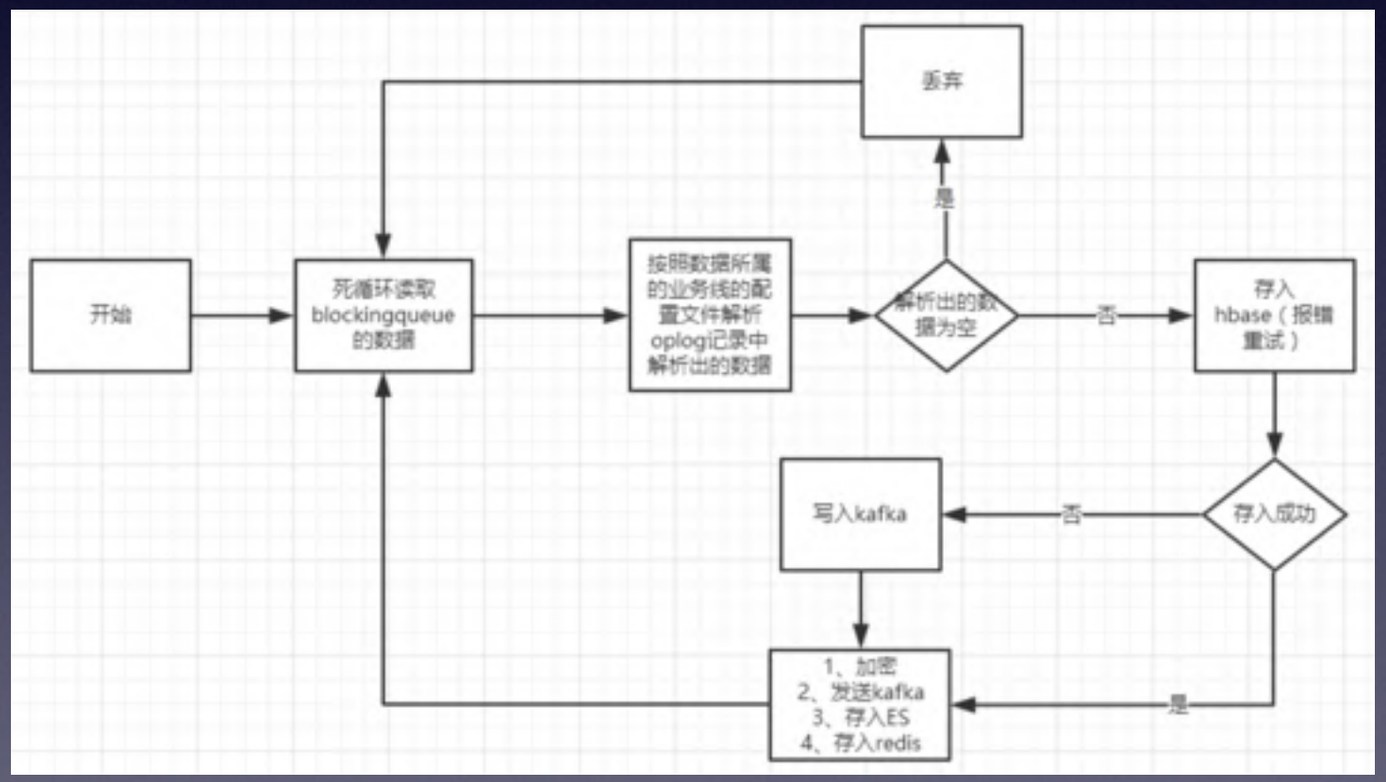
4.Butterfly-Sql计算引擎

同步数据的痛点

1.oplog扫描流程



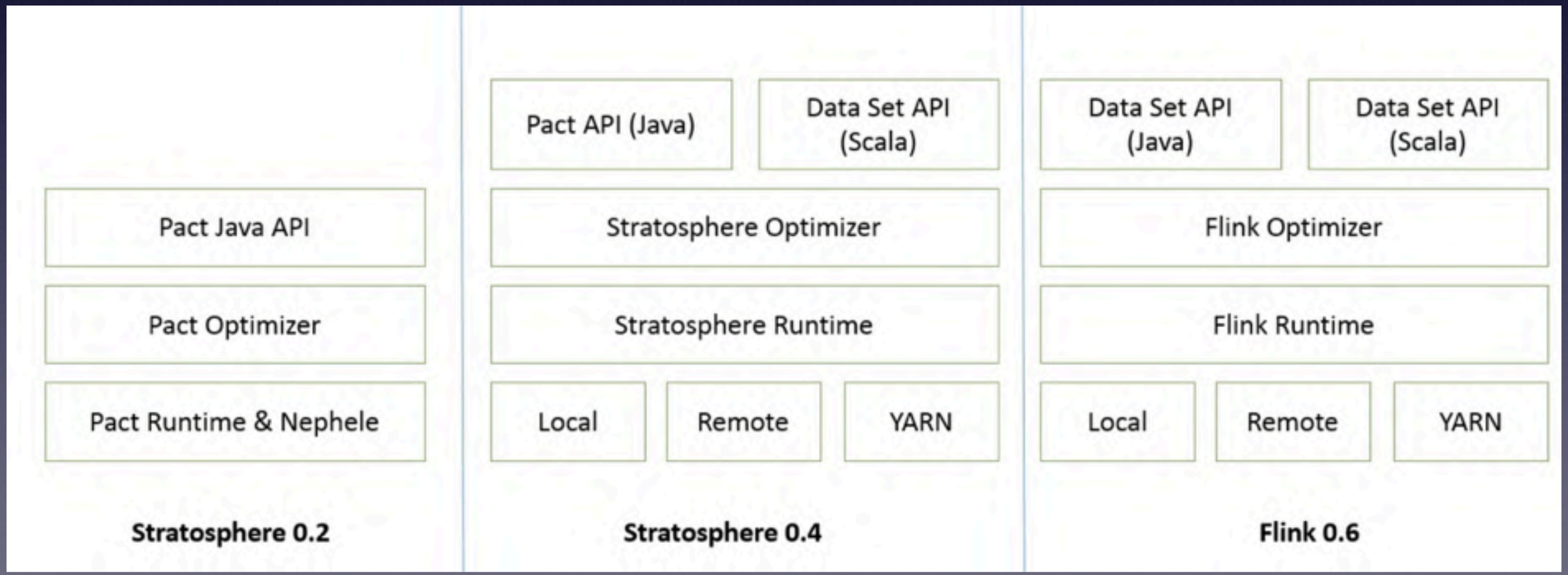
2.consumer业务流程



3.程序关闭流程

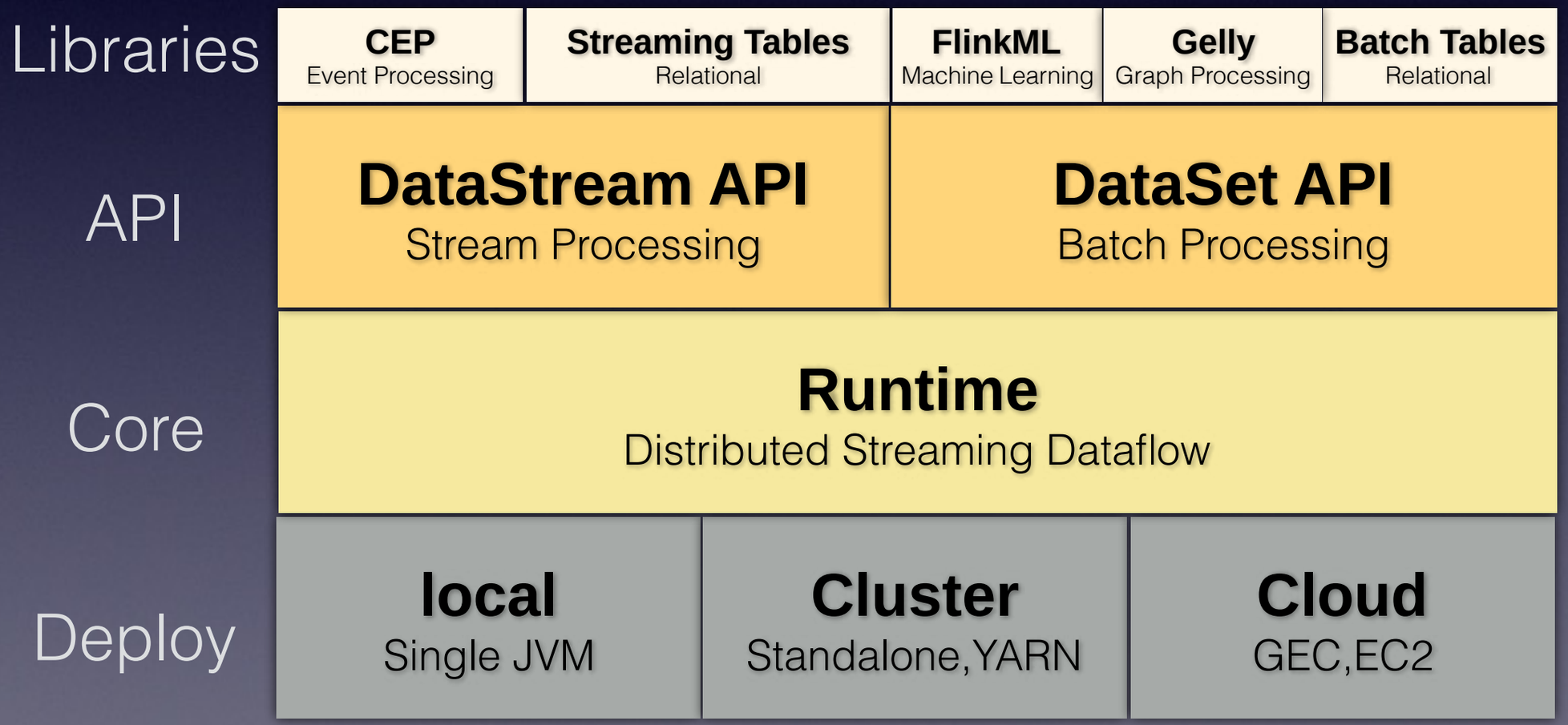
Flink

早在2008年，柏林理工大学一个名为Stratosphere研究性项目，此项目主要是构建下一代大数据分析平台，0.6版本之后改名为Flink。2014年4月16日被Apache孵化器所接受，然后迅速成为了ASF（Apache Software Foundation）的顶级项目之一

