**大数据实战项目**

**电商分析系统**

# 第一章 项目简介

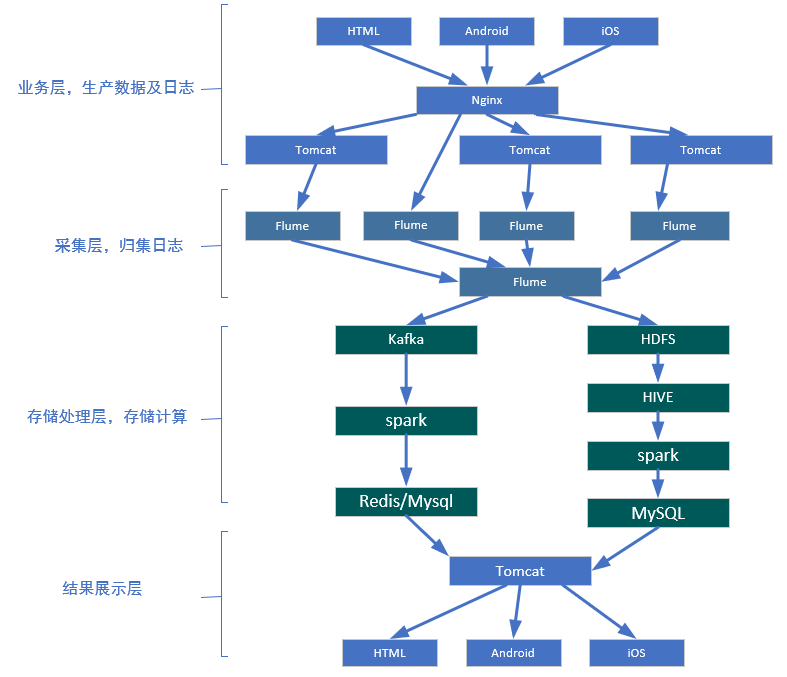
## 项目整体介绍

本项目来源于企业级电商网站的大数据统计分析平台，该平台以 Spark 框架为核心，对电商网站的日志进行离线和实时分析。

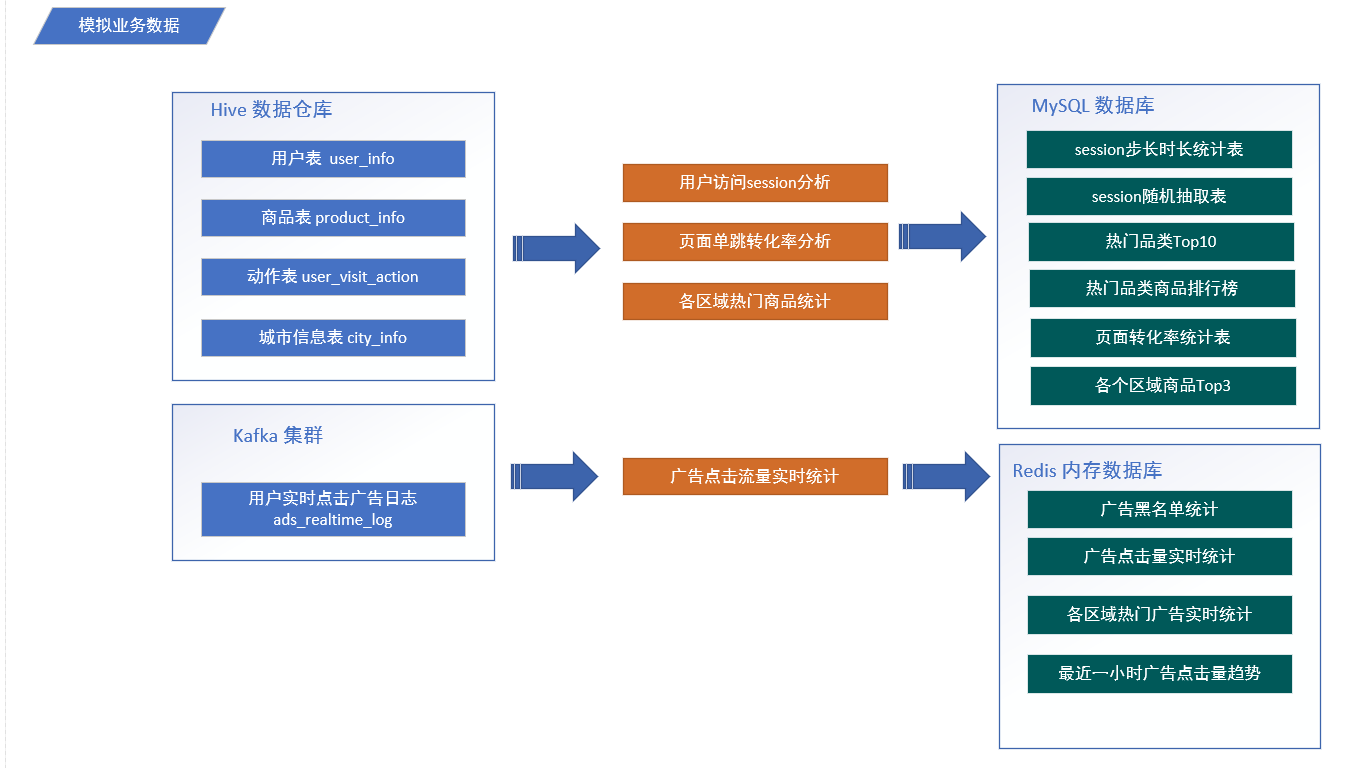
该大数据分析平台对电商网站的各种用户行为（访问行为、购物行为、广告点击行为等）进行分析，根据平台统计出来的数据，辅助公司中的 PM（产品经理）、数据分析师以及管理人员分析现有产品的情况，并根据用户行为分析结果持续改进产品的设计，以及调整公司的战略和业务。最终达到用大数据技术来帮助提升公司的业绩、营业额以及市场占有率的目标。

本项目使用了 Spark 技术生态栈中最常用的三个技术框架，Spark Core、Spark SQL 和 Spark Streaming，进行离线计算和实时计算业务模块的开发。实现了包括用户访问 session 分析、页面单跳转化率统计、热门商品离线统计、广告流量实时统计 4 个业务模块。通过合理的将实际业务模块进行技术整合与改造，该项目几乎完全涵盖了 Spark Core、Spark SQL 和 Spark Streaming 这三个技术框架中大部分的功能点、知识点，学员对于 Spark 技术框架的理解将会在本项目中得到很大的提高。

## 1.2 项目整体框架



## 1.3 业务需求部分



本项目分为离线分析系统与实时分析系统两大模块。

在离线分析系统中，我们将模拟业务数据写入 Hive 表中，离线分析系统从 Hive 中获取数据，并根据实际需求（用户访问 Session 分析、页面单跳转化率分析、各区域热门商品统计）对数据进行处理，最终将分析完毕的统计数据存储到 MySQL 的对应表格中。

在实时分析系统中，我们将模拟业务数据写入 Kafka 集群中，实时分析系统从 Kafka broker 中获取数据，通过 Spark Streaming 的流式处理对广告点击流量进行实时分析，最终将统计结果存储到 MySQL 的对应表格中。

## 1.4 数据源分析

### 1.4.1 user\_visit\_action

user\_visit\_action 表，存放网站或者 APP 每天的点击流数据。通俗地讲，就是用户对网站/APP 每点击一下，就会产生一条存放在这个表里面的数据。

user\_visit\_action 表中的字段解析如表所示：

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| date | 日期，代表这个用户点击行为是在哪一天发生的 |
| user\_id | 用户 ID，唯一标识某个用户 |
| session\_id | Session ID，唯一标识某个用户的一个访问 session |
| page\_id | 页面 ID，点击了某些商品/品类，也可能是搜索了某个关键词，然后进入了某个页面，页面的 id |
| action\_time | 动作时间，这个点击行为发生的时间点 |
| search\_keyword | 搜索关键词，如果用户执行的是一个搜索行为，比如说在网站/app 中，搜索了某个关键词，然后会跳转到商品列表页面； |
| click\_category\_id | 点击品类 ID，可能是在网站首页，点击了某个品类（美食、电子设备、电脑） |
| click\_product\_id | 点击商品 ID，可能是在网站首页，或者是在商品列表页，点击了某个商品（比如呷哺呷哺火锅 XX 路店 3 人套餐、iphone 6s） |
| order\_category\_ids | 下单品类 ID，代表了可能将某些商品加入了购物车，然后一次性对购物车中的商品下了一个订单，这就代表了某次下单的行为中，有哪些商品品类，可能有 6 个商品，但是就对应了 2 个品类，比如有 3 根火腿肠（食品品类），3 个电池（日用品品类） |
| order\_product\_ids | 下单商品 ID，某次下单，具体对哪些商品下的订单 |
| pay\_category\_ids | 付款品类 ID，对某个订单，或者某几个订单，进行了一次支付的行为，对应了哪些品类 |
| pay\_product\_ids | 付款商品 ID，支付行为下，对应的哪些具体的商品 |
| city\_id | 城市 ID，代表该用户行为发生在哪个城市 ,和城市信息表做关联 |

