## 数据系统

随着大数据的普及，各行各业都已经认识到数据重要性。如今我们在设计产品时除了充分考虑业务方面的需求，常常也很关注的数据产生的价值。甚至我们会充满信心地笃定数据才是系统的核心。

例如，我们的业务系统在部署上线后我们肯定不会忘了通过各种手段去监控系统的运行状态。在设计一款App应用的时候，我们同时会关注它上线后的用户量、用户每天的使用时长等行为信息。在设计开发一套物联网系统来监测商务大厦的使用率、能源消耗、安全警告时，我们也会从长远地思考如何挖掘这些数据，来对各种设备做预测性维护或者设计更好的节能方案。

无论是系统监控、App运营、预测性维护还是能源优化，都需要对各种数据的深入分析和挖掘，而这一切都离不开数据系统。数据系统包括了数据相关的流程，并且本书中的数据系统独立于业务系统。

### 数据系统的通用流程

无论大数据数据分析或是依据关心性数据库的数据分析流程，都会包括数据获取、数据存储和处理、数据应用三大步骤。

日志文件

源数据库

ETL

源数据

CDC

分析

数据

可视化

数据探索

机器学习模型训练

数据服务接口

#### 数据获取

获取源数据一般通过主动拉取(pull)或被动接受(push)。通常使用主动拉取的方式，程序更容易进行异常处理，具体我们将在第x章详细分析。

#### Staging数据暂存区

数据暂存，这个步骤一般不会避免。。。。

#### 数据处理和存储

我们面对的数据源可能包括非结构化数据、结构化业务数据。非结构化数据常见的是系统日志，数据需要经过ETL过程整理后进入数据库或者数据仓库，以便后续的处理。

业务数据通常存储在数据库中，属于结构化数据。我们在数据分析过程中可以直接从业务数据库中查询，但是业务数据的模型（表结构、索引等）通常对OLTP更高效，而对数据分析并不优化。另一方面，数据分析会查询大量的数据做统计，直接在业务数据库上查询也会给业务系统带来性能负担。我们需要将原始数据进行抽取、转换（ETL）成对分析高效的数据模型。

所以无论是结构化还是非结构化数据，我们通常都要将原始数据抽取到一个独立的数据存储，然后再进行数据计算或分析。值得注意的是，从源数据库抽取数据通常采用增量方式，通常可以根据数据记录时间戳增量抽取，或者利用数据修改的日志增量抽取（CDC，Changed Data Capture）。

数据处理可以离线批处理或者实时（准实时）方式。。。。

第x章讨论lambda架构是会详细讨论。。。

#### 数据应用

TODO

## 办公大厦数据分析案例

本节将通过办公大厦占用率分析和能源优化需求作为案例，讨论基于关系型数据库的数据分析流程。从本案例中我们能够直观地了解数据分析的流程，并以此为基础引出对大数据技术的诉求。

办公大厦每天的运营将产生很多数据，包括工作位子（工位）占用数据，大厦的能耗：照明、空调、水泵等。我们希望通过挖掘大厦占用率和能耗数据的关系，尝试发现异常能耗状况，进而优化能耗。目前假设大厦已经部署了各种传感器和完整的监控系统，我们计划在此基础上开发一套相对独立的数据分析系统。

### 数据源

* 电表读数

电表可以获取很多读数，为简化问题我们只关注瞬时功率。电表也有物理位置，不同于工位传感器的是，电表收集的是一片区域或“回路”的电能能耗，并且照明、空调等分不同的电表监测。电表数据每5分钟想大厦监控系统上传一次读数。