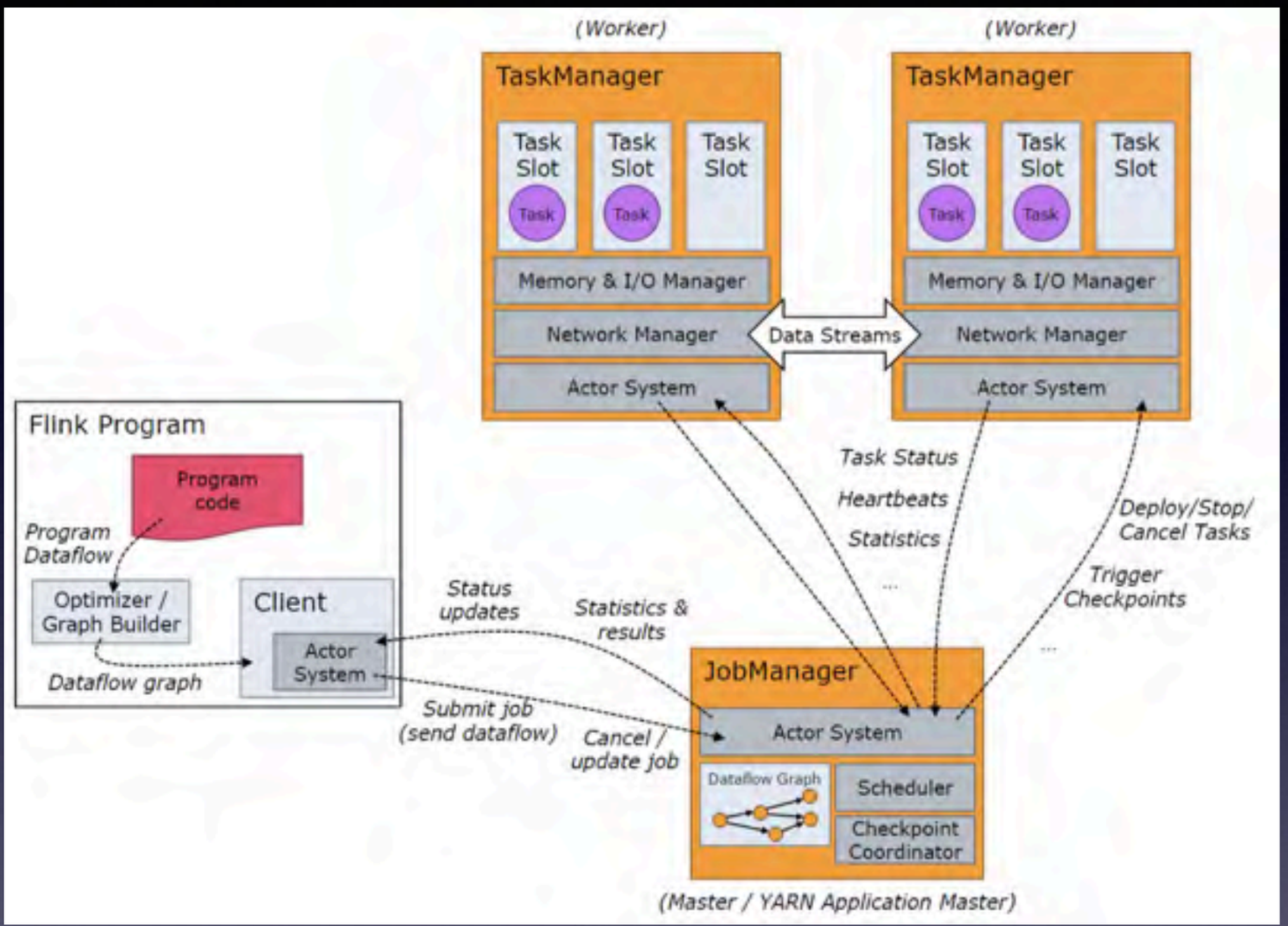


Flink Ecosystem

Flink将流处理和批处理，将二者统一起来：Flink是完全支持流处理，也就是说作为流处理看待时输入数据流是无界的；批处理被作为一种特殊的流处理，只是它的输入数据流被定义为有界的。



Flink 集群结构



Client为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。

JobManager 主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。

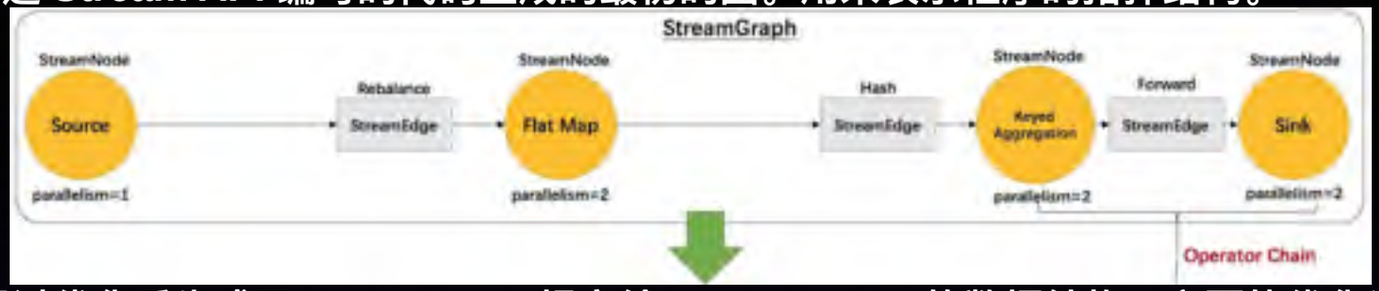
TaskManager 在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。

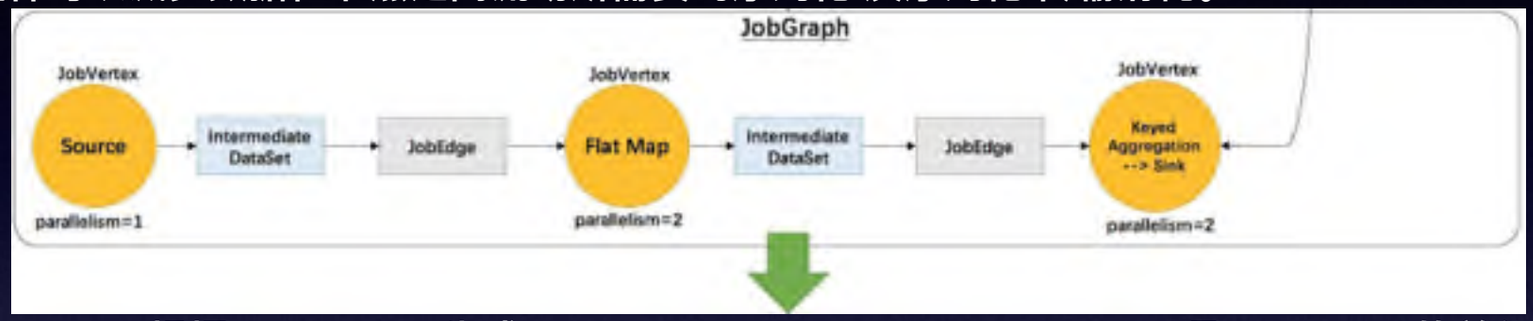
当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。三者均为独立的 JVM 进程。

Flink DAG

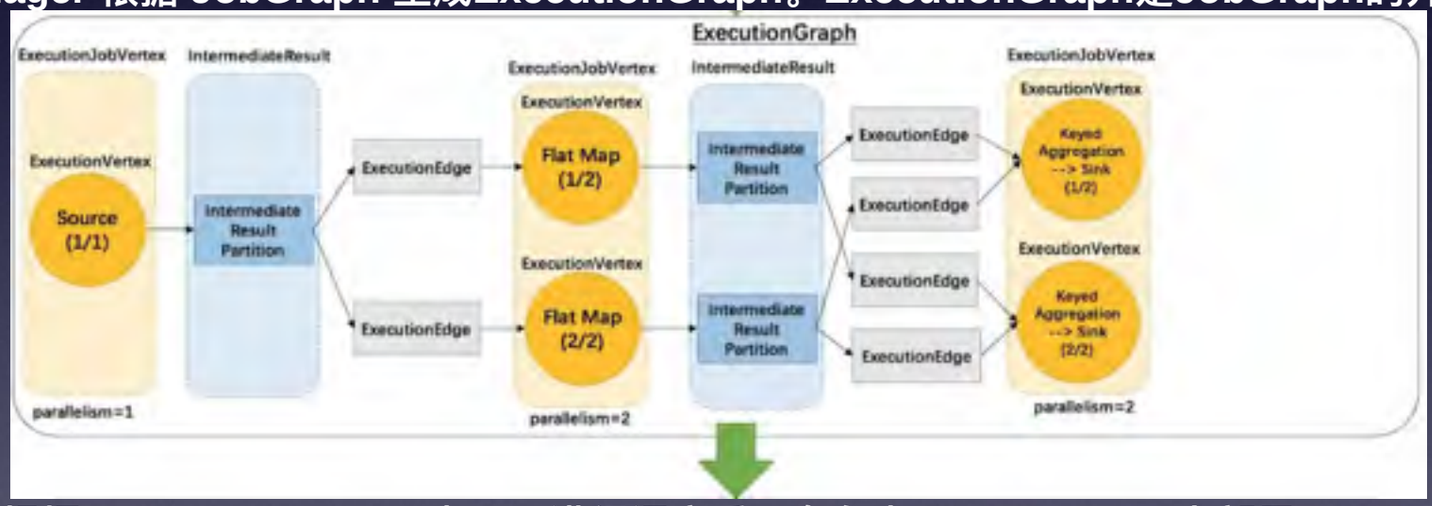
StreamGraph：是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。



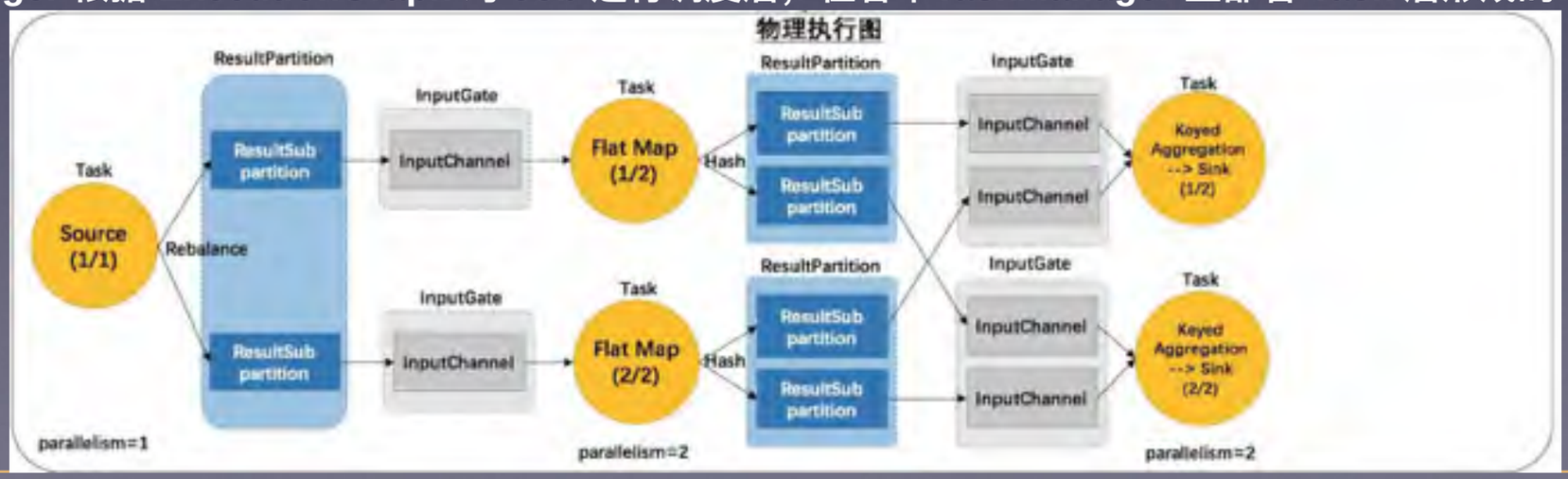
JobGraph：StreamGraph经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。