### 1.4.2 user\_info

user\_info 表，是一张普通的用户基本信息表；这张表中存放了网站/APP 所有注册用户的基本信息。

user\_i 表中的字段解析如表所示：

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| user\_id | 用户 ID，唯一标识某个用户 |
| username | 用户登录名 |
| name | 用户昵称或真实姓名 |
| age | 用户年龄 |
| professional | 用户职业 |
| gender | 用户性别 |

### 1.4.3 product\_info

product\_info 表，是一张普通的商品基本信息表；这张表中存放了网站/APP 所有商品的基本信息。

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| proudct\_id | 商品 ID，唯一标识某个商品 |
| product\_name | 商品名称 |
| extend\_info | 额外信息，例如商品为自营商品还是第三方商品 |

### 1.4.4 city\_info

|  |  |
| --- | --- |
| 字段名称 | 说明 |
| city\_id | 城市ID |
| city\_name | 城市名称 |
| area | 地区名称，如：华北，华东，西北 |

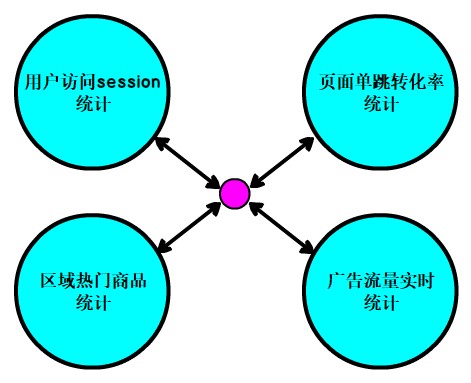
### 1.4.5 实时数据

程序每 5 秒向 Kafka 集群写入数据，格式如下：

格式 ：timestamp province city userid adid

|  |  |
| --- | --- |
| 字段名称 | 取值范围 |
| timestamp | 当前时间毫秒 类型：数字 |
| userId | 用户id，类型：数字 |
| area | 东北、华北、西南 类型：文本 |
| city | 城市中文名： 北京、上海、青岛等 类型：文本 |
| adid | 点击的广告编号，数字 |

## 1.5 项目需求



### 1.5.1 用户访问 session 统计

用户在电商网站上，通常会有很多的访问行为，通常都是进入首页，然后可能点击首页上的一些商品，点击首页上的一些品类，也可能随时在搜索框里面搜索关键词，还可能将一些商品加入购物车，对购物车中的多个商品下订单，最后对订单中的多个商品进行支付。

用户的每一次操作，其实可以理解为一个 action，在本项目中，我们关注点击、搜索、下单、支付这四个用户行为。

用户 session，是在电商平台的角度定义的会话概念，指的就是，从用户第一次进入首页，session 就开始了。然后在一定时间范围内，直到最后操作完（可能做了几十次、甚至上百次操作），离开网站，关闭浏览器，或者长时间没有做操作，那么 session 就结束了。

以上用户在网站内的访问过程，就称之为一次 session。简单理解，session 就是某一天某一个时间段内，某个用户对网站从打开/进入，到做了大量操作，到最后关闭浏览器。的过程，就叫做 session。

session 实际上就是一个电商网站中最基本的数据和大数据。那么面向消费者/ 用户端的大数据分析（C 端），最基本的就是面向用户访问行为/用户访问 session 的分析。

该模块主要是对用户访问 session 进行统计分析，包括 session 聚合指标计算（求平均每个session步长分布比例，访问时长分布比例）、按时间比例随机抽取 session、获取每天点击、下单和购买排名前 10 的品类、并获取 top10 品类中排名前 10 的 session。该模块可以让产品经理、数据分析师以及企业管理层形象地看到各种条件下的具体用户行为以及统计指标，从而对公司的产品设计以及业务发展战略做出调整。主要使用 Spark Core 实现。

### 1.5.2 页面单跳转化率统计

该模块主要是计算关键页面之间的单步跳转转化率，涉及到页面切片算法以及页面流匹配算法。该模块可以让产品经理、数据分析师以及企业管理层看到各个关键页面之间的转化率，从而对网页布局，进行更好的优化设计。主要使用 Spark Core 实现。

### 1.5.3 区域热门商品统计

该模块主要实现每天统计出各个区域的 top3 热门商品。该模块可以让企业管理层看到电商平台在不同区域出售的商品的整体情况，从而对公司的商品相关的战略进行调整。主要使用 Spark SQL 实现。

### 1.5.4 广告流量实时统计

网站 / app 中经常会给第三方平台做广告，这也是一些互联网公司的核心收入来源；当广告位招商完成后，广告会在网站 / app 的某个广告位发布出去，当用户访问网站 / app 的时候，会看到相应位置的广告，此时，有些用户可能就会去点击那个广告。

我们要获取用户点击广告的行为，并针对这一行为进行计算和统计。

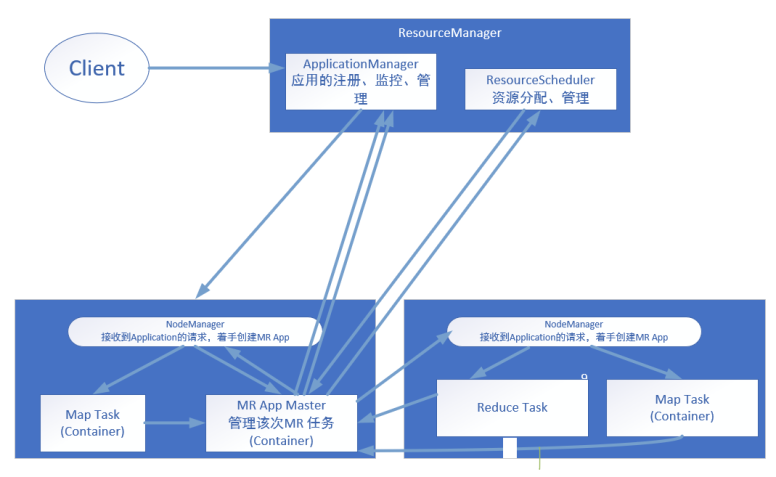
用户每次点击一个广告以后，会产生相应的埋点日志；在大数据实时统计系统中，会通过某些方式将数据写入到分布式消息队列中（Kafka）。

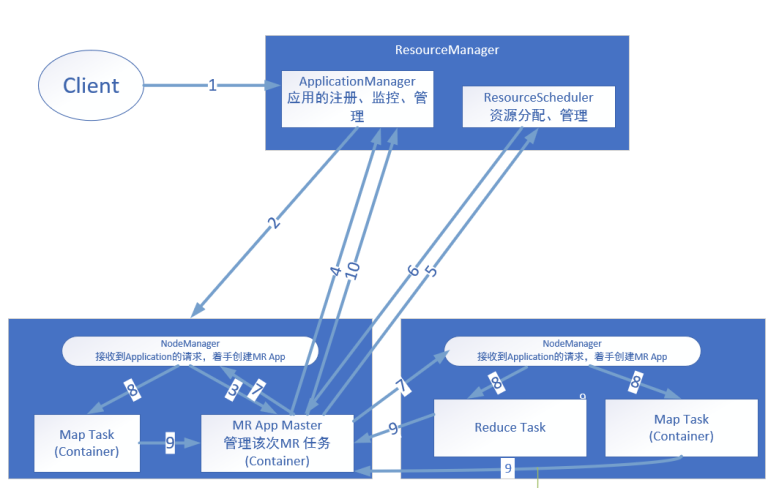
日志发送给后台 web 服务器（nginx），nginx 将日志数据负载均衡到多个 Tomcat 服务器上，Tomcat 服务器会不断将日志数据写入 Tomcat 日志文件中，写入后，就会被日志采集客户端（比如 flume agent）所采集，随后写入到消息队列中（kafka），我们的实时计算程序会从消息队列中（kafka）去实时地拉取数据，然后对数据进行实时的计算和统计。

这个模块的意义在于，让产品经理、高管可以实时地掌握到公司打的各种广告的投放效果。以便于后期持续地对公司的广告投放相关的战略和策略，进行调整和优化；以期望获得最好的广告收益。该模块负责实时统计公司的广告流量，包括广告展现流量和广告点击流量。实现动态黑名单机制，以及黑名单过滤；实现滑动窗口内的各城市的广告展现流量和广告点击流量的统计；实现每个区域每个广告的点击流量实时统计；实现每个区域 top3 点击量的广告的统计。主要使用 Spark Streaming 实现。

