# 相关技术与理论基础

## 总体技术概览

协同过滤可以根据相似度计算对象的不同分为UserCF和ItemCF。

开源框架Apache Hadoop可以使用多台廉价的机器组成一个分布式集群，它的三个核心组件HDFS、Yarn、MapReduce构成了大数据处理的三驾马车。HDFS提供了分布式文件存储，Yarn管理整个集群的资源和任务调度，MapReduce实现分布式的并行计算。

## 系统开发环境

前端使用Vue3.0+Vite2.0+ElementPlus UI在Windows 10系统完成开发，开发工具使用Vscode；后端使用Go语言在CentOS 8服务器上完成开发，开发工具使用Goland；大数据部分采用一台2核4G内存与两台1核2G内存的云服务器构成完全分布式的集群部署，操作系统均为CentOS 8.0；模型训练使用GPU服务器完成，显卡型号GeForce RTX 3090，显存24GB。

## 数据收集模块

谷歌的研发总监Peter Norvig曾说过：“更多的数据优于更好的算法，而好的数据优于多的数据”。这句话强调了数据对于整个系统至关重要。本次设计需要使用离线与在线的数据为队员们进行推荐，离线数据主要来源于Codeforces训练记录的爬取；在线数据主要来源于用户在网站上操作（构建知识库、提交题解、回复问题等），需要通过埋点日志记录并通过消息队列传递给Flink进行消费。下面介绍数据爬取模块时的技术选型和选取原因。

### 编程语言Golang

Golang（简称Go）是由Google公司的Robert Griesemer，Rob Pike和Ken Thompson在2009年开源的一种编程语言，并于2012年3月发布了第一个稳定版本。

为什么本文选择Go作为系统开发的主要语言？

**1. 性能优势：**简单是Go语言的核心，这意味着它可以快速编译，快速运行并且快速上手开发。实测中Go拥有着近乎C++的编译速度与Python的开发效率。

**2. 并发：**独特的并发设计是Go区别其他语言的最大特点。Go的并发哲学源自CSP理论[22]，选择Channel通信而不是加锁来控制共享资源，从设计上避免了数据竞争的问题。而并发的基本单位协程Goroutine相比线程来说，避免了操作系统内核对线程的调度，减少了大量CPU用来保存上下文和恢复现场的时间，对操作系统完全透明。这也是Go在编写HTTP服务器方面表现出色的主要原因之一。

**3. 跨平台且易于维护：**Go编译生成的二进制文件可以在Windows，Linux，macOS或其他平台上直接运行，降低了部署的难度。

### 爬虫框架Go-Colly

由于整个服务端由Go语言构建，便没有选择知名的Python爬虫框架Scrapy，而是采用了Go语言的Colly爬虫框架。下表2-1展示了两者之间的差异：

表 2‑1 爬虫框架Colly与Scrapy对比

|  |  |  |
| --- | --- | --- |
| **对比内容** | **Colly** | **Scrapy** |
| 爬取速度 | 较快 | 较慢 |
| API设计 | 简洁 | 每个版本都有改变 |
| 并发支持 | 原生支持 | 需要配置 |
| Robots协议支持 | 有 | 有 |
| 分布式支持 | 支持 | 需要配置 |
| 解析库支持 | Goquery | Pyquery / BeautifulSoup等 |
| 定时任务 | 不支持 | 支持 |
| 扩展性 | 不支持 | 有许多第三方扩展 |

相较于Scrapy而言，作为一个新生爬虫框架，依靠着Go语言强大的并发性能和简洁的语言风格，在爬取速度和使用难易程度上都有一定优势。但是在可扩展性和爬取后数据的分发，都和生态完整的Scrapy框架有着一定的差距。

### 定时任务Cron

由于Go-Colly框架不支持定时任务，结合笔者三台服务器被挖矿木马入侵时crontab文件被异常写入的经历，发现了Go-Cron库。其实现了类似于Linux系统Crontab服务的功能，利用这一功能完成了自动化定时爬取。

使用Cron不仅仅能够实现定时任务，还支持为定时任务自定义许多其他功能，让爬虫服务变成更加稳定以及便于维护。下表2-2列举了这些功能和实现意义：

表 2‑2 Cron自定义功能及其意义

|  |  |
| --- | --- |
| **功能** | **意义** |
| WithLocation | 指定时区 |
| WithParser | 使用自定义的解析器 |
| WithLogger | 自定义log来记录任务运行情况 |
| ThreadSafe | 显式的处理回调 |
| WithChain | 自定义JobWrapper，可以用来捕获异常、恢复上次未执行完的Job、跳过失败任务等 |

### 日志收集Zap

Go官方提供的日志库Go Logger虽然使用起来非常的简单，但是仅支持基本的日志事件，不支持输出INFO、DEBUG等多个日志等级。对于错误事件也仅仅是在抛出panic之前记录一条日志，缺少ERROR日志等级，无法在不退出程序的情况下记录错误。同时输出的日志不支持格式化，对接下来的数据处理和分析造成影响。Zap是由uber开源的高性能Go语言日志库。第三方的日志库有许多，但只有Zap针对语言底层，实现了一个无反射，零分配的JSON编码器，减少了在序列化数据时造成的时间开销。下表2-3展示了Zap与其他Go语言日志库的性能差距：