ExecutionGraph：JobManager 根据 JobGraph 生成ExecutionGraph。ExecutionGraph是JobGraph的并行化版本，是调度层最核心的数据结构。



物理执行图： JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。



Flink WaterMake

问题：

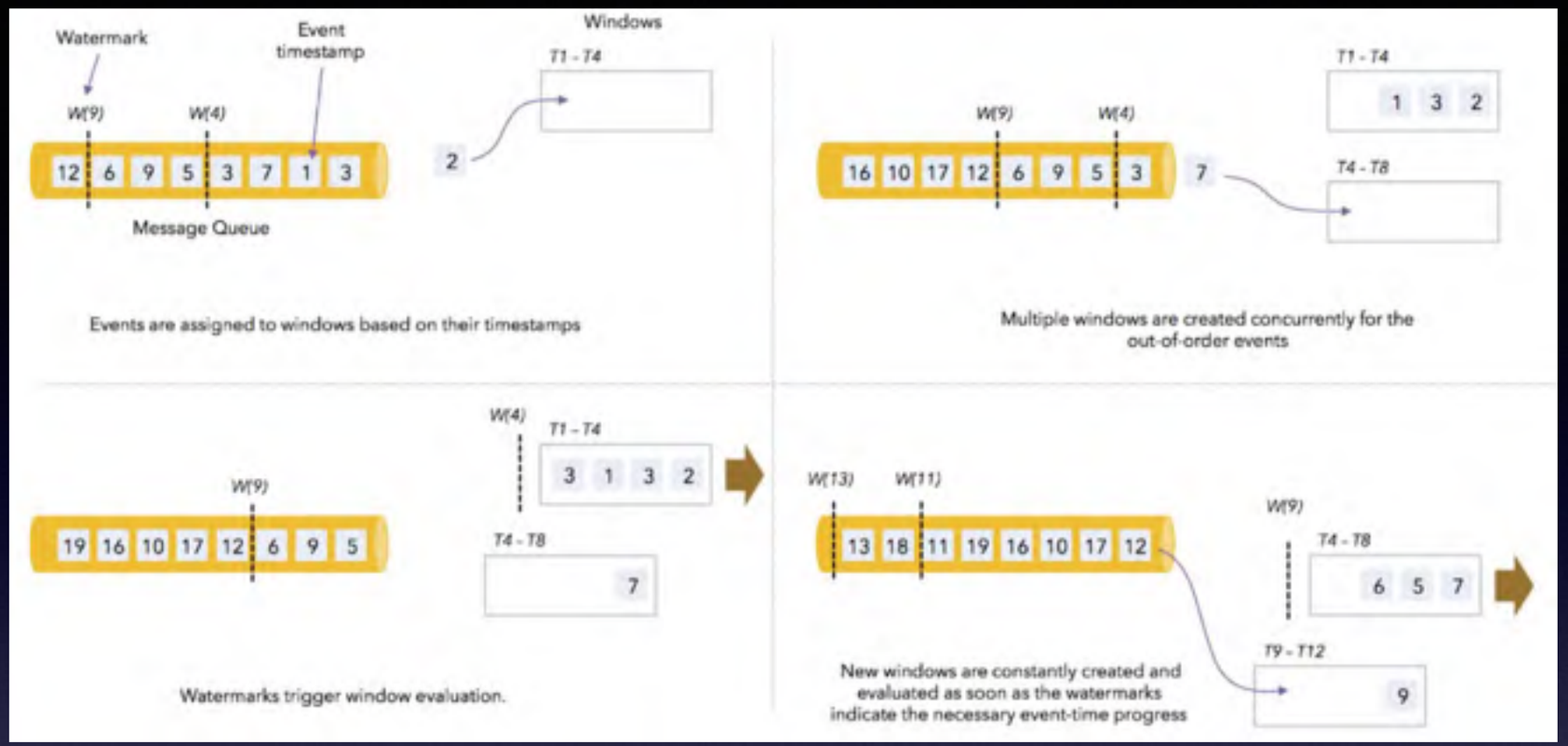
1. 消息本身带有时间戳，用户希望按照消息本身的时间特性进行分段处理。

2. 由于不同节点的时钟可能不同，以及消息在流经各个节点时延迟不同，在某个节点属于同一个时间窗口处理的消息，流到下一个节点时可能被切分到不同的时间窗口中，从而产生不符合预期的结果。

Flink借鉴了Google的MillWheel项目，通过WaterMark来支持基于Event Time时间窗口

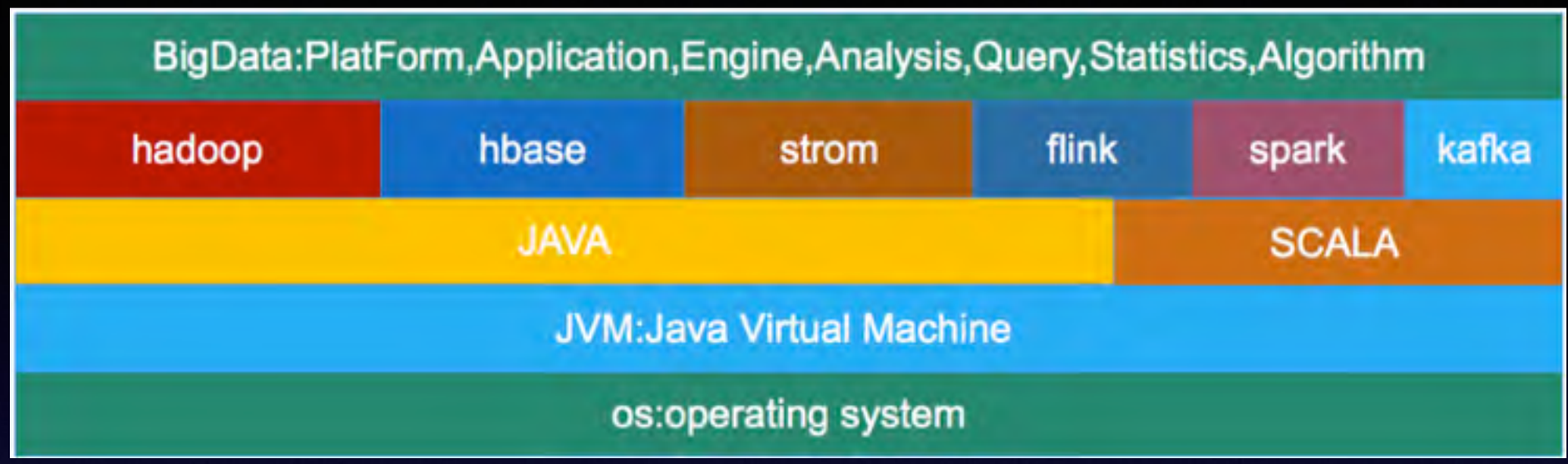
1. Event Time：表示事件创建时间
2. Ingestion Time：表示事件进入到Flink的时间
3. Processing Time：表示某个Operator对事件进行处理的本地系统时间

Flink WaterMake

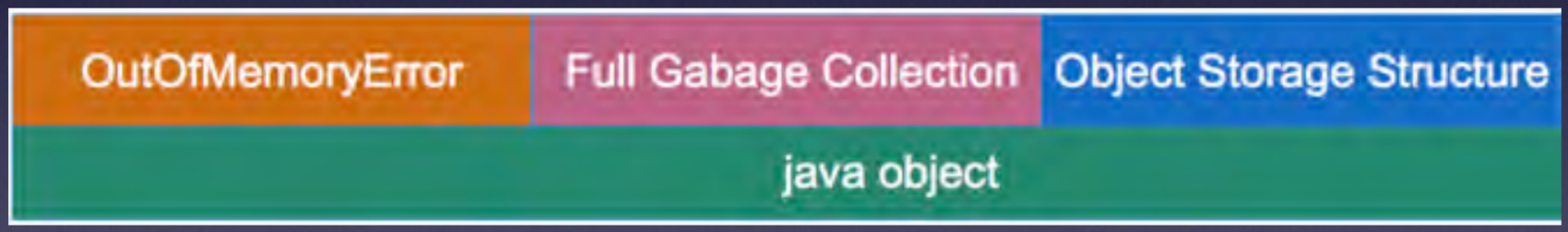


当算子通过基于Event Time的时间窗口来处理数据时，它必须在确定所有属于该时间窗口的消息全部流入此算子后才能开始数据处理。但是由于消息可能是乱序的，所以算子无法直接确认何时所有属于该时间窗口的消息全部流入此算子。WaterMark包含一个时间戳，Flink使用WaterMark标记所有小于该时间戳的消息都已流入，Flink的数据源在确认所有小于某个时间戳的消息都已输出到Flink流处理系统后，会生成一个包含该时间戳的WaterMark，插入到消息流中输出到Flink流处理系统中，Flink算子按照时间窗口缓存所有流入的消息，当算子处理到WaterMark时，它对所有小于该WaterMark时间戳的时间窗口数据进行处理并发送到下一个算子节点，然后也将WaterMark发送到下一个算子节点。

Flink 内存管理



现在大多数开源大数据处理框架都是基于JVM的,像Hadoop, Spark,Hbase,Kafka等.JVM上的程序一方面享受着它带来的好处，也要承受着JVM带来的弊端.

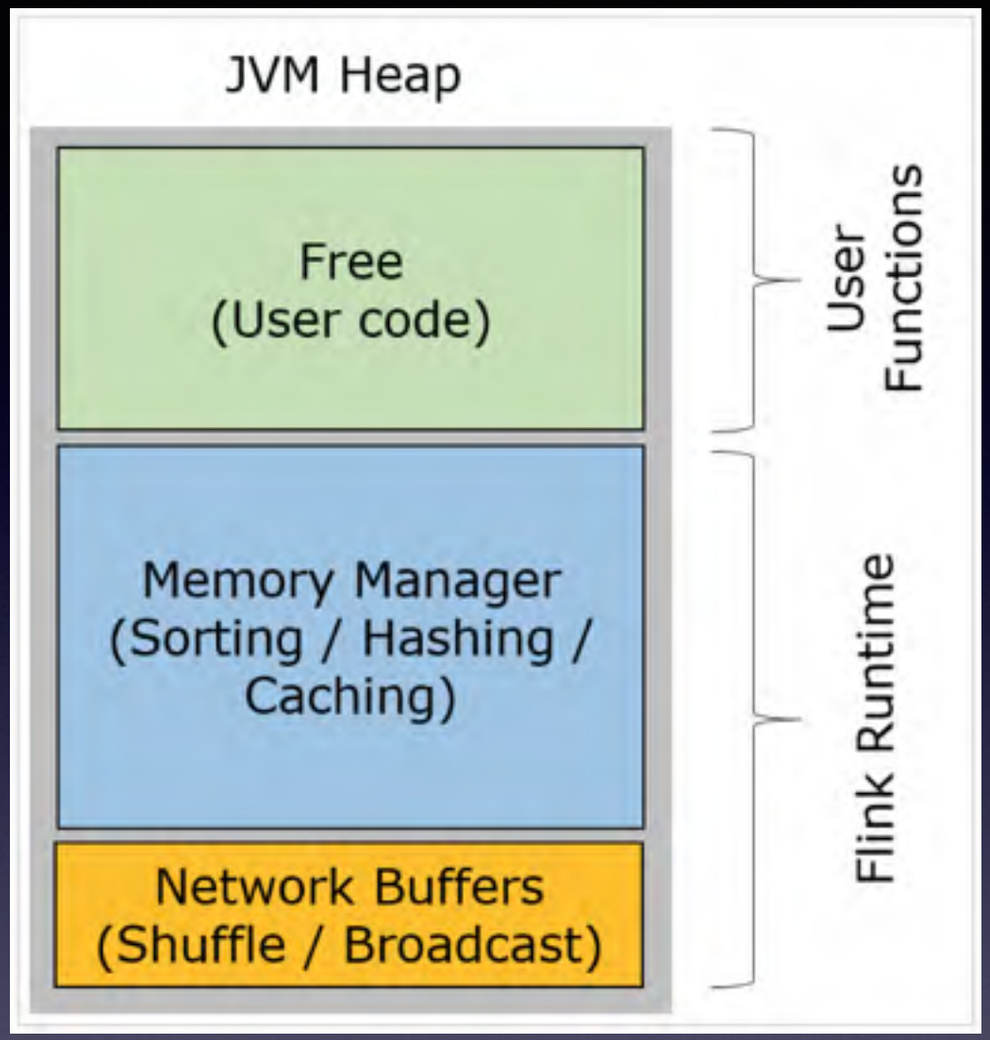


1.JVM的OOM问题

2.Full GC

3.Java对象存储密度低

Flink 内存



Network Buffers: 一定数量的32KB大小的 buffer，主要用于数据的网络传输。在 TaskManager 启动的时候就会分配。默认数量是 2048 个，可以通过 taskmanager.network.numberOfBuffers 来配置。