# 第二章预备知识

## 2.1 Spark





## 2.2 SparkCore

|  |  |
| --- | --- |
| 转换算子 |  |
| map | 转换集合中的每一个元素 |
| flatMap | 转换集合中的每一个元素，压平集合，把集合打碎，打碎成非集合元素 |
| filter | 过滤元素 |
| reduceByKey | 以key为单位进行聚合 |
| groupByKey | 以key为单位进行聚合（聚在一起，RDD[key,value]=>RDD[key,iterable[value]]） |
| left join | 左外连接，全集在左边 |
| join | 内关联，取左右交集 |
|  |  |

|  |  |
| --- | --- |
| 行动算子 |  |
| collect | 把RDD结果其提取到driver |
| take | 取前n个 |
| countByKey | 每个key的个数 Map[key,count] |
| foreach | 遍历RDD中每个元素 ，一般用于每个元素的输出 |
| foreachPartition | 按分区遍历 |

|  |  |
| --- | --- |
| 存储 |  |
| saveAsTextFile | 存储为文件 |
| toDF.write.saveAsTable | 存储到hive中 |
| toDF.write.format(jdbc).option.save | 存储到mysql |

## 2.3 Spark RDD 持久化

Spark 非常重要的一个功能特性就是可以将 RDD 持久化在内存中，当对 RDD 执行持久化操作时，每个节点都会将自己操作的 RDD 的 partition 持久化到内存中，并且在之后对该 RDD 的反复使用中，直接使用内存的 partition。这样的话，对于针对一个 RDD 反复执行多个操作的场景，就只要对 RDD 计算一次即可，后面直接使用该 RDD，而不需要反复计算多次该 RDD。

巧妙使用 RDD 持久化，甚至在某些场景下，可以将 Spark 应用程序的性能提高10 倍。对于迭代式算法和快速交互式应用来说，RDD 持久化是非常重要的。

例如，读取一个有着数十万行数据的 HDFS 文件，形成 linesRDD，这一读取过程会消耗大量时间，在 count 操作结束后，linesRDD 会被丢弃，会被后续的数据覆盖，当第二次再次使用 count 时，又需要重新读取 HDFS 文件数据，再次形成新的 linesRDD，这回导致反复消耗大量时间，会严重降低系统性能。

如果在读取完成后将 linesRDD 缓存起来，那么下一次执行 count 操作时将会直接使用缓存起来的 linesRDD，这会节省大量的时间。

要持久化一个 RDD，只要调用其 cache()或者 persist()方法即可。在该 RDD 第一次被计算出来时，就会直接缓存在每个节点中，而且 Spark 的持久化机制还是自动容错的，如果持久化的 RDD 的任何 partition 丢失了，那么 Spark 会自动通过其源

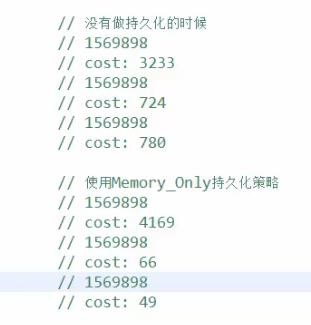
RDD，使用 transformation 操作重新计算该 partition。

cache()和 persist()的区别在于，cache()是 persist()的一种简化方式，cache()的底层就是调用的 persist()的无参版本，同时就是调用 persist(MEMORY\_ONLY)，将输入持久化到内存中。如果需要从内存中清除缓存，那么可以使用 unpersist()方法。

Spark 自己也会在 shuffle 操作时，进行数据的持久化，比如写入磁盘，主要是为了在节点失败时，避免需要重新计算整个过程。

|  |  |
| --- | --- |
| 持久化级别 | 含义 |
| MEMORY\_ONLY | 以非序列化的 Java 对象的方式持久化在 JVM 内存中。如果内存无法完全存储 RDD 所有的  partition，那么那些没有持久化的 partition 就会在下一次需要使用它们的时候，重新被计算 |
| MEMORY\_AND\_DISK | 同上，但是当某些 partition 无法存储在内存中时，会持久化到磁盘中。下次需要使用这些 partition |
|  | 时，需要从磁盘上读取 |
| MEMORY\_ONLY\_SER | 同 MEMORY\_ONLY，但是会使用 Java 序列化方式，将 Java 对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大  CPU 开销 |
| MEMORY\_AND\_DISK\_SER | 同 MEMORY\_AND\_DISK，但是使用序列化方式持久化 Java 对象 |
| DISK\_ONLY | 使用非序列化 Java 对象的方式持久化，完全存储到磁盘上 |
| MEMORY\_ONLY\_2  MEMORY\_AND\_DISK\_2  等等 | 如果是尾部加了 2 的持久化级别，表示将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可 |

以下为对一个 156 万行大小为 168MB 的文本文件进行处理，textFile 后只进行 count 操作，持久化与不持久化的结果如下：



## 2.4 Spark 共享变量

Spark 一个非常重要的特性就是共享变量。

默认情况下，如果在一个算子的函数中使用到了某个外部的变量，那么这个变量的值会被拷贝到每个 task 中，此时每个 task 只能操作自己的那份变量副本。如果多个 task 想要共享某个变量，那么这种方式是做不到的。

Spark 为此提供了两种共享变量，一种是 Broadcast Variable（广播变量），另一种是 Accumulator（累加变量）。Broadcast Variable 会将用到的变量，仅仅为每个Executor节点拷贝一份，更大的用途是优化性能，减少网络传输以及内存损耗。Accumulator 则可以让多个 task 共同操作一份变量，主要可以进行累加操作。Broadcast Variable 是共享读变量，task 不能去修改它，而 Accumulator 可以让多个 task 操作一个变量。

### 2.4.1 广播变量

广播变量允许程序员在每个机器上保留缓存的只读变量，而不是给每个任务发送一个副本。 例如，可以使用它们以有效的方式为每个节点提供一个大型输入数据集的副本。Spark 还尝试使用高效的广播算法分发广播变量，以降低通信成本。

Spark action 被划分为多个 Stages，被多个“shuffle”操作（宽依赖）所分割。Spark 自动广播每个阶段任务所需的公共数据（一个 Stage 中多个 task 使用的数据），以这种方式广播的数据以序列化形式缓存，并在运行每个任务之前反序列化。 这意味着，显式创建广播变量仅在跨多个阶段的任务需要相同数据或者以反序列化格式缓存数据很重要时才有用。

Spark 提供的 Broadcast Variable 是只读的，并且在每个节点上只会有一个副本，而不会为每个 task 都拷贝一份副本，因此，它的最大作用，就是减少变量到各个节点的网络传输消耗，以及在各个节点上的内存消耗。此外，Spark 内部也使用了高效的广播算法来减少网络消耗。

可以通过调用 SparkContext 的 broadcast()方法来针对每个变量创建广播变量。然后在算子的函数内，使用到广播变量时，每个Executor节点只会拷贝一份副本了，每个节点可以使用广播变量的 value()方法获取值。

### 2.4.2 累加器

累加器（accumulator）：Accumulator 是仅仅被相关操作累加的变量，因此可以在并行中被有效地支持。它们可用于实现计数器（如 MapReduce）或总和计数。

Accumulator 是存在于 Driver 端的，从Executor节点不断把值发到 Driver 端，在 Driver 端计数（Spark UI 在 SparkContext 创建时被创建，即在 Driver 端被创建，因此它可以读取 Accumulator 的数值），存在于 Driver 端的一个值，从节点是读取不到的。

Spark 提供的 Accumulator 主要用于多个节点对一个变量进行共享性的操作。 Accumulator 只提供了累加的功能，但是却给我们提供了多个 task 对于同一个变量并行操作的功能，但是 task 只能对 Accumulator 进行累加操作，不能读取它的值，只有 Driver 程序可以读取 Accumulator 的值。

自定义累加器类型的功能在 1.X 版本中就已经提供了，但是使用起来比较麻烦，在 2.0 版本后，累加器的易用性有了较大的改进，而且官方还提供了一个新的抽象类：AccumulatorV2 来提供更加友好的自定义类型累加器的实现方式。

官方同时给出了一个实现的示例：CollectionAccumulator 类，这个类允许以集合的形式收集 spark 应用执行过程中的一些信息。例如，我们可以用这个类收集 Spark 处理数据时的一些细节，当然，由于累加器的值最终要汇聚到 driver 端，为了避免 driver 端的 outofmemory 问题，需要对收集的信息的规模要加以控制，不宜过大。

|  |
| --- |
| **class** VisitSessionAccumulator **extends** AccumulatorV2[String, mutable.HashMap[String,Long]]{  **var** *aggrMap*=**new** mutable.HashMap[String,Long]()   *//判断是否为空* **override def** isZero: Boolean = {  *aggrMap*.isEmpty  }   *//复制累加器* **override def** copy(): AccumulatorV2[String, mutable.HashMap[String,Long]] = {  **val** accumulator = **new** VisitSessionAccumulator()  accumulator.*aggrMap*++=*aggrMap* accumulator  }   *//重置累加器* **override def** reset(): Unit = {  *aggrMap*=**new** mutable.HashMap[String,Long]()  }   *//向累加器添加一个值* **override def** add(key: String): Unit ={  **val** newCount = **this**.*aggrMap*.getOrElse(key,0L)+1L  **this**.*aggrMap*(key)=newCount  }   *// 两个分区的数据合并* **override def** merge(other: AccumulatorV2[String, mutable.HashMap[String,Long]]): Unit = {  other **match** {  **case** visitSessionAcc: VisitSessionAccumulator=> visitSessionAcc.*aggrMap*.foldLeft(**this**.*aggrMap*){**case** (aggrMap,(key,count))=>  aggrMap(key)=aggrMap.getOrElse(key,0L)+count  aggrMap  }  }   }   *//获取累加器中的值* **override def** value: mutable.HashMap[String,Long] = {  **this**.*aggrMap* } } |