* 工位传感器

我们在每个工位上方安装红外传感器，获取工位占用状态（占用/未占用）。工位有物理位置信息，例如3楼A区的第1号工位。传感器每5分钟向大厦监控系统上传一次状态数据。

* 温湿度读数

温湿度读数可以通过空调控制面板数据或者温湿度传感器收集室内不同位置的温湿度。

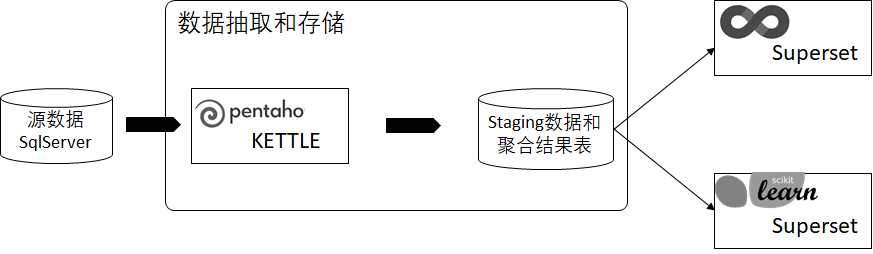
### 数据分析策略

电表读数是我们关系的优化对象，直觉上将不同检测类型的电表需要区分开去分析。每个电表读数(meter)的可能影响因素包括工位占用率 (occupy rate) 、温度(temperate)、湿度(humidity)：

对一年内的数据，我们分别统计各个电表读数跟影响因素之间的回归关系，挖掘异常状态，进而调查原由并采取优化措施。第x章详细介绍了一些异常检测的算法。

### 数据流

监控系统数据存储在关系型数据库，我们首先也采用关系型数据库方案对数据抽取、转换、可视化和数据挖掘等。



#### 数据获取和ETL

收据抽取后存储的RDBM有很多可选，MySQL，Postgres等，我们选择Postgres。大厦监控系统的数据库是SqlServer，那么数据获取过程就是典型的数据库到数据库的抽取过程，常用的开源或免费系统也比较多：Kettle，Talend等，我们选用Kettle。

如图，Kettle定时每小时从原始数据库抽取数据，并暂存在Staging表中。接着数据将进一步做聚合计算，我们将数据按小时聚合。例如，对每个工位每5分钟上传一次状态数据，kettle整点时做一次聚合，计算工位在上一小时内总占用次数和总空闲次数。

数据小范围内聚合一方面可以减少数据量，降低数据之间的冗余度。另一方面，各类传感器、电表的数据上传时刻不一定相同，时间点不对齐，无法直接根据时间关联各类数值。按小时聚合后，我们将每个电表每小时的电表读数、温度平均值、湿度平均值、控制范围内工位占用率作为一条数据分析输入记录。

#### 统计预计算和报表可视化

有了聚合数据，我们借助开源数据可视化系统可以方便的在多种维度上对数据进行统计和展现。常见的开源数据可视化系统有Superset、Redash和Tableau等，我们选用Superset。

|  |  |
| --- | --- |
|  | 数据展现的图表通常需要查询较大时间范围内的数据，例如左图统计“过去30天内早上9点到下午6点，按小时统计大厦工位占用率”。我们需要将每个小时的数据跨30天做聚合。为了使数据统计更快速，我们可以周期性地预先计算每个图表所需的查询结果。 |
|  |  |

#### 数据探索和挖掘

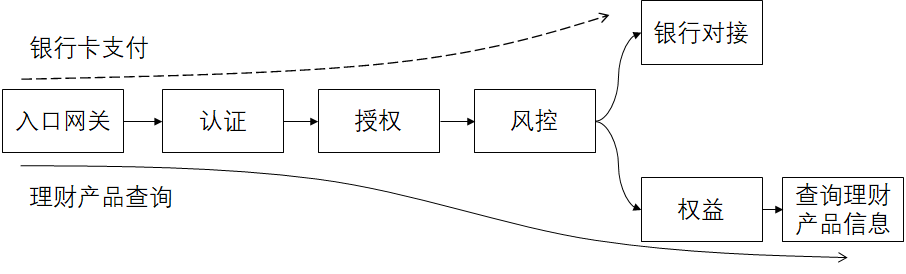
为了挖掘异常的能耗使用，我们需要分析各个电表的数据和影响因素的回归关系。需要挖掘哪些因素对能耗有显著性，建立影响因素跟能耗使用之间的基准关系。这个过程需要借助领域知识、机器学习等技能。我们选用Sklearn作为机器学习工具，利用多元线性回归建立能耗基准模型。

## 企业业务性能监控系统案例

办公大厦数据分析系统采用的是基于传统的关系型数据库方案，当数据量逐渐增大后关系型的存储和查询性能都将出现瓶颈。可以通过增加内存、CPU和磁盘容量等手段提高服务器性能，但是垂直扩展单机性能的成本很高，而且这种扩展很容易达到性能上限。这些因素带来了大数据技术的出现和发展。