Memory Manager Pool: 这是一个由 MemoryManager 管理的，由众多MemorySegment组成的超大集合。Flink 中的算法（如 sort/shuffle/join）会向这个内存池申请 MemorySegment，将序列化后的数据存于其中，使用完后释放回内存池。默认情况下，池子占了堆内存的 70% 的大小。

Remaining (Free) Heap: 这部分的内存是留给用户代码以及 TaskManager 的数据结构使用的。因为这些数据结构一般都很小，所以基本上这些内存都是给用户代码使用的。从GC的角度来看，可以把这里看成的新生代，也就是说这里主要都是由用户代码生成的短期对象。

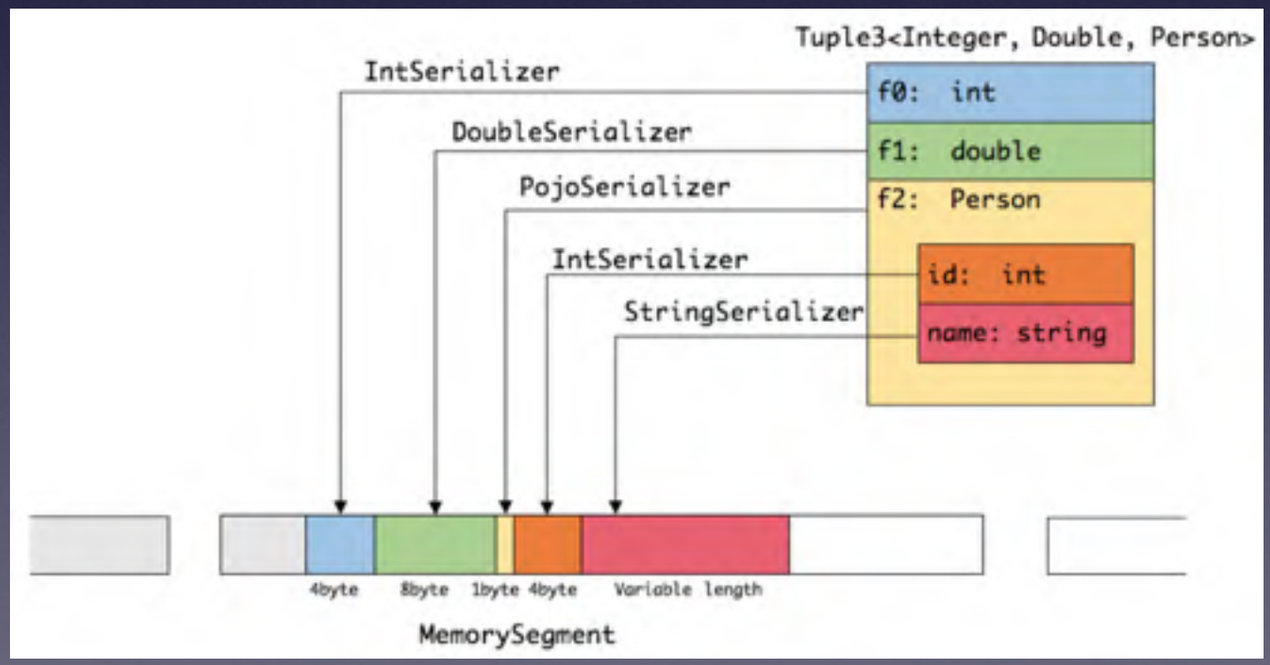
MemorySegment: flink在堆外有一块预分配的固定大小的内存块MemorySegment，flink会将对象高效的序列化到这块内存中。MemorySegment由许多小的内存cell组成，每个cell大小32kb，这也是flink分配内存的最小单位。你可以把MemorySegment想象成是为Flink 定制的 java.nio.ByteBuffer。它的底层可以是一个普通的 Java 字节数组（byte[]），也可以是一个申请在堆外的 ByteBuffer。每条记录都会以序列化的形式存储在一个或多个MemorySegment中。

Flink 序列化

Flink 实现了自己的序列化框架。对于数据集可以只保存一份对象Schema信息，节省大量的存储空间。Flink支持任意的Java或是Scala类型。Flink 能够自动识别数据类型。Flink 通过 Java Reflection 框架分析基于 Java 的 Flink 程序 UDF (User Define Function)的返回类型的类型信息，通过 Scala Compiler 分析基于 Scala 的 Flink 程序 UDF 的返回类型的类型信息。类型信息由 TypeInformation 类表示，TypeInformation 支持以下几种类型：

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类型。
WritableTypeInfo: 任意Hadoop Writable接口的实现类。
TupleTypeInfo: 任意的Flink Tuple 类型(支持Tuple1到Tuple25)。Flink tuples是固定长度固定类型的Java Tuple
CaseClassTypeInfo: 任意的 Scala CaseClass(包括 Scala tuples)。
PojoTypeInfo: 任意的POJO(Java或Scala)Java对象的所有成员变量，或public修饰符定义，或有getter/setter方法。
GenericTypeInfo: 任意无法匹配之前几种类型的类。
注意：
1.前六种类型数据集，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。
2.最后一种数据类型，Flink会使用Kryo进行序列化和反序列化。

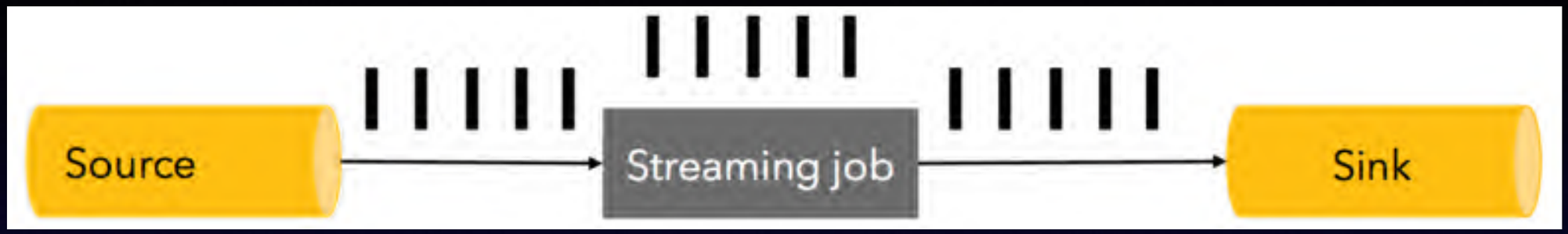
```
//1.Person类型
public class Person {
    public int id;
    public String name;
}
//Tuple3<age:Integer, height:Double, Person>对象
(25,175.5,Person(1,"zhangsan"))
```



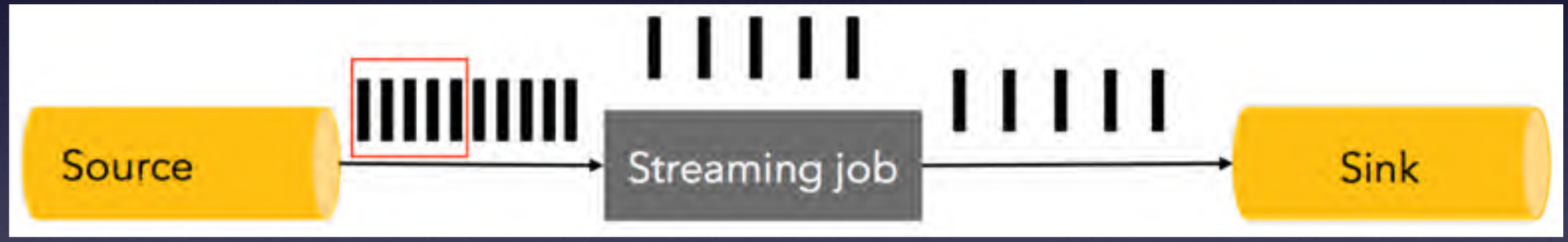
可见这种序列化方式存储密度是相当紧凑的。其中int占4字节，double占8字节，POJO多个一个字节的header，PojoSerializer只负责将header序列化进去，并委托每个字段对应的serializer对字段进行序列化。

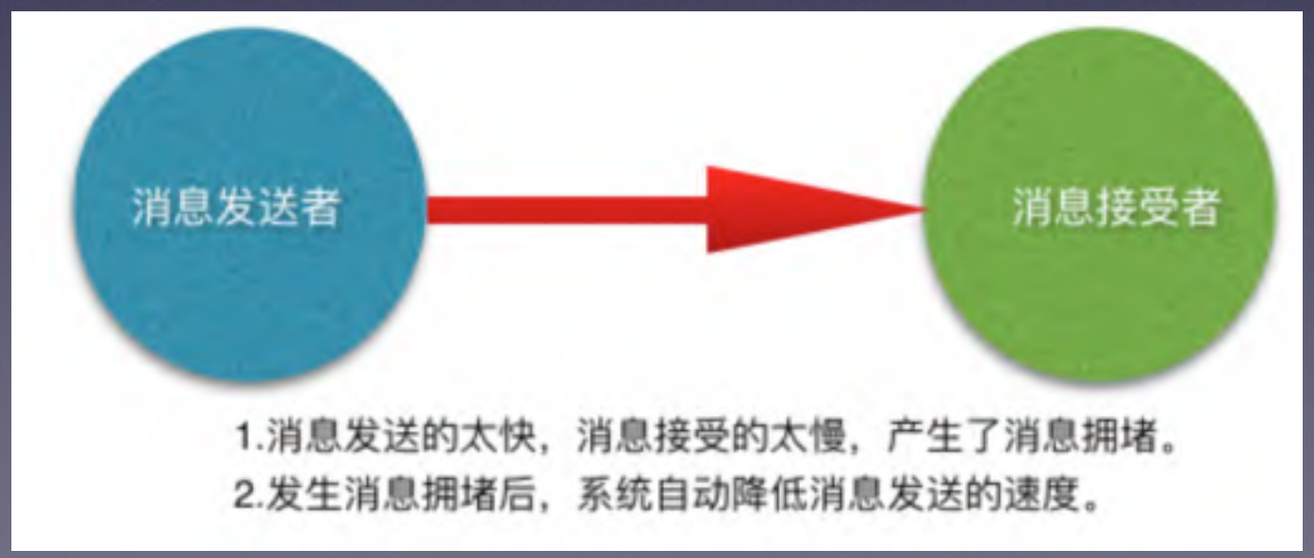
BackPressure(反压)