## 2.5 SparkSQL

### 2.5.1 RDD、DataFrame 与 DataSet

#### 2.5.1.1 RDD

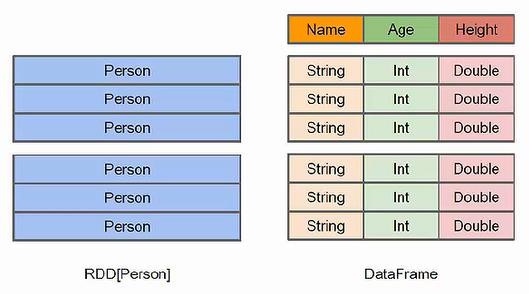
RDD，全称为 Resilient Distributed Datasets，即分布式数据集，是 Spark 中最基本的数据抽象，它代表一个不可变、 可分区、里面的元素可以并行计算的集合。在 Spark 中，对数据的所有操作不外乎创建 RDD、转化已有 RDD 以及调用 RDD 操作进行求值。每个 RDD 都被分为多个分区，这些分区运行在集群中的不同的节点上。

RDD 可以包含 Python、Java、Scala 中任意类型的对象，甚至可以包含用户自定义的对象。RDD 具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD 允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升查询速度。

RDD 支持两种操作：transformation 操作和 action 操作。RDD 的 transformation 操作是返回一个新的 RDD 的操作，比如 map 和 filter()，而 action 操作则是向驱动器程序返回结果或者把结果写入外部系统的操作，比如 count()和 first()。

#### 2.5.1.2 DataFrame

DataFrame 是一个分布式数据容器。相比于 RDD，DataFrame 更像传统数据库中的二维表格，除了数据之外，还记录数据的结构信息，即 schema。同时，与 Hive 类似，DataFrame 也支持嵌套数据类型（struct，array 和 map）。从 API 易用性的角度上看，DataFrame API 提供的是一套高层的关系操作，比函数式的 RDD API 要更加友好，门槛更低。由于与 R 和 Pandas 中的 DataFrame 类似，Spark DataFrame 很好地继承了传统单机数据分析的开和体验。



RDD 与 DataFrame 对比

如上图所示，左侧的 RDD[Person]虽然以 Person 为类型参数，但是 Spark 框架本身不了解 Person 类的内部结构。而右侧的 DataFrame 却提供了详细的结构信息，使得 SparkSQL 可以清楚地知道该数据集中包含那些列，每列的名称是什么。 DataFrame 多了数据的结构信息，即 schema。RDD 是分布式的 Java 对象的集合。

DataFrame 是分布式的 Row 对象的集合。DataFrame 处理提供了比 RDD 更为丰富的算子以外，更重要的是提升了执行效率、减少数据读取以及执行计划的优化，比如 filter 下推、裁剪等。

DataFrame 是 DataSet 的特例，DataFrame = DataSet[Row]，所以可以通过 as 方法将 DataFrame 转换为 DataSet。Row 是一个类型，跟 Car、Person 这些类型一样，所有的表结构信息都用 Row 来表示。

#### 2.5.1.3 DataSet

DataSet 是 DataFrame API 的一个拓展，是 Spark 最新的数据抽象。DataSet 具有用户友好的 API 风格，既具有类型安全检查也具有 DataFrame 的查询优化特性。 DataSet 支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。

样例类被用来在 DataSet 中定义数据的结构信息，样例类中每个属性的名称直接映射到 DataSet 中的字段名称。

DataSet 是强类型的。比如可以有 DataSet[Car]，DataSet[Person]

DataFrame 只知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没有办法在编译的时候检查是否类型失败的，比如你可以对一个 String 类型进行加减法操作，在执行的时候才会报错，而 DataSet 不仅仅知道字段，而且知道字段类型，所以有更为严格的错误检查。就更 JSON 对象和类对象之间的类比。