本节将介绍基于大数据的业务性能监控系统，简单介绍系统架构、数据流以及各大数据组件。本书的后续章节将详细介绍这个大数据系统中的各个组件的设计思路。

成熟的企业业务架构通常不是单一架构，而是由众多子模块或子系统构成。子模块更抽象也更通用：具体的业务功能依赖子模块，众多子模块通过排列组合可以构建出不同的业务功能。



上图是简化的两个业务功能，“银行卡支付”功能依赖的子模块包括认证、授权、风控以及银行对接。其中认证、授权、风控子系统也被“浏览理财产品”功能依赖。我们将介绍的业务性能监控系统，监控对象包括各子模块的整体性能，以及各子模块分解到具体业务功能的性能。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 子系统 | TPS | 成功率 | 成功调用平均耗时 | 失败调用平均耗时 | 平均耗时 |
| 认证 |  |  |  |  |  |
| 授权 |  |  |  |  |  |
| 风控 |  |  |  |  |  |
| 银行对接 |  |  |  |  |  |
| 权益系统 |  |  |  |  |  |
| 理财产品查询系统 |  |  |  |  |  |

表x：各子模块整体性能监控

将各个子模块性能指标分解(breakdown)到各个业务功能：

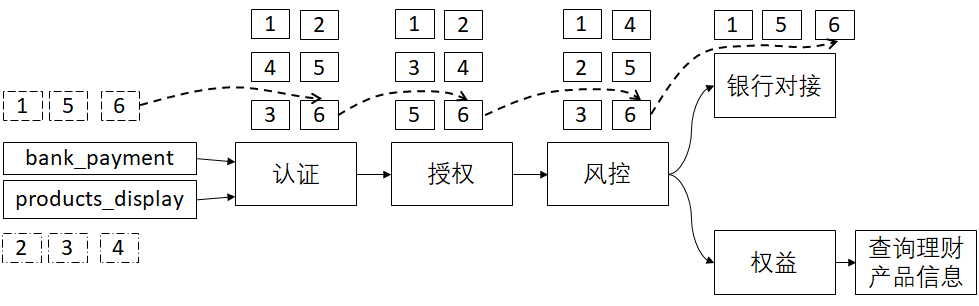
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 业务功能 | 子系统 | TPS | 成功率 | 成功调用平均耗时 | 失败调用平均耗时 | 平均耗时 |
| 银行卡支付 | 认证 |  |  |  |  |  |
| 授权 |  |  |  |  |  |
| 风控 |  |  |  |  |  |
| 银行对接 |  |  |  |  |  |
| 查询理财产品 | 认证 |  |  |  |  |  |
| 授权 |  |  |  |  |  |
| 风控 |  |  |  |  |  |
| 权益系统 |  |  |  |  |  |
| 理财产品查询 |  |  |  |  |  |

表x：各模块分解性能监控

理论上认证、授权以及风控系统往往无需感知具体的业务，例如认证服务的功能是验证用户的登录信息是否合法，它不必关心认证的具体目的是银行卡支付或是其它。具体业务功能只需将子模块进行组合即可。

然而，如果子系统完全无感知具体业务信息，会给开发运维定位问题带来困难：我们无法快速过滤出异常发生的调用链上各子模块的日志。因此子模块被调用时，会传递调用链相关的信息。例如在“bank\_payment”入口服务处为链路调用生成一个“链路号”(link\_id)，并在链路上各个子模块接口调用时将link\_id传递下去。研发或运维人员在定位具体调用异常时，可以详细分析各个子模块中有关异常link\_id的日志信息，进而定位问题。

我们在设计业务性能监控系统时也需要依赖link\_id，例如我们需要监控“银行卡支付”功能，在数据处理过程中首先筛选出”bank\_payment”系统日志中的link\_id，再根据这些link\_id在随后的各个子模块系统日志中筛选出银行卡支付相关的日志。



如上图中，银行卡支付产生的link\_id分别有1, 5 ,6。以link\_id=6为例，串联出各子模块中有关link\_id=6的系统日志数据。经过这种日志串联处理，我们便能将各子模块的日志分解到具体业务功能。