正常情况：消息处理速度>=消息的发送速度，不发生消息拥堵，系统运行流畅



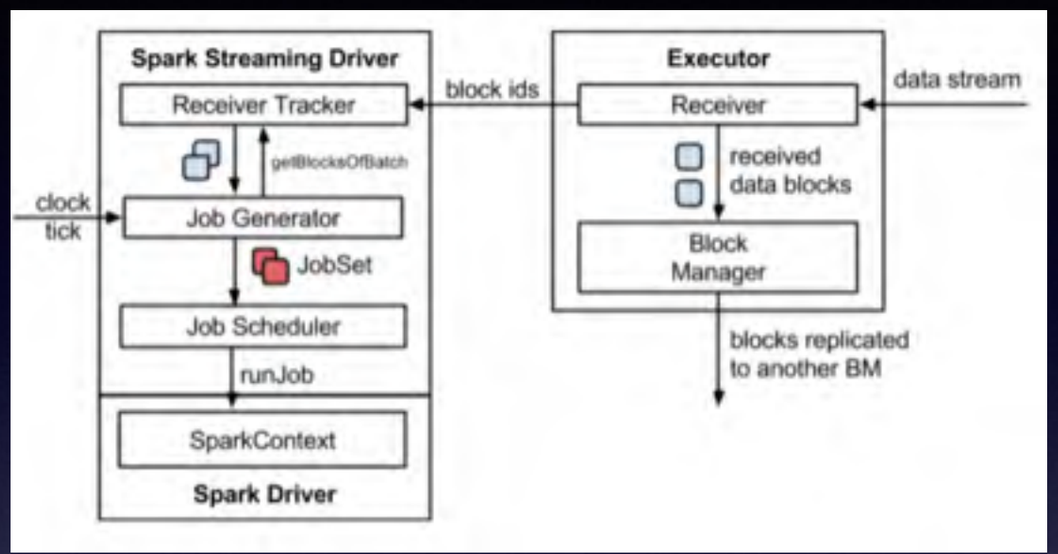
异常情况：消息处理速度< 消息的发送速度，发生了消息拥堵，系统运行不畅。

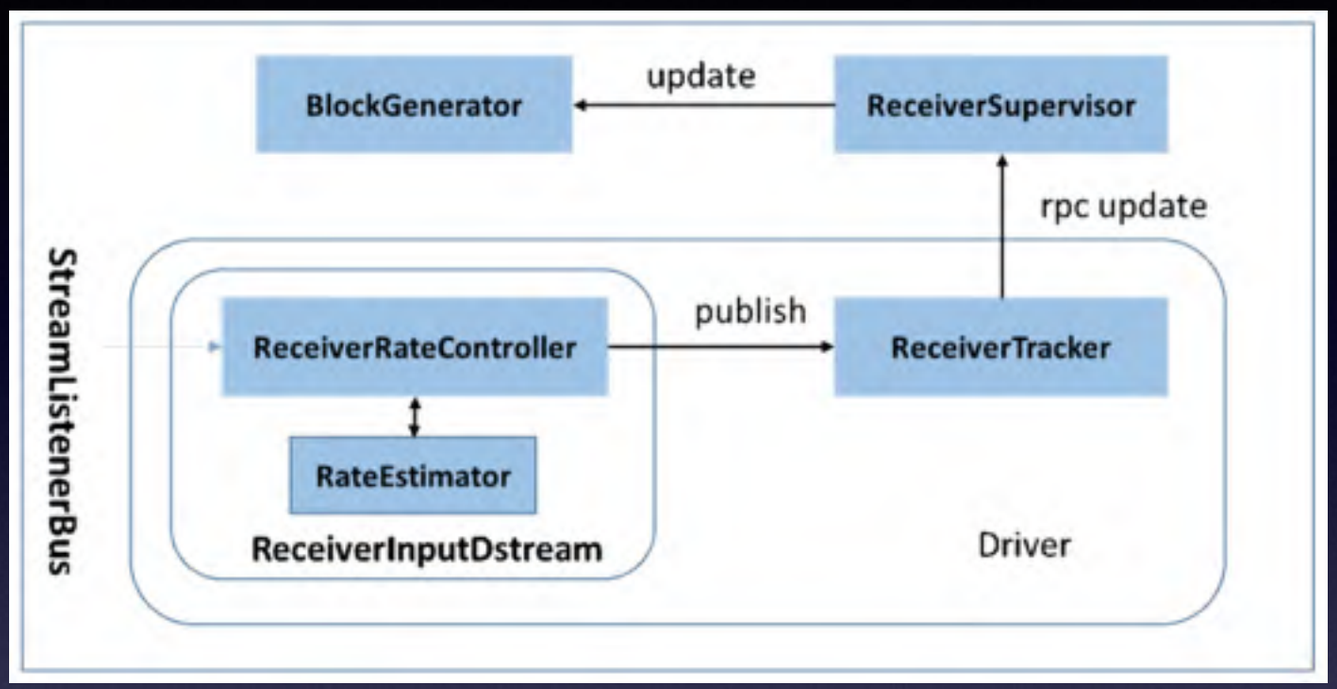




Spark BackPressure(反压)

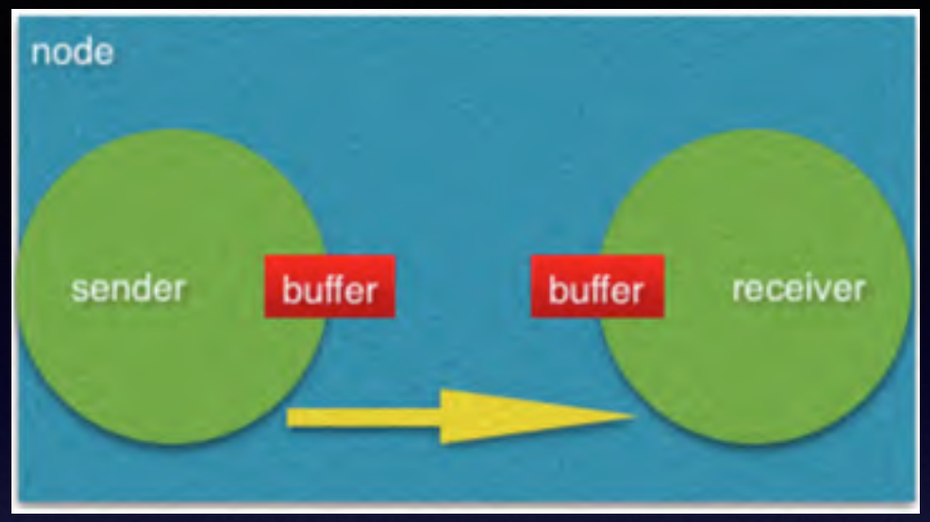
spark streaming



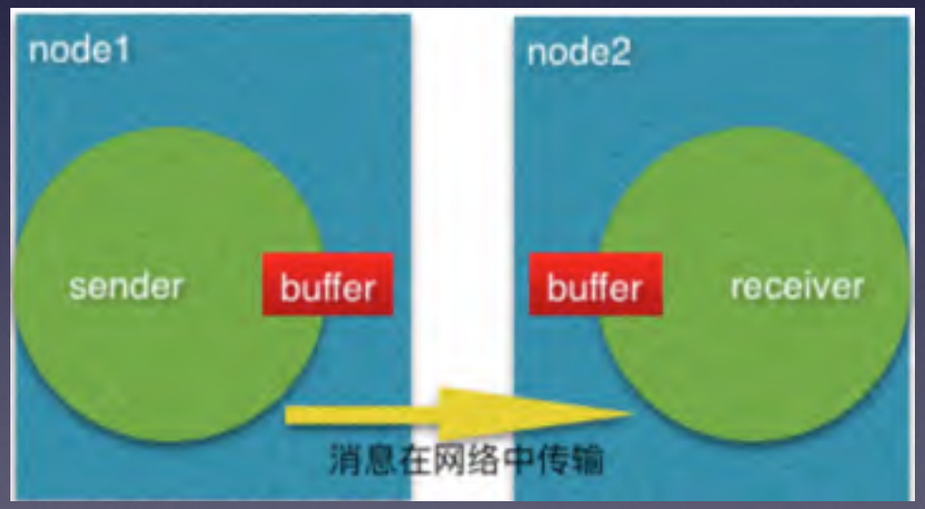


Spark 1.5以前版本，用户如果要限制Receiver的数据接收速率，可以通过设置静态配制参数spark.streaming.receiver.maxRate。
Spark Streaming 从v1.5开始引入反压机制（back-pressure）,通过动态控制数据接收速率来适配集群数据处理能力,spark.streaming.backpressure.enabled。
在原架构的基础上加上一个新的组件RateController,这个组件负责监听OnBatchCompleted事件,然后从中抽取processingDelay 及schedulingDelay信息. Estimator依据这些信息估算出最大处理速度（rate）， 最后由基于Receiver的Input Stream将rate通过ReceiverTracker与ReceiverSupervisorImpl转发给BlockGenerator。

Flink BackPressure(反压)