### 2.5.2 RDD 、DataFrame和DataSet互操作

#### 2.5.2.1 DataFrame/DataSet 转 RDD

#### 2.5.2.2 RDD 转 DataFrame

#### 2.5.2.3 DataFrame 转 DataSet

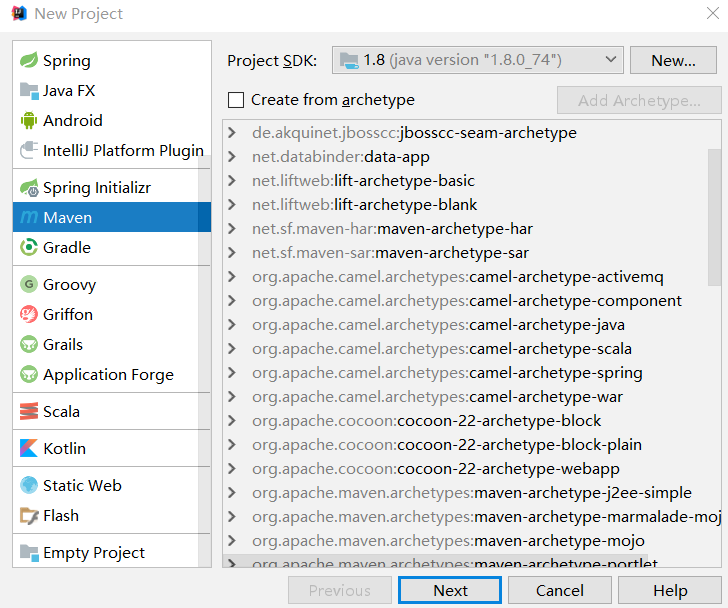
#### 2.5.2.4 RDD 转 DataSet

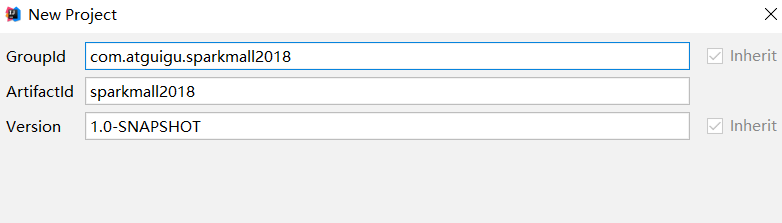
#### 2.5.2.5 DataSet 转 DataFrame

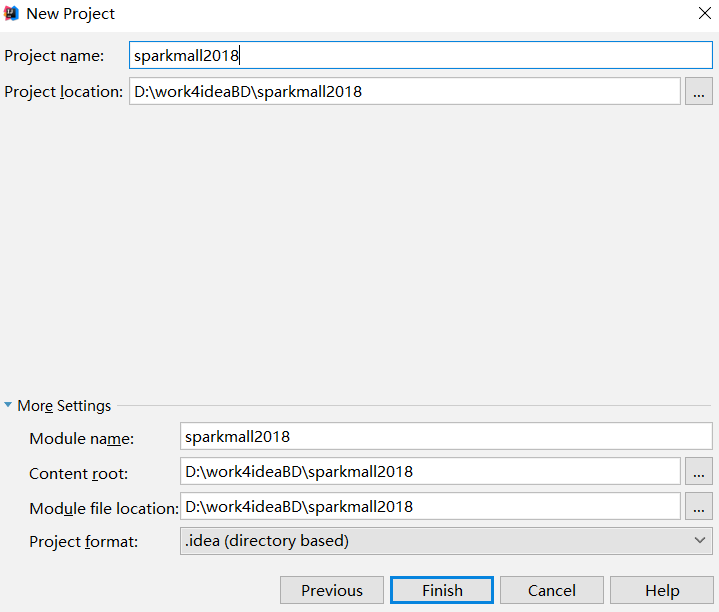
# 第三章 框架的搭建

## 3.1 建立项目父工程

idea中 右键点击New project







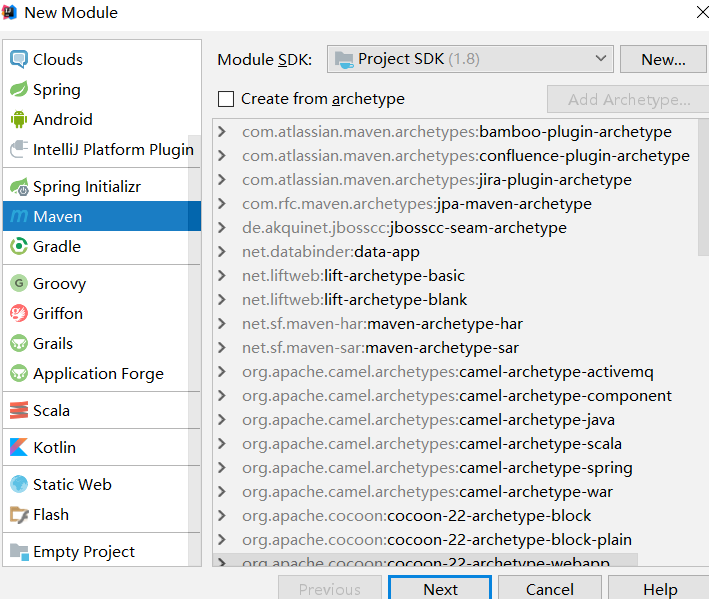
#### pom.xml

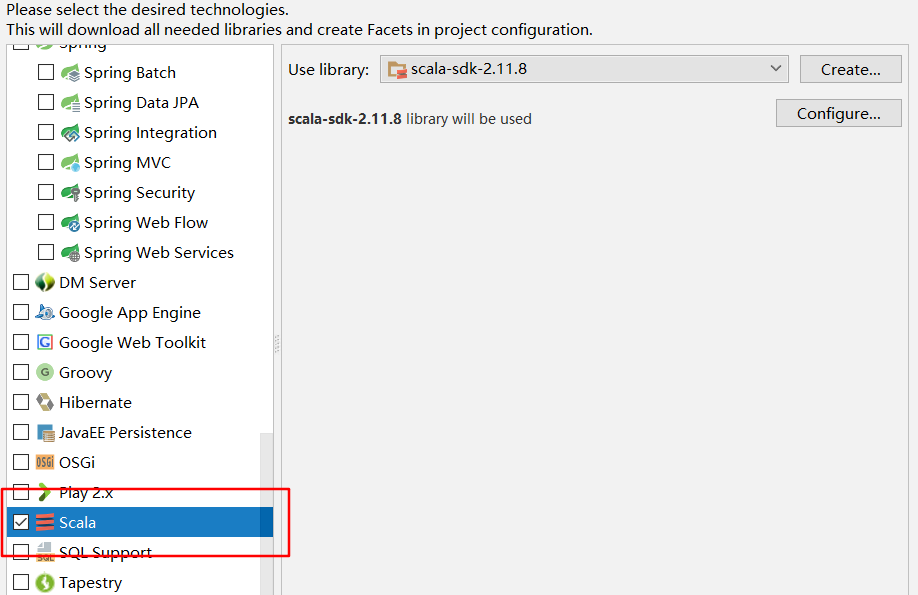
这个pom作为其他模块的parent来被继承

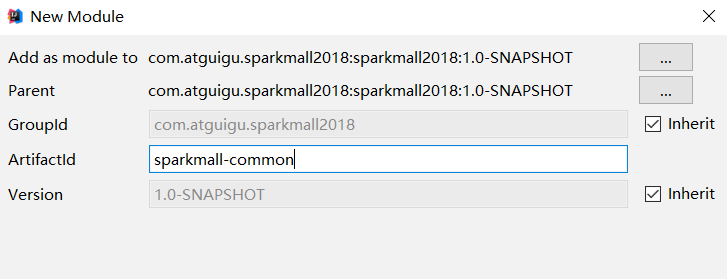
|  |
| --- |
| *<?*xml version="1.0" encoding="UTF-8"*?>* <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>   <groupId>com.atguigu.sparkmall2018</groupId>  <artifactId>sparkmall2018</artifactId>  <packaging>pom</packaging>  <version>1.0-SNAPSHOT</version>  <modules>   </modules>  <properties>  <spark.version>2.1.1</spark.version>  <scala.version>2.11.8</scala.version>  <log4j.version>1.2.17</log4j.version>  <slf4j.version>1.7.22</slf4j.version>  </properties>  <dependencies>  *<!--此处放日志包，所有项目都要引用-->  <!-- 所有子项目的日志框架 -->* <dependency>  <groupId>org.slf4j</groupId>  <artifactId>jcl-over-slf4j</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>  <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  *<!-- 具体的日志实现 -->* <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>  </dependencies>  <dependencyManagement>  *<!--此处声明工具依赖，各个模块，选择使用-->* <dependencies>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-core\_2.11</artifactId>  <version>${spark.version}</version>  *<!-- provider如果存在，那么运行时该Jar包不存在，也不会打包到最终的发布版本中，只是编译器有效 -->  <!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-sql\_2.11</artifactId>  <version>${spark.version}</version>  *<!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-streaming\_2.11</artifactId>  <version>${spark.version}</version>  *<!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-mllib\_2.11</artifactId>  <version>${spark.version}</version>  *<!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-graphx\_2.11</artifactId>  <version>${spark.version}</version>  *<!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.scala-lang</groupId>  <artifactId>scala-library</artifactId>  <version>${scala.version}</version>  *<!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-hive\_2.11</artifactId>  <version>${spark.version}</version>  *<!--<scope>provided</scope>-->* </dependency>  <dependency>  <groupId>org.apache.spark</groupId>  <artifactId>spark-streaming-kafka-0-10\_2.11</artifactId>  <version>${spark.version}</version>  </dependency>   </dependencies>    </dependencyManagement>   <build>  *<!-- 声明并引入子项目共有的插件【插件就是附着到Maven各个声明周期的具体实现】 -->* <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.6.1</version>  *<!-- 所有的编译都依照JDK1.8来搞 -->* <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  </plugins>   *<!-- 仅声明子项目共有的插件，如果子项目需要此插件，那么子项目需要声明 -->* <pluginManagement>  <plugins>  *<!-- 该插件用于将Scala代码编译成class文件 -->* <plugin>  <groupId>net.alchim31.maven</groupId>  <artifactId>scala-maven-plugin</artifactId>  <version>3.2.2</version>  <executions>  <execution>  *<!-- 声明绑定到maven的compile阶段 -->* <goals>  <goal>compile</goal>  <goal>testCompile</goal>  </goals>  </execution>  </executions>  </plugin>   *<!-- 用于项目的打包插件 -->* <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-assembly-plugin</artifactId>  <version>3.0.0</version>  <executions>  <execution>  <id>make-assembly</id>  <phase>package</phase>  <goals>  <goal>single</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </pluginManagement>  </build>  </project> |