表 2‑3 Zap与其他日志库性能对比

|  |  |  |  |
| --- | --- | --- | --- |
| **Package** | **Time** | **Time % to zap** | **Objects Allocated** |
| zap | 862 ns/op | +0% | 5 allocs/op |
| zerolog | 4021 ns/op | +366% | 76 allocs/op |
| go-kit | 4542 ns/op | +427% | 105 allocs/op |
| apex/log | 26785 ns/op | +3007% | 115 allocs/op |
| logrus | 29501 ns/op | +3322% | 125 allocs/op |
| log15 | 29906 ns/op | +3369% | 122 allocs/op |

从上表可以看出，Zap极大的减少了每次操作所需要的资源分配，比另一个热门日志库logrus性能提升了33倍。在实用性方面，Zap也弥补了上述所提到的官方库缺点，让我们可以随意的定制所需要的日志输出。

### 消息队列Kafka-Go

为了能够将每次用户的点击日志实时的交付给Flink进行处理，需要使用一个消息队列来进行数据的传输。笔者在对比了当下四种Go连接kafka的客户端之后选取了Kafka-Go。下面列举各个客户端并说明不选取得理由。

**1. sarama：**虽然是目前star数最多的客户端，但是非常难用。缺少文档、不支持Go语言上下文、将数值作为指针传递，导致大量动态内存分配和频繁的垃圾回收。

**2. confluent-kafka-go：**它是围绕librdkafka（一个C++的kafka客户端）的cgo包装器，这意味着它将对使用该包的所有Go代码引入C库的依赖关系。它具有比sarama更好的文档，但是仍然缺乏对Go上下文的支持。

**3. goka：**这是一个Go-Kafka的最新客户端，但是缺少对事件有序日志的支持。

**而**kafka-go提供了低级和高级API来与Kafka进行交互，并实现Go标准库的接口，从而使其易于使用并与现有软件集成。

## 大数据特征处理模块

随之收集而来的数据越来越多，单个服务器可能无法保证数据的可靠存储。大数据技术是整个系统数据存储、清洗与训练中不可缺少的一环。本次设计采用的大数据技术相互之间的联系如下：

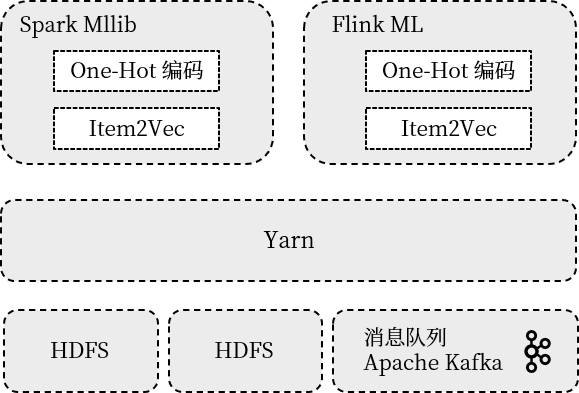


图 2‑1大数据技术关系图

如上图2-1所示：Hadoop HDFS作为分布式文件存储，是所有落盘数据最终的去处。消息队列Kafka作为Flink的数据来源，将用户的日志流交给其进行消费。Yarn作为资源调度和任务分配的中心对Spark和Flink的任务进行管理。而Spark Mllib与Flink ML分布式并行处理特征数据，并将生成的Embedding传给TensorFlow完成模型训练。

### 分布式文件系统HDFS

由于系统中主要使用HDFS作为分布式文件存储，而MapReduce的任务被Spark取代，在此主要介绍以下HDFS实现高可用的原理。

HDFS从设计之初就假设存储节点并不可靠，因此，检测故障并快速自动地从故障中恢复是HDFS的核心目标。HDFS的主要组件包括一个namenode和其管理的多个datanode，以及Secondarynamenode检测节点。