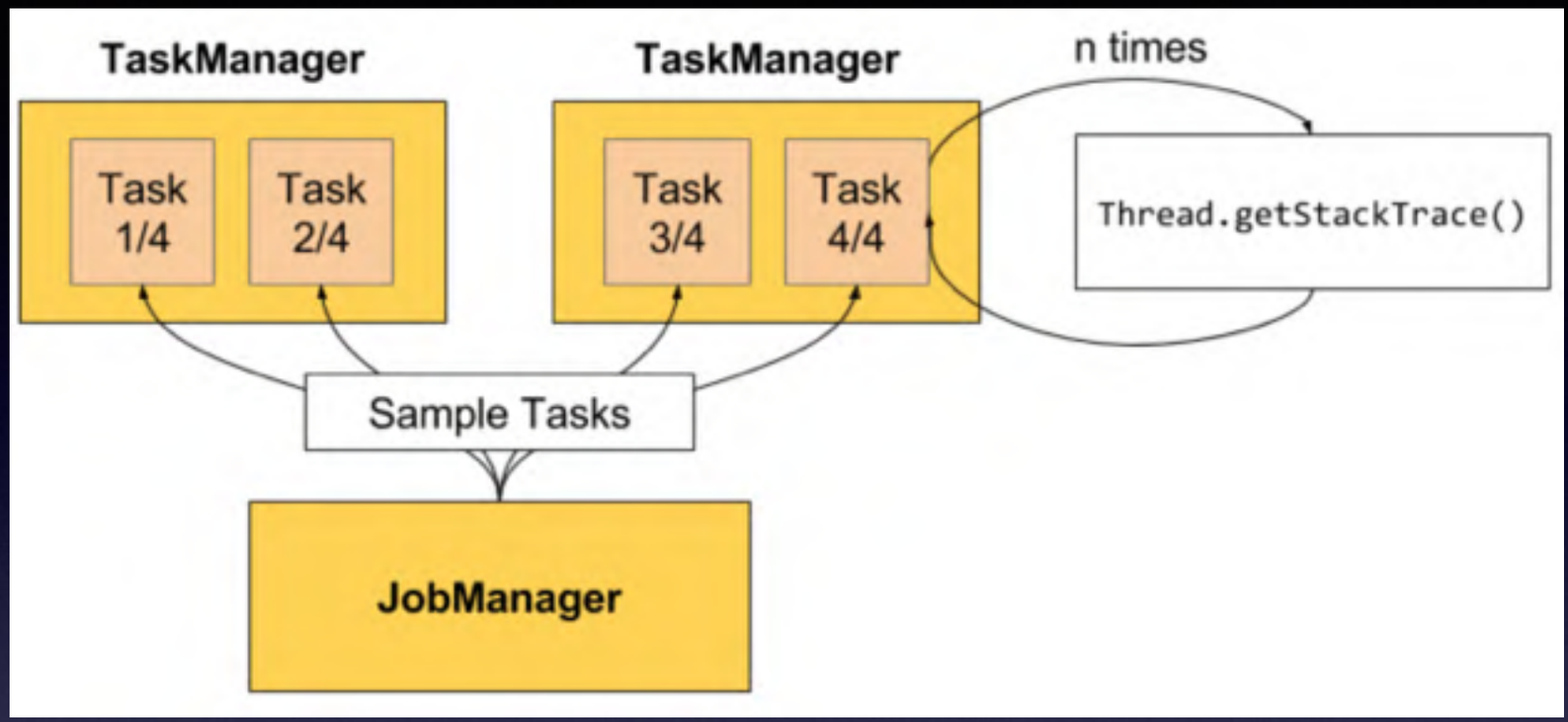


本地传输：如果task1和task2都运行在同一个工作节（TaskManager），缓冲区可以被直接共享给下一个task，一旦task2消费了数据它会 被回收。如果task 2比task1慢，task1可用的缓冲区从而迫使task1降速。



网络传输：如果task1和task2运行在不同的工作节点上。一旦缓冲区内的数据被发送出去(TCP Channel)，它就会被回收。在接收端，数据被拷贝到输入缓冲池的缓冲区中，如果没有缓冲区可用，从TCP连接中的数据读取动作将会被中断。如果有足够的数据已经进入可发送状态，会等到情况稳定到阈值以下才会进行发送。这可以保证没有太多的数据在路上。如果新的数据在消费端没有被消费（因为没有可用的缓冲区），这种情况会降低发送者发送数据的速度。

Flink BackPressure(反压) 监控



Flink Web界面上提供了对运行Job的Backpressure行为的监控，它通过使用Sampling线程对正在运行的Task进行堆栈跟踪采样来实现。默认情况下，JobManager会每间隔50ms触发对一个Job的每个Task依次进行100次堆栈跟踪调用，过计算得到一个比值，例如，radio=0.01，表示100次中仅有1次方法调用阻塞。

Flink目前定义了如下Backpressure状态：
OK: 0 <= Ratio <= 0.10
LOW: 0.10 < Ratio <= 0.5
HIGH: 0.5 < Ratio <= 1

Flink 容错

消息可靠性语义

1. At most once 消息可能会丢，但绝不会重复传输

2. At least one 消息绝不会丢，但可能会重复传输

3. Exactly once 每条消息肯定会被传输一次且仅传输一次

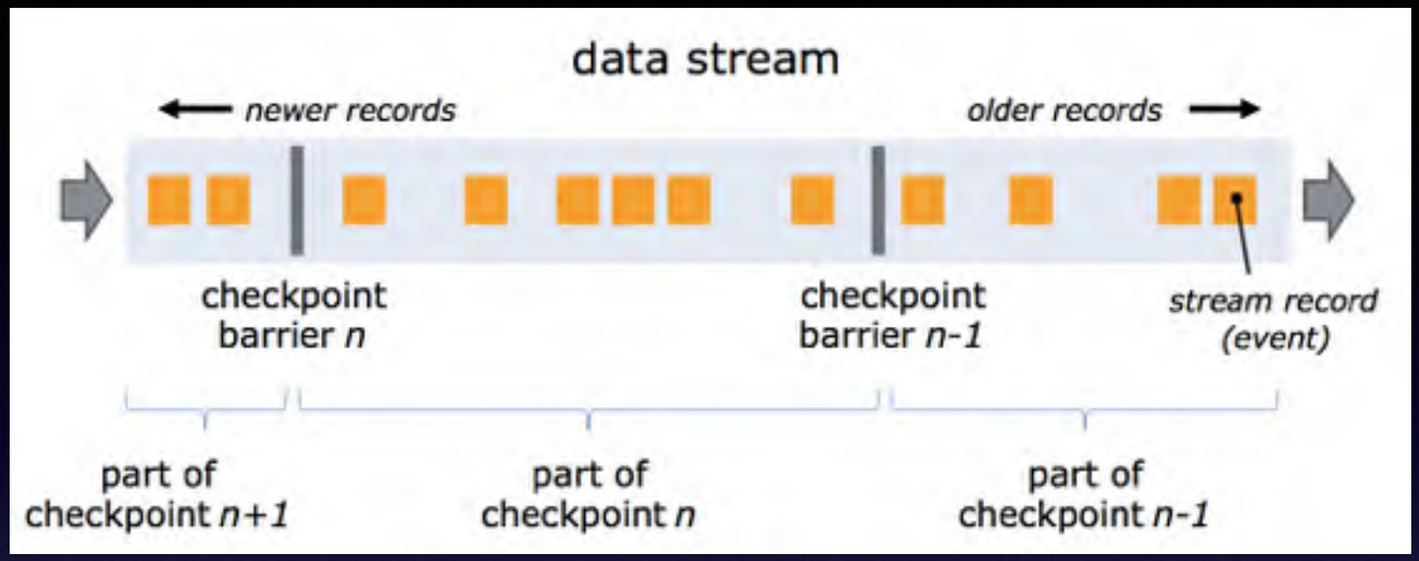
Flink 容错机制的核心就是持续创建分布式数据流及其状态的一致快照。这些快照在系统遇到故障时，充当可以回退的一致性检查点（checkpoint）。基于轻量级分布式快照。Lightweight Asynchronous Snapshots for Distributed Dataflows 描述了Flink 创建快照的机制实现的容错。此论文是受分布式快照算法 Chandy-Lamport 启发，并针对 Flink 执行模型量身定制。

简而言之，Flink的分布式快照（snapshot）。其包含两方面：

1.数据源所有数据的位置

2.并行操作的状态

Flink checkpoint



Asynchronous Barrier Snapshots

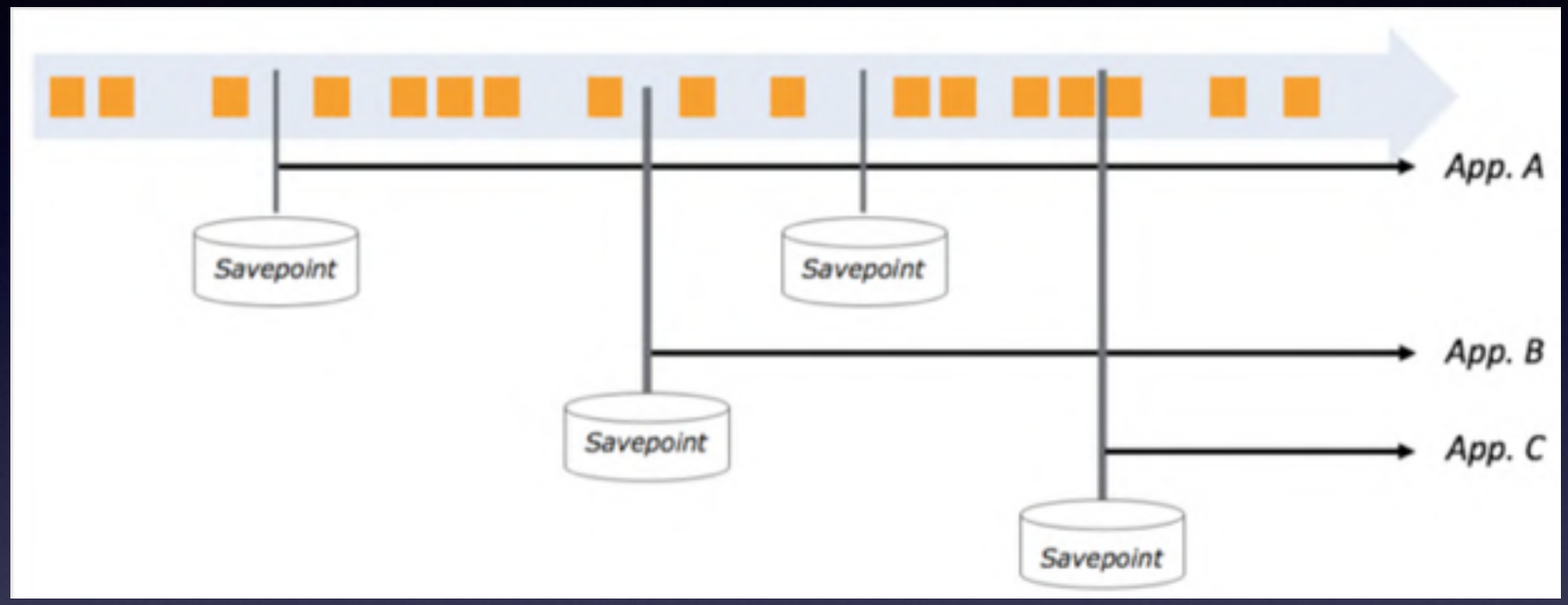
Barrier机制：Flink 分布式快照的核心概念之一就是数据栅栏（barrier）。这些 barrier 被插入到数据流中，作为数据流的一部分和数据一起向下流动。Barrier 不会干扰正常数据，数据流严格有序。一个 barrier 把数据流分割成两部分：一部分进入到当前快照，另一部分进入下一个快照。每一个 barrier 都带有快照 ID，并且 barrier 之前的数据都进入了此快照。Barrier 不会干扰数据流处理，所以非常轻量。多个不同快照的多个 barrier 会在流中同时出现，即多个快照可能同时创建。

```
// start a checkpoint every 1000 ms
env.enableCheckpointing(1000)
// advanced options:

// set mode to exactly-once (this is the default)
env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)

// checkpoints have to complete within one minute, or are discarded
env.getCheckpointConfig.setCheckpointTimeout(60000)

// allow only one checkpoint to be in progress at the same time
env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)
```

Flink提供了选项，可以关闭Exactly once并仅保留at least once，以提供最大限度的吞吐能力。

Flink savepoint

简单的说：checkpoint是Flink实现容错的，savepoint仅仅只是checkpoint的一个扩展。如果checkpoint开启，那Flink会周期性的创建所有操作状态的checkpoint。savepoint和checkpoint最大的不同是，checkpoint会按时间间隔自动创建，而savepoint需要手动触发。