### 数据源

业务子系统产生的日志文件，文件存储在业务系统服务器本地。日志主要分为两类：

### 子系统的API请求调用开始日志

日志中将包含以下信息：

|  |  |  |  |
| --- | --- | --- | --- |
| timestamp | API name | link id | “API\_START” |

**timestamp**： 请求开始时间；

**API name**：请求接口名称，例如“风控接口”的名称为risk\_management；

**link id**：子模块被调用时，上下文中会包含调用链id。

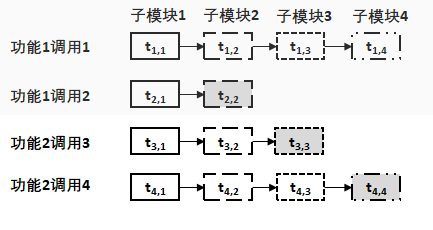
#### 子系统的API请求调用完成日志：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| timestamp | API name | link id | “API\_FINISH” | status |

**timestamp**： 请求完成的时间；

**status**： 接口调用状态，成功或失败。

### 系统监控策略



#### 统计各子系统调用性能

按分钟、小时、天时间维度，分别统计每个子系统接口的性能基础指标：

|  |  |
| --- | --- |
| 统计项 | 描述 |
| 调用频数 () | 接口开始(start)次数 |
| 成功调用次数 (*SN*) | 接口完成(finish)且成功的次数 |
| 失败调用次数 (*FN*) | 接口完成(finish)且失败的次数 |
| 成功调用耗时 (*ST*) | 接口完成(finish)且成功的耗时累加 |
| 失败调用耗时 (*FT*) | 接口完成(finish)且失败的耗时累加 |

基于接口的性能基础指标，我们可以计算以下给定时间范围() 内的统计量，用于数据可视化等。以下基础指标都添加了时间下表*t*，例如表示时间*t*（例如2019-02-03 12:05一分钟内的）的调用频数。简化记：

1. 调用频数 ，时间范围内调用频数之和
2. 成功率
3. 失败率
4. 异常率
5. 调用成功的平均耗时
6. 调用失败的平均耗时
7. 调用的平均耗时cost

一般调用总会成对地出现开始(start)和完成(finish)日志，但也会有异常情况，例如只出现开始日志，不会出现完成日志，我们将这类调用归为异常调用。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | *N* | *SN* | *FN* | *ST* | *FT* |
| 子系统1 | 4 | 4 | 0 |  | 0 |
| 子系统2 | 4 | 3 | 1 |  |  |
| 子系统3 | 3 | 2 | 1 |  |  |
| 子系统4 | 2 | 1 | 1 |  |  |

#### 统计业务功能端到端调用链性能

按分钟、小时、天时间维度，统计具体业务功能的调用链上各子模块性能指标。链路上N个子模块，。链路上第i个子模块的统计指标如下：

|  |  |
| --- | --- |
| 统计项 | 描述 |
| 调用频数 () | 开始(start)次数 |
| 成功调用次数 () | 完成(finish)且成功的次数 |
| 失败调用次数 () | 完成(finish)且失败的次数 |
| 成功调用耗时 () | 完成(finish)且成功的耗时累加 |
| 失败调用耗时 () | 完成(finish)且失败的耗时累加 |

基于每个子模块的性能基础指标，我们可以计算以下给定时间范围() 内每个子模块的统计量，用于数据可视化等。以下基础指标都添加了时间下表*t*，例如表示时间*t*（例如2019-02-03 12:05一分钟内的）的调用频数。简化记：