**1. Namenode(NN)：**HDFS的唯一主节点。主节点可以跟踪文件，管理文件系统，并在其中具有所有已存储数据的元数据。并直接与客户端联系。

**2. Secondarynamenode(SN)：**名称节点的检查节点。指示名称节点创建并向其发送镜像文件和操作日志，然后由SN创建压缩镜像。当主节点出现故障时，SN会利用备份的镜像文件帮助其快速恢复。

**3. Datanode（DN）：**数据节点。每3秒向主节点报告一次，超过2分钟将被视为掉线，这时由其他节点上存储的备份数据会重新构建一个DN。

下图2-2展示了HDFS的组织架构：



图 2‑2 HDFS架构图

### 分布式计算平台Spark

Spark生态系统如下图2-3所示：

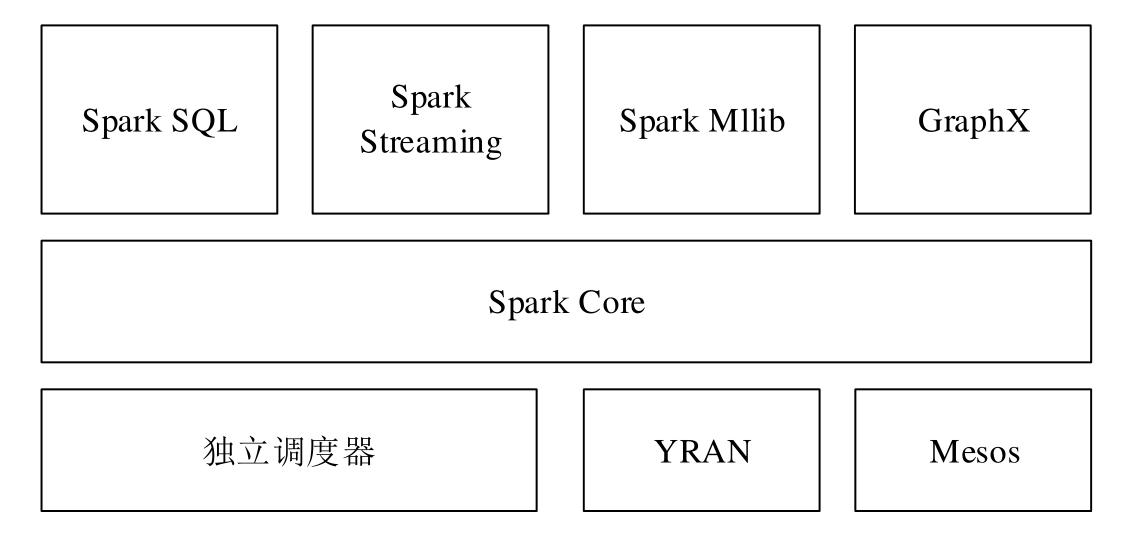


图 2‑3 Spark生态系统

Spark是一个面向内存的计算框架，这使得它的计算效率远远超过MR任务。而SparkSQL更是在此基础上抽象出DataFrame数据结构，并为其做了大量的性能优化，使得Python/Java等语言有了等同于原生scala编程的执行效率。

下表2-4列出了Spark各个模块的功能：

表 2‑4 Spark各模块功能表

|  |  |
| --- | --- |
| 模块名 | 功能 |
| Spark Core | **Spark Core** 定义操作RDD的相关API，实现内存管理、错误恢复、任务调度、与存储系统交互等基础功能。 |
| Spark SQL | **Spark SQL**用于操作各种结构化数据，通过Spark SQL，可以使用SQL来查询或修改数据。 |
| Spark Streaming | **Spark Streaming** 作为流式计算组件，定义用于操作实时数据流的相关API。 |
| Spark MLlib | **Spark MLlib** 定义一些常见的机器学习相关算法和功能，如分类、聚类、回归分析、协同过滤等，同时支持数据导入、模型评估等额外功能。 |
| 集群管理器 | **集群管理器**负责整个集群的资源管理以及节点之间的通信，从而灵活地实现集群规模的伸缩扩展。Spark 支持在 Hadoop YARN、Apache Mesos 等多种集群管理器上部署和运行。 |

### 消息队列Kafka

在消息队列的技术选型上，主要有Apache开源的Kafka和基于Go语言的Nsq。虽然Nsq拥有着更简单的部署方式和更高的传输性能，但是由于其不能保证消费数据的有序性，最终本论文采用Kafka来作为消息队列。

下图2-4展示了kafka一种典型的应用场景并对其相关概念进行介绍：

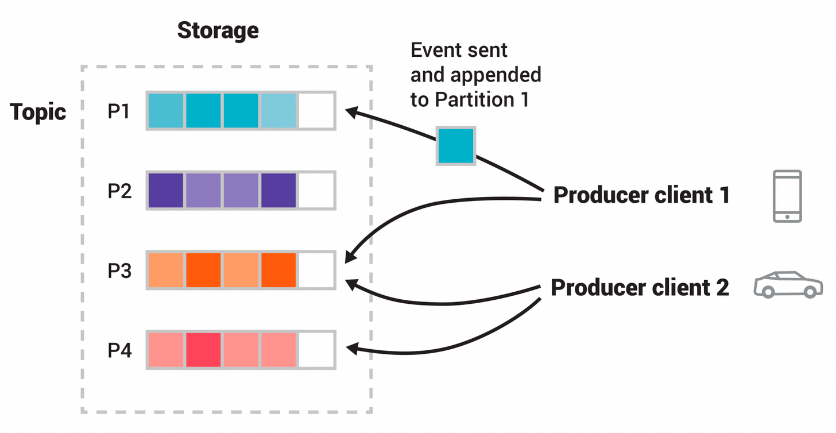


图 2‑4 Kafka典型应用场景

**1. Event：**一个事件记录了消息的所有元数据（如键、值、时间戳等）

**2. Topic：**是所有事件的一个文件夹，可以同时被多个消费者读取。

**3. Storage：**所有的Topic被持久化保存在磁盘上（Nsq是在内存中）。

**4. Partition：**主题分区，代表着一个事件存储在多个Kafka客户端内。

**5. Producer：**生产者是向Kafka写入事件的客户端应用程序。

**6. Consumer：**消费者是读取和处理事件的客户端应用程序。