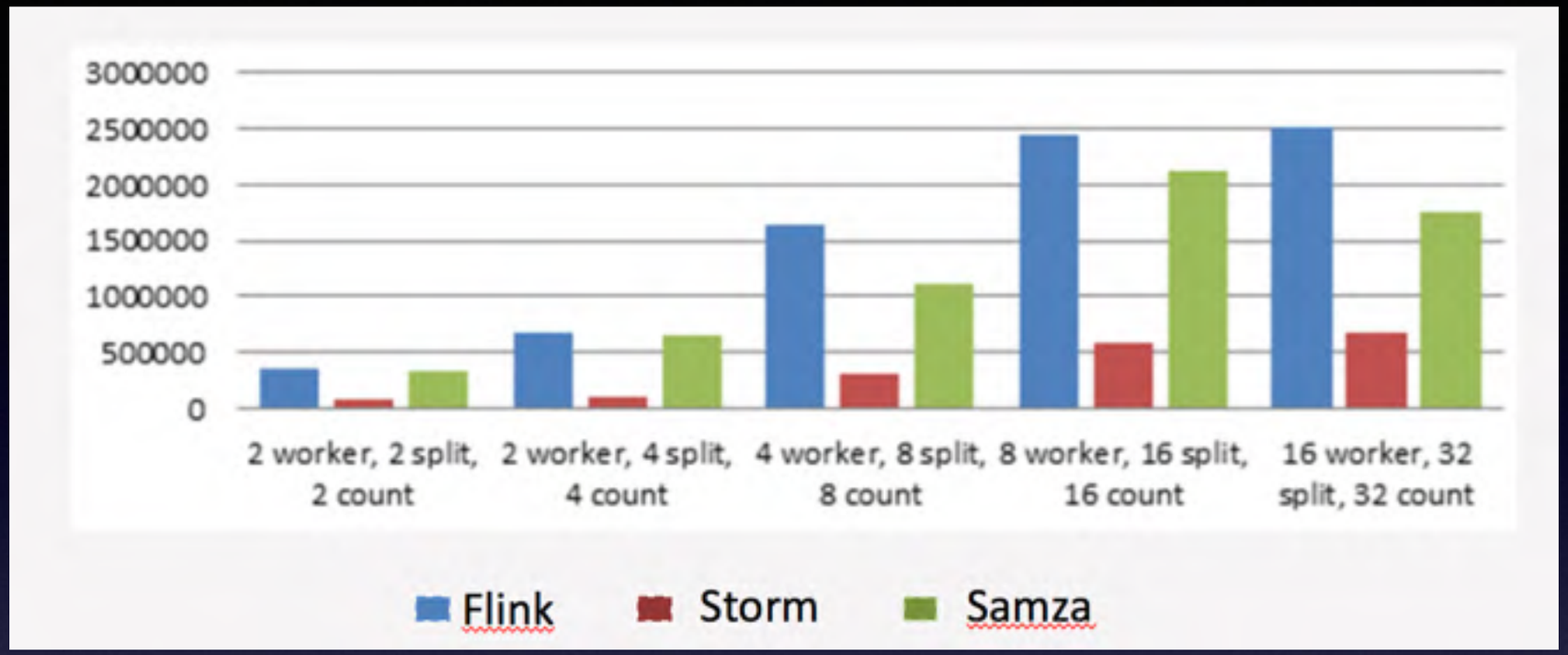
flink list

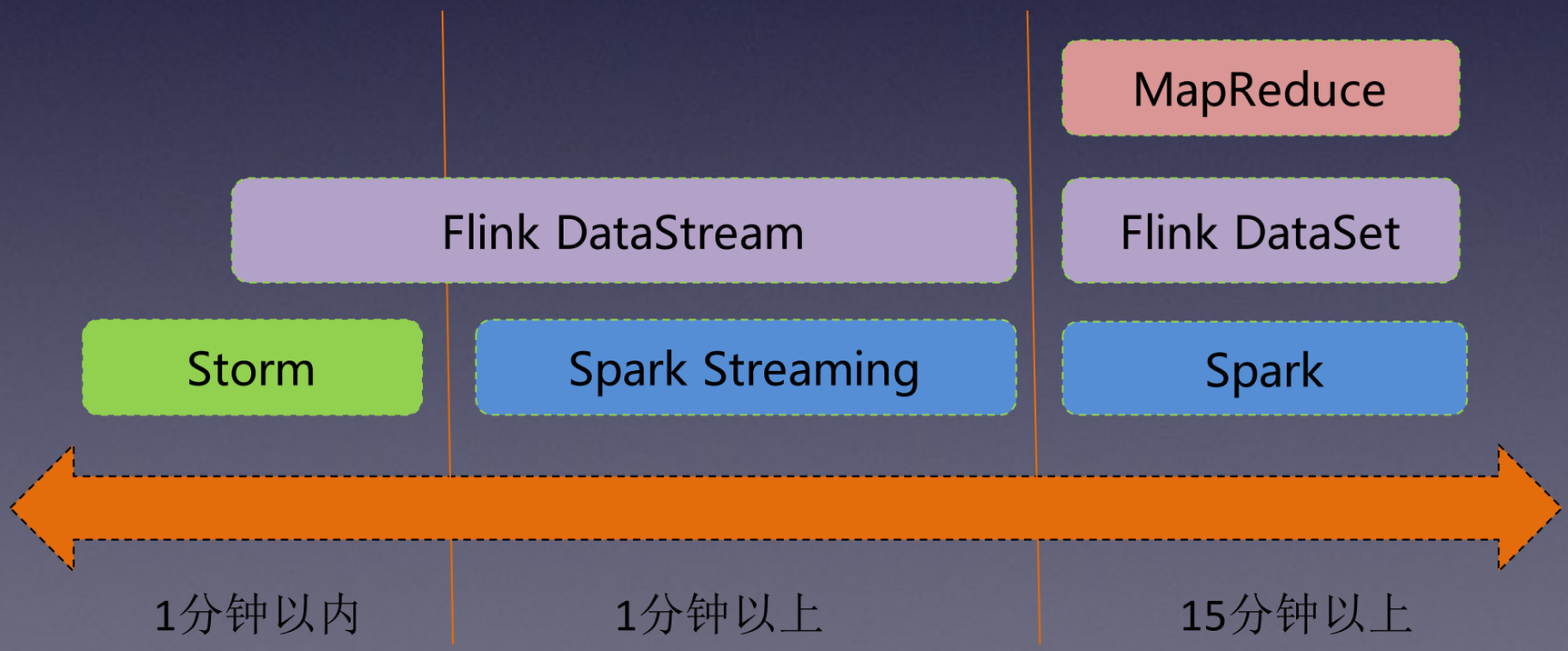
flink savepoint job_id

flink cancel job_id

flink run -d -s hdfs://savepoint/1 ###.jar

对比


3000000
2500000
2000000
1500000
1000000
500000
0
2 worker, 2 split, 2 count
2 worker, 4 split, 4 count
4 worker, 8 split, 8 count
8 worker, 16 split, 16 count
16 worker, 32 split, 32 count
Flink
Storm
Samza


MapReduce
Flink DataStream
Flink DataSet
Storm
Spark Streaming
Spark
1分钟以内
1分钟以上
15分钟以上

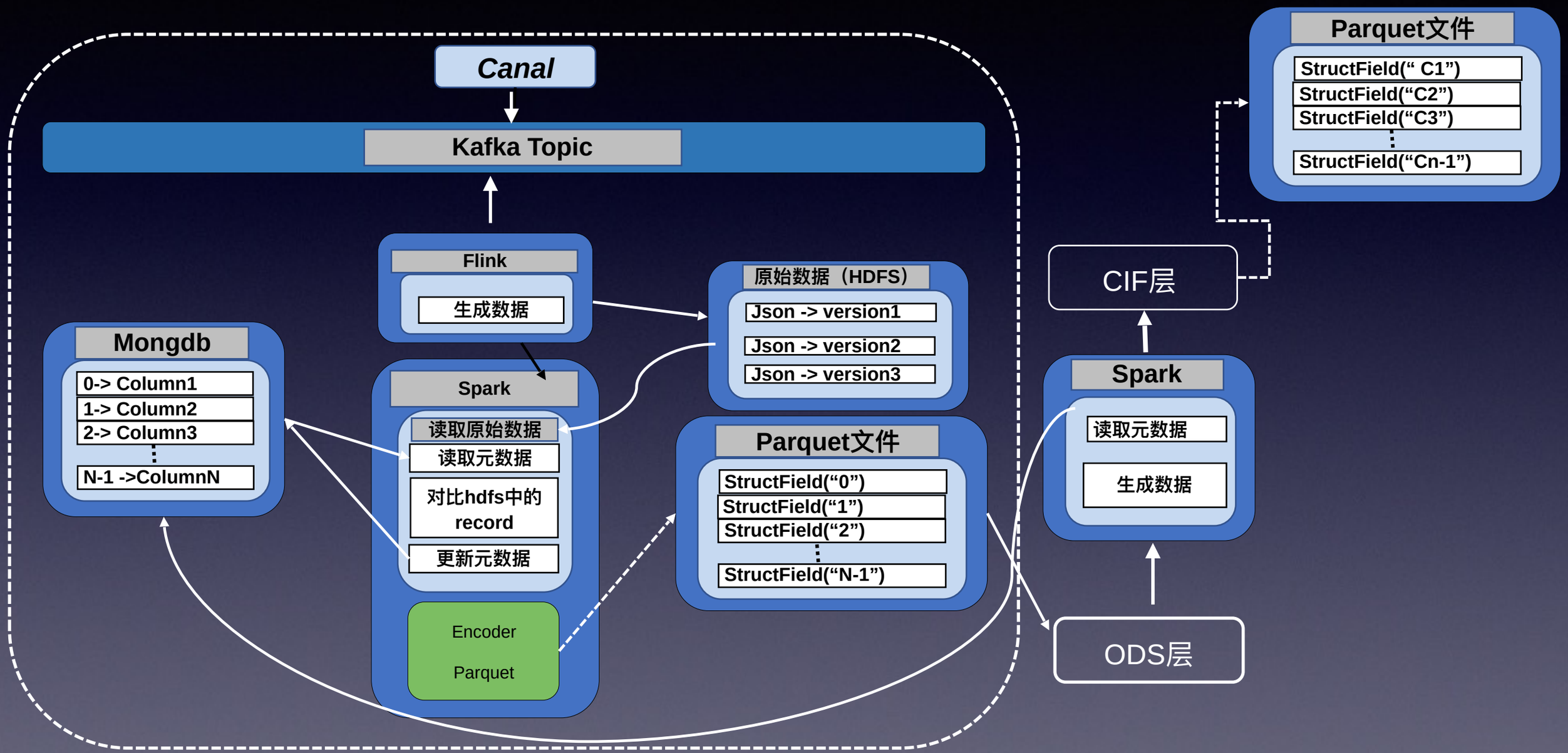
1.总体架构

2.实时数据同步-Flink

3.元数据管理

4.Butterfly-Sql计算引擎

Trickle


Canal
Kafka Topic
Flink
生成数据
Mongdb
0-> Column1
1-> Column2
2-> Column3
N-1 ->ColumnN
Spark
读取原始数据
读取元数据
对比hdfs中的record
更新元数据
Encoder
Parquet
原始数据（HDFS）
Json -> version1
Json -> version2
Json -> version3
Parquet文件
StructField(“0”)
StructField(“1”)
StructField(“2”)
StructField(“N-1”)
Parquet文件
StructField(“ C1”)
StructField(“C2”)
StructField(“C3”)
StructField(“Cn-1”)
CIF层
Spark
读取元数据
生成数据
ODS层