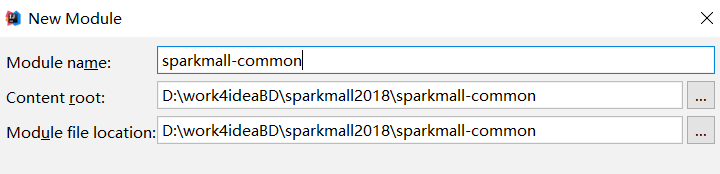
## 3.2 建立项目公共模块

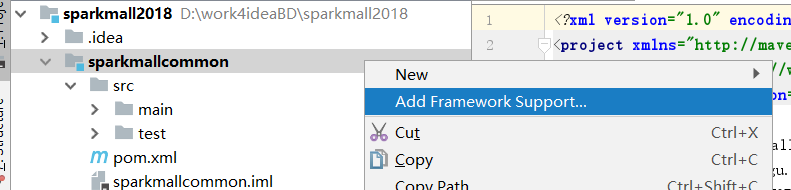
公共模块主要是放公共的依赖包和配置文件，以及用户自定义的一些工具类。

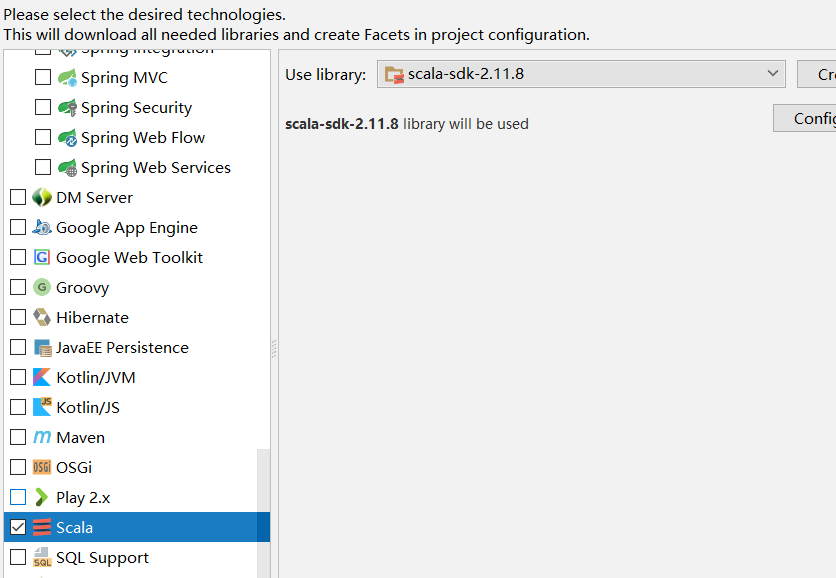












pom.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**parent**>  <**artifactId**>sparkmall2018</**artifactId**>  <**groupId**>com.atguigu.sparkmall2018</**groupId**>  <**version**>1.0-SNAPSHOT</**version**>  <**relativePath**>../../sparkmall2018</**relativePath**>  </**parent**>  <**modelVersion**>4.0.0</**modelVersion**>   <**artifactId**>sparkmall-common</**artifactId**>  <**dependencies**>  *<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.47</**version**>  </**dependency**>    <**dependency**>  *<!-- common-pool2使用的是面向接口的编程，它为我们提供的是一个抽象的对象池管理方式，  根据我们业务的不同，我们需要重写或实现一些方法和接口 -->* <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-pool2</**artifactId**>  <**version**>2.4.2</**version**>  </**dependency**>   <**dependency**>  *<!-- 有助于以各种格式读取配置/偏好文件的工具 -->* <**groupId**>org.apache.commons</**groupId**>  <**artifactId**>commons-configuration2</**artifactId**>  <**version**>2.2</**version**>  </**dependency**>  <**dependency**>  *<!-- commons-beanutil中包括大量和JavaBean操作有关的工具方法，  使用它能够轻松利用Java反射机制来完毕代码中所须要的功能，而不须要具体研究反射的原理和使用 -->* <**groupId**>commons-beanutils</**groupId**>  <**artifactId**>commons-beanutils</**artifactId**>  <**version**>1.9.3</**version**>  </**dependency**>   <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.47</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.kafka</**groupId**>  <**artifactId**>kafka-clients</**artifactId**>  <**version**>0.10.2.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-streaming\_2.11</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-streaming-kafka-0-10\_2.11</**artifactId**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->* <**dependency**>  <**groupId**>redis.clients</**groupId**>  <**artifactId**>jedis</**artifactId**>  <**version**>2.9.0</**version**>  </**dependency**>   *<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.10</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/net.sf.json-lib/json-lib -->* <**dependency**>  <**groupId**>net.sf.json-lib</**groupId**>  <**artifactId**>json-lib</**artifactId**>  <**version**>2.4</**version**>  <**classifier**>jdk15</**classifier**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.json4s/json4s-native -->* <**dependency**>  <**groupId**>org.json4s</**groupId**>  <**artifactId**>json4s-native\_2.11</**artifactId**>  <**version**>3.2.11</**version**>  </**dependency**>   *<!-- https://mvnrepository.com/artifact/org.json4s/json4s-jackson -->* <**dependency**>  <**groupId**>org.json4s</**groupId**>  <**artifactId**>json4s-jackson\_2.11</**artifactId**>  <**version**>3.2.11</**version**>  </**dependency**>   </**dependencies**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**> </**project**> |

### 增加配置文件

在resources 中添加配置文件 ，

#### log4j.properties

|  |
| --- |
| **log4j.rootLogger**=**error, stdout,R log4j.appender.stdout**=**org.apache.log4j.ConsoleAppender log4j.appender.stdout.layout**=**org.apache.log4j.PatternLayout log4j.appender.stdout.layout.ConversionPattern**=**%d{yyyy-MM-dd HH:mm:ss,SSS} %5p --- [%50t] %-80c(line:%5L) : %m%n  log4j.appender.R**=**org.apache.log4j.RollingFileAppender log4j.appender.R.File**=**../log/agent.log log4j.appender.R.MaxFileSize**=**1024KB log4j.appender.R.MaxBackupIndex**=**1  log4j.appender.R.layout**=**org.apache.log4j.PatternLayout log4j.appender.R.layout.ConversionPattern**=**%d{yyyy-MM-dd HH:mm:ss,SSS} %5p --- [%50t] %-80c(line:%6L) : %m%n** |

#### config.properties

|  |
| --- |
| *# jbdc配置* **jdbc.datasource.size**=**10**  **jdbc.driver.class=com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://localhost:3306/sparkmall?useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true jdbc.user**=**root jdbc.password**=**000000** *# Kafka配置* **kafka.broker.list**=**hadoop1:9092,hadoop2:9092,hadoop3:9092** *# Redis配置* **redis.host**=**hadoop1 redis.port**=**6379**  *# hive 的数据库名（选配）* **hive.database**=**sparkmall0808** |

#### condition.properties 存放于resources目录下

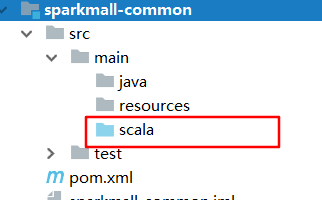
|  |
| --- |
| *# 可以使用的属性如下： # startDate： 格式： yyyy-MM-DD [必选] # endDate: 格式： yyyy-MM-DD [必选] # startAge: 范围： 0 - 59 # endAge: 范围： 0 - 59 # professionals： 范围：professionals[0 - 59] # cities: 城市id # gender: 范围： 0 - 1 # keywords: 范围： ("火锅", "蛋糕", "重庆辣子鸡", "重庆小面", "呷哺呷哺", "新辣道鱼火锅", "国贸大厦", "太古商场", "日本料理", "温泉") # categoryIds：0 - 99，以逗号分隔 # targetPageFlow： 0 - 99， 以逗号分隔* **condition.params.json**=**{startDate:"2018-11-01", \  endDate:"2018-12-28", \  startAge: "20", \  endAge: "50", \  professionals: "", \  city: "", \  gender:"", \  keywords:"", \  categoryIds:"", \  targetPageFlow:"1,2,3,4,5,6,7"}** |

hive-site.xml

用于idea连接虚拟机中的hive ，也可以不配置，那么spark在idea运行时，会自动在windows下建立仓库

|  |
| --- |
| *<?***xml version="1.0"***?> <?***xml-stylesheet type="text/xsl" href="configuration.xsl"***?>* <**configuration**>  <**property**>  <**name**>javax.jdo.option.ConnectionURL</**name**>  <**value**>jdbc:mysql://hadoop1:3306/metastore?createDatabaseIfNotExist=true</**value**>  <**description**>JDBC connect string for a JDBC metastore</**description**>  </**property**>   <**property**>  <**name**>javax.jdo.option.ConnectionDriverName</**name**>  <**value**>com.mysql.jdbc.Driver</**value**>  <**description**>Driver class name for a JDBC metastore</**description**>  </**property**>   <**property**>  <**name**>javax.jdo.option.ConnectionUserName</**name**>  <**value**>root</**value**>  <**description**>username to use against metastore database</**description**>  </**property**>   <**property**>  <**name**>javax.jdo.option.ConnectionPassword</**name**>  <**value**>123123</**value**>  <**description**>password to use against metastore database</**description**>  </**property**>  <**property**>  <**name**>hive.cli.print.current.db</**name**>  <**value**>true</**value**>  <**description**>Whether to include the current database in the Hive prompt.</**description**>  </**property**>  <**property**>  <**name**>hive.cli.print.header</**name**>  <**value**>false</**value**>  <**description**>Whether to print the names of the columns in query output.</**description**>  </**property**>  <**property**>  <**name**>hive.cli.print.header</**name**>  <**value**>true</**value**> </**property**>  <**property**>  <**name**>hive.cli.print.current.db</**name**>  <**value**>true</**value**> </**property**>   </**configuration**> |

### 建立scala 源码目录



在com.atguigu.sparkmall2018.common.util下增加

#### ConfigurationUtil

用来读取配置文件的工具

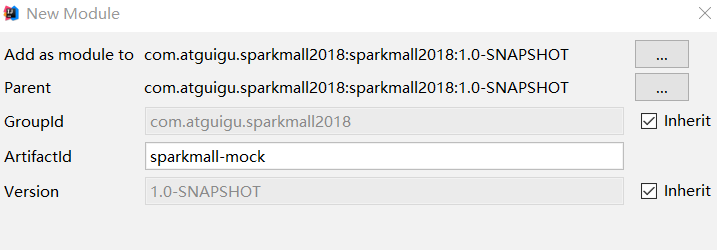
|  |
| --- |
| **import** org.apache.commons.configuration2.{FileBasedConfiguration, PropertiesConfiguration} **import** org.apache.commons.configuration2.builder.FileBasedConfigurationBuilder **import** org.apache.commons.configuration2.builder.fluent.Parameters  **object** ConfigurationUtil {   *// FileBasedConfigurationBuilder:产生一个传入的类的实例对象  // FileBasedConfiguration:融合FileBased与Configuration的接口  // PropertiesConfiguration:从一个或者多个文件读取配置的标准配置加载器  // configure():通过params实例初始化配置生成器  // 向FileBasedConfigurationBuilder()中传入一个标准配置加载器类，生成一个加载器类的实例对象，然后通过params参数对其初始化* **def** apply(propertiesName:String) = {  **val** configurationUtil = **new** ConfigurationUtil()  **if** (configurationUtil.*config* == **null**) {  configurationUtil.*config* = **new** FileBasedConfigurationBuilder[FileBasedConfiguration](*classOf*[PropertiesConfiguration])  .configure(**new** Parameters().properties().setFileName(propertiesName)).getConfiguration  }  configurationUtil  }  }  **class** ConfigurationUtil(){  **var** *config*:FileBasedConfiguration=**null** } |