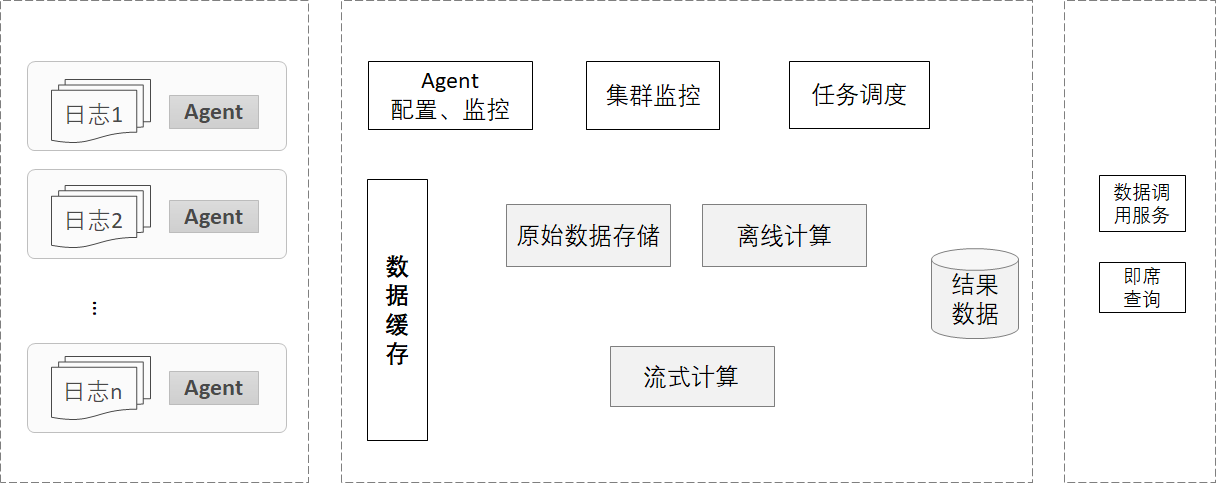
1. 调用频数 ，时间范围内调用频数之和
2. 成功率
3. 失败率
4. 异常率
5. 调用成功的平均耗时
6. 调用失败的平均耗时
7. 调用的平均耗时

我们还可以进一步利用漏斗模型展现更丰富的信息，例如的转化率：

1. 转化率 conv\_

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | *N* | *SN* | *FN* | *ST* | *NT* |
| 功能1 | 子模块1 | 2 | 2 | 0 |  | 0 |
| 子模块2 | 4 | 3 | 1 |  |  |
| 子模块3 | 3 | 2 | 1 |  |  |
| 子模块4 | 2 | 1 | 1 |  |  |
| 功能2 | 子模块1 | 2 | 4 | 0 |  |  |
| 子模块2 | 4 | 3 | 1 |  |  |
| 子模块3 | 3 | 2 | 1 |  |  |
| 子模块4 | 2 | 1 | 1 |  |  |

### 数据流



#### 数据获取

各子系统需要将性能监控所需的参数写入本地日志文件。在子系统运行的服务器上启动Flume agent，定时读取文件并过滤出性能监控所需的内容，并将数据推送到数据缓存(Kafka)。

为了能够灵活应对监控需求的变更， flume agent的配置信息可以通过可视化界面动态更新。当配置更新后，各agent都会重启内部数据获取线程。通常agent的部署规模可能成百上千甚至更多，我们部署了agent自身状态监控告警服务。我们将在第x章详细讨论数据获取相关的设计。

#### 数据缓存

引入数据缓存有两方面的考虑，一方面多种数据消费组件可同时从缓存中获取数据，例如本系统中数据被流式计算和离线计算消费；另一方面数据消费组件可以根据自身的消费能力控制数据消费节奏，消费进程可以做离线更新且无数据丢失之忧。Kafka是大数据生态圈中使用得最广泛的组件之一。

#### 在线数据计算

在线计算及时消费数据缓存中的数据，数据不被持久化而是直接被在线处理，并将统计结果存储到数据库或推送到消息队列。在线计算一般可分为流式计算和小批量计算两类。我们选择的Spark streaming属于小批量计算，一方面性能监控能够容忍秒级别的延迟；另一方面我们在计算子模块，接口调用耗时的时候，需要结合 单次调用的“开始” 和 “完成” 两条日志数据，将日志中时间戳相减计算，基于小批量的处理模式更容易实现这种计算。

需要注意的是“开始”和“完成”两条日志数据可能不在一个批量数据中，此时需要考虑跨batch的统计。此外，调用链数据处理过程中，我们需要将同一个link\_id的日志串联在一起处理日志，大数也会涉及到跨多个batch统计，我们将在第x章详细介绍跨batch数据处理的细节。