Schema 数据结构


```
{
    "_id" : ObjectId("58c692ad2b3507b26aa761f8"),
    "key" : "puhui.company_account",
    "cols" : [
        {
            "name" : "cif_tr_sortTime",
            "colType" : "STRING",
            "originType" : "long",
            "time" : NumberLong(1489408669055),
            "optionFlag" : "Normal",
            "index" : 0
        },
        {
            "name" : "id",
            "colType" : "long",
            "originType" : "bigint",
            "time" : NumberLong(1489408669055),
            "optionFlag" : "Normal",
            "index" : 1
        },
        {
            "name" : "name",
            "colType" : "string",
            "originType" : "varchar",
            "time" : NumberLong(1489408669055),
            "optionFlag" : "Detele",
            "index" : 2
        }
    ]
}
```


库名+表名
表中字段的名称
映射成cif表中的字段类型
原始数据在mysql/mongo的类型
更新到shema数据结构的时间戳(字段更新的依据)
标识是否删除
在parquet文件字段的名称

1.总体架构

2.实时数据同步-Flink

3.元数据管理

4.Butterfly-Sql计算引擎

解析器-Scala Parsers

众所周知Scala是一个是一门多范式的编程语，除了能给强大函数支持以外，scala还有提供强大的语法解析功能，它就是Scala parser combinators，如果你需要构建一个解析引擎，如：json解析，sql解析，甚至是构建一门编程语言。

Scala parser combinators，它完全是从 Scala 的函数性方面构建的。解析器组合子使我们可以将语言的各种片段 “组合” 成部件，这些部件可以提供不需要代码生成，而且看上去像是一种语言规范的解决方案。而且语言级别原生支持的。Scala parser combinators 有两种解析器，一个是基于正则的解析器RegexParsers，另一个是基于词法的文法解析的解析器StandardTokenParsers。StandardTokenParsers十分强大，著名的Spark sql的sql解析器就是基于StandardTokenParsers开发出来的。

怎么来构建这个DSL？我们不得不提到编译理论中最精华的部分，一个语言处理器（包括解释器和编译器）的基本运算至少由两个阶段组成：
解析器，用于获取输入的文本并将其转换成 Abstract Syntax Tree（AST）。
代码生成器（在编译器的情况下），用于获取 AST 并从中生成所需字节码；或是求值器（在解释器的情况下），用于获取 AST 并计算它在 AST 里面所发现的内容。

BNF

DSL的语法规则是基于BNF，BNF是描述编程语言的文法。自然语言存在不同程度的二义性。这种模糊、不确定的方式无法精确定义一门程序设计语言。必须设计一种准确无误地描述程序设计语言的语法结构，这种严谨、简洁、易读的形式规则描述的语言结构模型称为文法。最著名的文法描述形式是由Backus定义Algol60语言时提出的Backus-Naur范式（Backus-Naur Form, BNF）及其扩展形式EBNF。BNF能以一种简洁、灵活的方式描述语言的语法。具体内容可参考针对编译原理的书。现在，几乎新编程语言都使用巴科斯范式来定义编程语言的语法规则。

在BNF中，双引号中的字("word")代表着这些字符本身。而double_quote用来代表双引号。

在双引号外的字（有可能有下划线）代表着语法部分。

< > : 内包含的为必选项。

[ ] : 内包含的为可选项。

{ } : 内包含的为可重复0至无数次的项。

| : 表示在其左右两边任选一项，相当于"OR"的意思。

::= : 是“被定义为”的意思

"..." : 术语符号

[...] : 选项，最多出现一次

{...} : 重复项，任意次数，包括 0 次

(...) : 分组

| : 并列选项，只能选一个

用 BNF 形式来表达

```
expr   ::= term {'+' term | '-' term}
term   ::= factor {'*' factor | '/' factor}
factor ::= floatingPointNumber | '(' expr ')'
```

其中花括号（{}）表明内容可能重复（0 次或多次），竖线（|）表明或的关系。因此一个 factor 可能是一个 floatingPointNumber，或者一个左括号加上一个 expr 再加上一个右括号。

StandardTokenParsers

使用Scala基于词法单元的解析器解析上述EBNF文法，Scala基于词法单元的解析器是需要继承StandardTokenParsers这个类的，该类提供了很方便的解析函数，以及词法集合。
可以通过使用lexical.delimiters列表来存放在文法翻译器执行过程中遇到的分隔符，使用lexical.reserved列表来存放执行过程中的关键字。比如，DSL中，看到”+","-","*","/","(",")"这些都是分隔符，其实我们也可以把它们当做是关键字，但是我习惯上将带有英文字母的单词作为关键字处理。因而，关键字集合便是存放在lexical.reserve这里，当然这个计算器的例子没有关键字，如果有"if","then","SUM","COUNT"这些，就要放到lexical.reserve

主要的类/接口有：

* Parser：一个或者一类输入的解析类，多个Parser之间可以互相组合
* Parsers：定义输入的类型，定义/组合/连接多个Parser
* Reader：输入源抽象，包括字符串/流等实现
* ParseResult：解析结果，成功/失败/错误等
* Positional：定位源的位置
* Position：表示位置

连接符:Parser有几个主要的方法，通过这些方法，可以完成多个parser的组合。

* | 选择连接符，当左边或右边的Parser成功时成功。
* ~ 序列连接符，当左右的Parser都成功时成功
* ~> 和~类似 不同的是左边的结果不会包含在最后的结果中
* <~ 和~类似 右边的结果不会包含在最后的结果中
* ^^ 转换连接符，当左边的操作成功时可以通过这个连接符操作结果
* opt(p)：可选的匹配，匹配成功返回Some(x)，失败返回None
* rep repeat的意思 重复

Butterfly

```scala
case class SelectStmt(projections: Seq[SqlProj],
                      relations: Option[Seq[SqlRelation]],
                      filter: Option[SqlExpr],
                      groupBy: Option[SqlGroupBy],
                      orderBy: Option[SqlOrderBy],
                      limit: Option[Int], ctx: Context = null) extends Node {
  def copyWithContext(c: Context) = copy(ctx = c)
  def sql =
    Seq(Some("select"),
        Some(projections.map(_.sql).mkString(", ")),
        relations.map(x => "from " + x.map(_.sql).mkString(", ")),
        filter.map(x => "where " + x.sql),
        groupBy.map(_.sql),
        orderBy.map(_.sql),
        limit.map(x => "limit " + x.toString)).flatten.mkString(" ")
}
```

```
Some(
  SelectStmt(
   List(Projection(StarProj(),None)),
   TableRelationAST(user,None),
   Some(And(Eq(FieldIdent(None,sex),Literal(male)),Ls(FieldIdent(None,age),Literal(100)))),
   Some(SqlGroupBy(List(FieldIdent(Some(user),name), FieldIdent(Some(user),sex)))),
   None,
   None
  )
)
select * from user group by user.name, user.sex where sex = 'male' and age<100
```

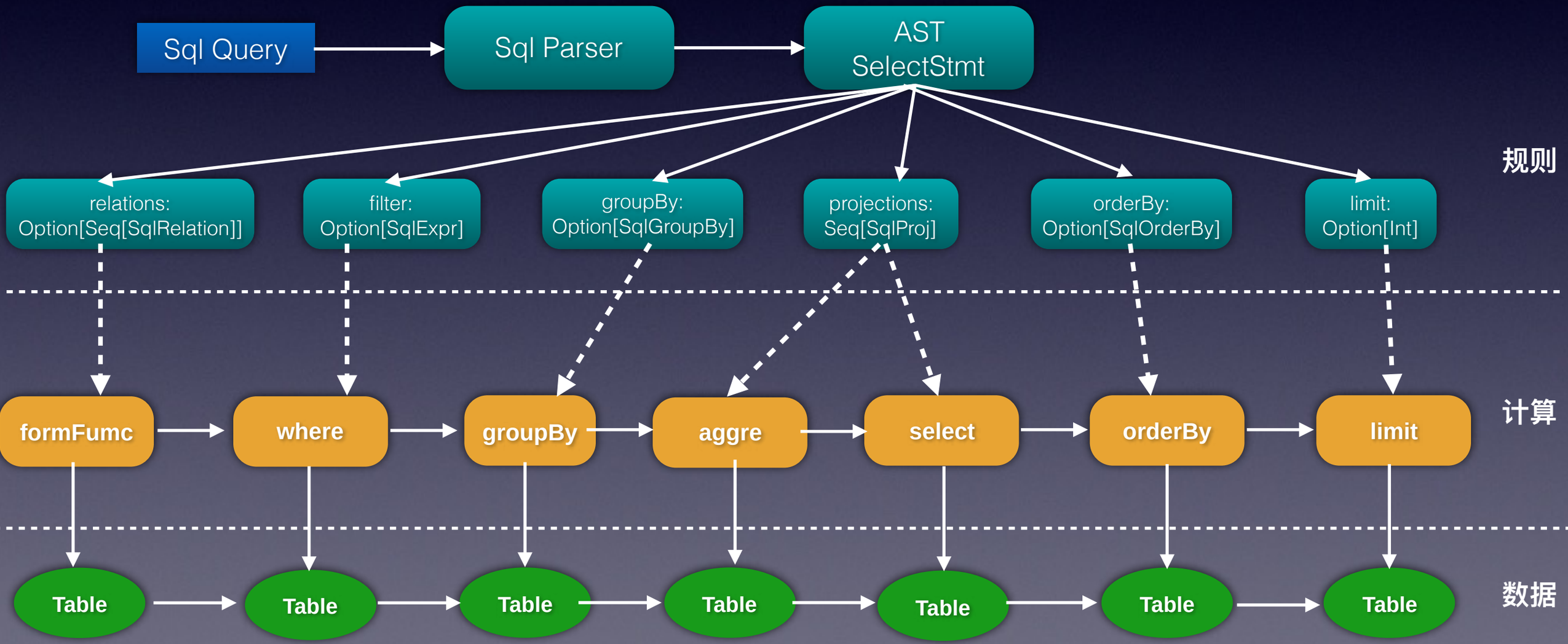
```
Some(
  SelectStmt(
   List(Projection(Max(FieldIdent(None,age)),Some(mmmm)), Projection(Min(FieldIdent(None,age)),Some(yyyy)), Projection(
   StarProj(),None)),
   TableRelationAST(user,None),
   None,
   Some(SqlGroupBy(List(FieldIdent(None,sex), FieldIdent(None,name)))),
   None,
   None
  )
)
select max(age) as mmmm,min(age) as yyyy,* from user group by sex,name
```

Butterfly

Butterfly 可分解为三个模块，即Sql Parser&AST, Table和Engine。

Sql Parser&AST :将Sql 语句转换为抽象语法树，简单的说就生成计算规则。
Engine :根据AST计算出Table，由7个串行的函数集所构成
Table :所运算的数据。主要分为Schema数据定义和数据类型转换(MeteData)

Schema模块 :数据集的定义，对Table中字段的定义，对字段中数据类型的定义。
Table模块 :其实就是集合，集合中的元素是Row。Row其实就是Map[String, MetaData[_]]。
LoadTableData 和 loadSchema： 从数据源获取数据。



谢谢

凡普金科正在积极招募极客，包括但不限于大数据工程师、算法工程师、java工程师，
在这里你会遇到一群同样优秀热血的极客，不断挑战自己的，获得惊呆同行的待遇！