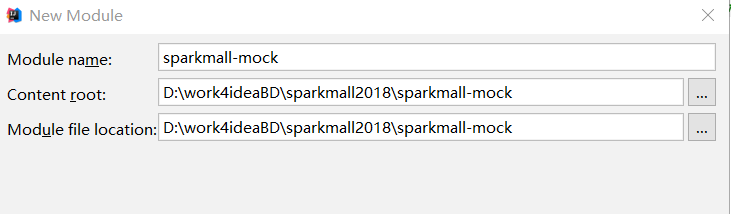
#### DataModel类

model 包下 增加 DataModel

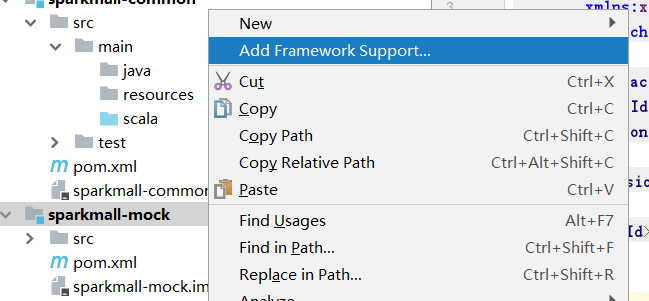
|  |
| --- |
| *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 输入表 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  /\*\*  \* 用户访问动作表  \*  \** @param date *用户点击行为的日期  \** @param user\_id *用户的ID  \** @param session\_id *Session的ID  \** @param page\_id *某个页面的ID  \** @param action\_time *点击行为的时间点  \** @param search\_keyword *用户搜索的关键词  \** @param click\_category\_id *某一个商品品类的ID  \** @param click\_product\_id *某一个商品的ID  \** @param order\_category\_ids *一次订单中所有品类的ID集合  \** @param order\_product\_ids *一次订单中所有商品的ID集合  \** @param pay\_category\_ids *一次支付中所有品类的ID集合  \** @param pay\_product\_ids *一次支付中所有商品的ID集合  \*/* case class UserVisitAction(date: String,  user\_id: Long,  session\_id: String,  page\_id: Long,  action\_time: String,  search\_keyword: String,  click\_category\_id: Long,  click\_product\_id: Long,  order\_category\_ids: String,  order\_product\_ids: String,  pay\_category\_ids: String,  pay\_product\_ids: String,  city\_id:Long  )  */\*\*  \* 用户信息表  \*  \** @param user\_id *用户的ID  \** @param username *用户的名称  \** @param name *用户的名字  \** @param age *用户的年龄  \** @param professional *用户的职业  \** @param gender *用户的性别  \*/* case class UserInfo(user\_id: Long,  username: String,  name: String,  age: Int,  professional: String,  gender: String  )  */\*\*  \* 产品表  \*  \** @param product\_id *商品的ID  \** @param product\_name *商品的名称  \** @param extend\_info *商品额外的信息  \*/* case class ProductInfo(product\_id: Long,  product\_name: String,  extend\_info: String  )  case class CityInfo (city\_id:Long,  city\_name:String,  area:String  ){  } |

## 3.3 建立模拟数据模块





增加scala模块



### pom.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  <**parent**>  <**artifactId**>sparkmall2018</**artifactId**>  <**groupId**>com.atguigu.sparkmall2018</**groupId**>  <**version**>1.0-SNAPSHOT</**version**>  </**parent**>  <**modelVersion**>4.0.0</**modelVersion**>   <**artifactId**>sparkmall-mock</**artifactId**>   <**dependencies**>  <**dependency**>  <**groupId**>com.atguigu.sparkmall2018</**groupId**>  <**artifactId**>sparkmall-common</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-core\_2.11</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.11</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-hive\_2.11</**artifactId**>  </**dependency**>    </**dependencies**>  <**build**>  <**plugins**>  <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  </**plugin**>  </**plugins**>  </**build**> </**project**> |

### 增加生成数据用的随机工具类

随机生成某个时间区间内的日期

#### RandomDate

|  |
| --- |
| **object** RandomDate {   **def** apply(startDate:Date,endDate:Date,step:Int): RandomDate ={  **val** randomDate = **new** RandomDate()  **val** avgStepTime = (endDate.getTime- startDate.getTime)/step  randomDate.*maxTimeStep*=avgStepTime\*2  randomDate.*lastDateTime*=startDate.getTime  randomDate  }  **class** RandomDate{  **var** *lastDateTime* =0L  **var** *maxTimeStep*=0L   **def** getRandomDate()={  **val** timeStep = **new** Random().nextInt(*maxTimeStep*.toInt)  *lastDateTime* = *lastDateTime*+timeStep   **new** Date( *lastDateTime*)  }  }  } |
|  |

#### RandomNum

随机生成区间范围内的数字（左右包含）

|  |
| --- |
| **object** RandomNum {   **def** apply(fromNum:Int,toNum:Int): Int = {  fromNum+ **new** Random().nextInt(toNum-fromNum+1)  }  **def** multi(fromNum:Int,toNum:Int,amount:Int,delimiter:String,canRepeat:Boolean) ={**"1,2,3"**   **// 实现方法 在fromNum和 toNum之间的 多个数组拼接的字符串 共amount个**  **用delimiter分割 canRepeat为false则不允许重复**  **此处逻辑请同学们自行实现**  }  } |

#### RandomOptions

根据权重比例随机生成选项

|  |
| --- |
| **object** RandomOptions {   **def** apply[T](opts:RanOpt[T]\*): RandomOptions[T] ={  **val** randomOptions= **new** RandomOptions[T]()  **for** (opt <- opts ) {  randomOptions.*totalWeight*+=opt.weight  **for** ( i <- 1 to opt.weight ) {  randomOptions.*optsBuffer*+=opt.value  }  }  randomOptions  }    **def** main(args: Array[String]): Unit = {  **val** randomName = *RandomOptions*(*RanOpt*(**"zhangchen"**,10),*RanOpt*(**"li4"**,30))  **for** (i <- 1 to 40 ) {  *println*(i+**":"**+randomName.getRandomOpt())   }  }   }   **case class** RanOpt[T](value:T,weight:Int){ } **class** RandomOptions[T](opts:RanOpt[T]\*) {  **var** *totalWeight*=0  **var** *optsBuffer* =**new** ListBuffer[T]   **def** getRandomOpt(): T ={  **val** randomNum= **new** Random().nextInt(*totalWeight*)  *optsBuffer*(randomNum)  } } |

### 离线数据的生成类 MockerOffline

|  |
| --- |
| **import** java.text.SimpleDateFormat **import** java.util.UUID  **import** com.atguigu.sparkmall2018.common.model.{CityInfo, ProductInfo, UserInfo, UserVisitAction} **import** org.apache.spark.SparkConf **import** org.apache.spark.sql.SparkSession **import** com.atguigu.sparkmall2018.mock.util.{RanOpt, RandomDate, RandomNum, RandomOptions}  **import** scala.collection.mutable.ListBuffer **import** org.apache.spark.sql.\_    **object** MockerOffline  {   **val** *userNum*=100;  **val** *productNum*=100  **val** *sessionNum*=10000   **val** *pageNum*=50  **val** *categoryNum*=20    **val** *logAboutNum*=100000 *//日志大致数量，用于分布时间* **val** *professionRandomOpt* = *RandomOptions*(*RanOpt*(**"学生"**,40),*RanOpt*(**"程序员"**,30),*RanOpt*(**"经理"**,20),*RanOpt*(**"老师"**,10))   **val** *genderRandomOpt*=*RandomOptions*(*RanOpt*(**"男"**,60),*RanOpt*(**"女"**,40) )  **val** *ageFrom*=10  **val** *ageTo*=59   **val** *productExRandomOpt*=*RandomOptions*(*RanOpt*(**"自营"**,70),*RanOpt*(**"第三方"**,30) )   **val** *searchKeywordsOptions* = *RandomOptions*(*RanOpt*(**"手机"**,30),*RanOpt*(**"笔记本"**,70),*RanOpt*(**"内存"**,70),*RanOpt*(**"i7"**,70),*RanOpt*(**"苹果"**,70),*RanOpt*(**"吃鸡"**,70))  **val** *actionsOptions* = *RandomOptions*(*RanOpt*(**"search"**,20), *RanOpt*(**"click"**,60), *RanOpt*(**"order"**,6),*RanOpt*( **"pay"**,4),*RanOpt*(**"quit"**,10))    **def** main(args: Array[String]): Unit = {   **val** sparkConf = **new** SparkConf().setAppName(**"Mock"**).setMaster(**"local[\*]"**)  **val** sparkSession = SparkSession.*builder*().config(sparkConf).enableHiveSupport().getOrCreate()   *// 模拟数据* **val** userVisitActionData = **this**.*mockUserAction*()  **val** userInfoData = **this**.*mockUserInfo*()  **val** productInfoData = **this**.*mockProductInfo*()  **val** cityInfoData=**this**.*mockCityInfo*()   *// 将模拟数据装换为RDD* **val** userVisitActionRdd = sparkSession.sparkContext.makeRDD(userVisitActionData)  **val** userInfoRdd = sparkSession.sparkContext.makeRDD(userInfoData)  **val** productInfoRdd = sparkSession.sparkContext.makeRDD(productInfoData)  **val** cityInfoRdd = sparkSession.sparkContext.makeRDD(cityInfoData)   **import** sparkSession.implicits.\_  **val** userVisitActionDF = userVisitActionRdd.toDF()  **val** userInfoDF = userInfoRdd.toDF()  **val** productInfoDF= productInfoRdd.toDF()  **val** cityInfoDF= cityInfoRdd.toDF()    *insertHive*(sparkSession,**"user\_visit\_action"**,userVisitActionDF)  *insertHive*(sparkSession,**"user\_info"**,userInfoDF)  *insertHive*(sparkSession,**"product\_info"**,productInfoDF)  *insertHive*(sparkSession,**"city\_info"**,cityInfoDF)   sparkSession.close()  }   **def** insertHive(sparkSession: SparkSession,tableName:String,dataFrame:DataFrame): Unit ={  sparkSession.sql(use数据库)  sparkSession.sql(**"drop table if exists "**+tableName)  dataFrame.write.saveAsTable(tableName)  *println*(**"保存："**+tableName+**"完成"**)  sparkSession.sql(**"select \* from "**+tableName).show(100)   }    **def** mockUserInfo()={    **val** rows=**new** ListBuffer[UserInfo]()   **for** (i <- 1 to *userNum* ) {  **val** user = *UserInfo*(i,  **"user\_"** + i,  **"name\_"** + i,  *RandomNum*(*ageFrom*, *ageTo*), *//年龄  professionRandomOpt*.getRandomOpt(),  *genderRandomOpt*.getRandomOpt()  )  rows+=user  }  rows.toList  }   **def** mockUserAction()={    **val** rows=**new** ListBuffer[UserVisitAction]()   **val** startDate = **new** SimpleDateFormat(**"yyyy-MM-dd"**).parse(**"2018-11-26"**)  **val** endDate = **new** SimpleDateFormat(**"yyyy-MM-dd"**).parse(**"2018-11-27"**)  **val** randomDate = *RandomDate*(startDate,endDate,*logAboutNum*)  **for** (i <- 1 to *sessionNum* ) {  **val** userId= *RandomNum*(1,*userNum*)  **val** sessionId = UUID.*randomUUID*().toString  **var** isQuit=**false   while**(!isQuit){  **val** action = *actionsOptions*.getRandomOpt()   **if**(action==**"quit"**){  isQuit=**true** }**else**{  **val** actionDateTime = randomDate.getRandomDate()  **val** actionDateString = **new** SimpleDateFormat(**"yyyy-MM-dd"**).format(actionDateTime)  **val** actionDateTimeString = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**).format(actionDateTime)   **var** searchKeyword: String = **null  var** clickCategoryId: Long = -1L  **var** clickProductId: Long = -1L  **var** orderCategoryIds: String = **null  var** orderProductIds: String = **null  var** payCategoryIds: String = **null  var** payProductIds: String = **null   var** cityId:Long= *RandomNum*(1, 26).toLong   action **match** {  **case "search"** => searchKeyword=*searchKeywordsOptions*.getRandomOpt()  **case "click"** =>clickCategoryId=*RandomNum*(1,*cargoryNum*)  clickProductId= *RandomNum*(1,*productNum*)  **case "order"** =>orderCategoryIds=RandomNum.*multi*(1,*cargoryNum*,*RandomNum*(1,5),**","**,**false**)  orderProductIds=RandomNum.*multi*(1,*cargoryNum*,*RandomNum*(1,5),**","**,**false**)  **case "pay"** =>payCategoryIds=RandomNum.*multi*(1,*cargoryNum*,*RandomNum*(1,5),**","**,**false**)  payProductIds=RandomNum.*multi*(1,*cargoryNum*,*RandomNum*(1,5),**","**,**false**)  }   **val** userVisitAction = *UserVisitAction*(  actionDateString,  userId.toLong,  sessionId,  *RandomNum*(1, *pageNum*).toLong,  actionDateTimeString,  searchKeyword,  clickCategoryId.toLong,  clickProductId.toLong,  orderCategoryIds,  orderProductIds,  payCategoryIds,  payProductIds,  cityId  )  rows+=userVisitAction  }  }   }  rows.toList  }   **def** mockProductInfo()= {  **val** rows=**new** ListBuffer[ProductInfo]()  **for** (i <- 1 to *productNum* ) {  **val** productInfo = *ProductInfo* (  i,  **"商品\_"**+i,  *productExRandomOpt*.getRandomOpt()  )  rows+=productInfo  }  rows.toList  }    **def** mockCityInfo()= {  *List*(*CityInfo*(1L,**"北京"**,**"华北"**),*CityInfo*(2L,**"上海"**,**"华东"**),  *CityInfo*(3L,**"深圳"**,**"华南"**),*CityInfo*(4L,**"广州"**,**"华南"**),  *CityInfo*(5L,**"武汉"**,**"华中"**),*CityInfo*(6L,**"南京"**,**"华东"**),  *CityInfo*(7L,**"天津"**,**"华北"**),*CityInfo*(8L,**"成都"**,**"西南"**),  *CityInfo*(9L,**"哈尔滨"**,**"东北"**),*CityInfo*(10L,**"大连"**,**"东北"**),  *CityInfo*(11L,**"沈阳"**,**"东北"**),*CityInfo*(12L,**"西安"**,**"西北"**),  *CityInfo*(13L,**"长沙"**,**"华中"**),*CityInfo*(14L,**"重庆"**,**"西南"**),  *CityInfo*(15L,**"济南"**,**"华东"**),*CityInfo*(16L,**"石家庄"**,**"华北"**),  *CityInfo*(17L,**"银川"**,**"西北"**),*CityInfo*(18L,**"杭州"**,**"华东"**),  *CityInfo*(19L,**"保定"**,**"华北"**),*CityInfo*(20L,**"福州"**,**"华南"**),  *CityInfo*(21L,**"贵阳"**,**"西南"**),*CityInfo*(22L,**"青岛"**,**"华东"**),  *CityInfo*(23L,**"苏州"**,**"华东"**),*CityInfo*(24L,**"郑州"**,**"华北"**),  *CityInfo*(25L,**"无锡"**,**"华东"**),*CityInfo*(26L,**"厦门"**,**"华南"**)    )  } |

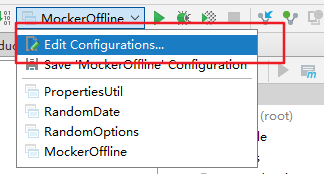
## 3.4 项目执行的问题

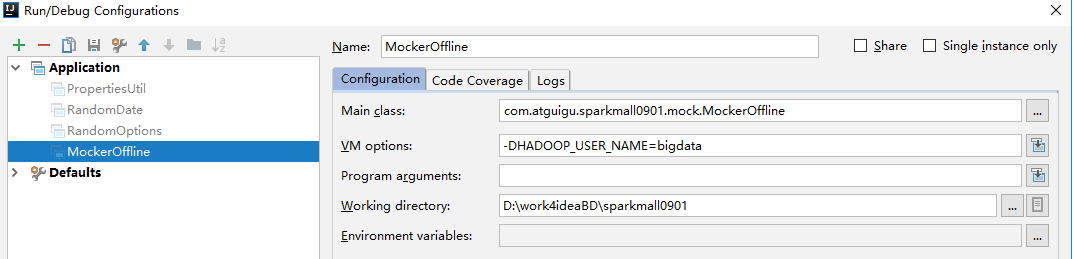
#### 3.4.1 Hadoop文件写入权限不足

在windows的idea中访问执行时，会向linux下的hadoop提交文件，可能会发生权限问题，原因是执行时，idea会将当前windows登陆用户做为hadoop用户来提交文件。因此会导致权限不足的问题

解决办法 ：在执行程序的Edit Configurations中 做如下设置，把VM options或Environment variables中加入

-DHADOOP\_USER\_NAME=xxxxxx （你的hadoop用户）





#### 3.4.2 hive访问失败

可能的原因是hive-site.xml误修改了元数据存储库名称。