#### 离线数据计算

在线计算具有较高的实效性，但是却可能带来计算误差。例如在统计某个子系统被调用次数时，在线计算将一个batch中该子系统调用次数做累加操作后更新到数据库(HBase)。如果出现异常，我们无法判断HBase中是否成功更新了最后一个batch中的数据更新是否成功。

离线数据计算可以弥补在线计算的计算误差，但却具有更大的计算延迟。离线计算通常先将数据缓存中的数据存储在文件（如HDFS）中，再定时执行离线计算程序。在大数据计算中我们常常将离线和在线计算结合，既能达到离线计算的准确性又能拥有在线计算的实效性。在第x章我们将详细分析数据到达的三个语义以及Lambda架构模式。

除了数据离线计算外我们通常还会使用类Sql的数据查询工具，例如测试人员需要这些工具来验证数据处理结果的正确性，常用的查询工具包括Hive, Presto等。

#### 存储

HBase

#### 数据可视化

数据服务

#### 业务故障数据分析

## 解决数据获取问题

数据获取是源数据和数据系统之间的过度步骤，数据获取组件可以看作业务系统和数据系统之间的桥梁，是数据系统入口处的组件。本章将从数据获取组件的容错性、获取模式方面展开讨论，并给出在实际应用场景中基于Flume的设计方案。

### 数据获取组件的鲁棒性

数据获取组件的稳定性对整个数据系统而言至关重要，它的鲁棒性决定了数据系统的整体性能。如果数据获取时常发生数据丢失，会直接影响到数据统计的准确性，想要回溯问题然后再将数据补救回来是很麻烦的。另外数据的丢失，在数据建模时需要额外考虑缺失数据的影响，这会增加模型的复杂性。

我们将数据获取组件可能面对的异常分为两类：数据组件自身异常和下游消费服务异常，我们将在数据获取模式部分详细讨论不同获取模式应对这种异常的方式。

#### 数据获取组件自身异常

首先考虑数据组件自身的异常状况，如果组件自身异常退出数据将无法正常获取，待异常处理完组件重启我们希望数据能从上次异常退出点开始重新获取数据。

#### 下游数据系统组件异常

如果数据获取组件正常，但是数据系统中下游组件异常，比如数据暂存或数据缓存中间件不可用，此时数据获取组件不断获得数据源的数据，但是下游数据组件无法正常接收数据。我们希望数据组件能缓存这部分未消费数据，待下游服务恢复正常后，再将缓存区数据发送给下游消费组件。

数据缓存属于数据获取模块自身的功能而不依赖上下游组件，通常的做法是将未被消费的数据缓存在本地文件系统。设计和实现时需要考虑操作事务、异常回滚等。例如Flume的file channel将数据持久化在本地，待下游sink恢复后再重试将数据发送给下游并清空file channel。Flume基于事务实现了这一过程的一致性，是应对下游数据服务异常的不错选择。

### 数据获取的模式划分

现实应用中数据获取的场景非常多样，我们可以从多个角度来划分数据获取模式。

* 从数据获取方式角度划分，数据获取一般分为主动拉取（pull模式）和被动接受（push模式）；
* 从数据获取的数据内容角度划分，数据或可以分为原始数据模式（raw data）和事件源模式（event sourcing）；

|  |  |  |
| --- | --- | --- |
|  | pull模式 | push模式 |
| Raw data |  |  |
| Event Sourcing |  |  |

#### 数据获取方式划分模式

可以通过一个类比来直观理解主动拉取和被动接受数据两种模式。在交互式互联网电视(IPTV)出现之前，传统电视节目都是直播，观众需要准时打开电视机看自己喜欢的节目，如果哪天错过了一档节目则很难再有机会看到它。观众无法随心所欲地选择重看或者反复欣赏某个节目。传统电视类似于我们将要介绍的数据 “被动接受” 模式。

交互式互联网电视则类似数据 “主动拉取” 模式，观众可以在任何时间点播自己喜欢的节目。有了回看功能，观众不必担心会错过两个同时播出的节目，甚至还可以翻看三年前某天的新闻报道并重新解读报道的含义。这样关注就能够根据自己的日程安排和欣赏速度观看节目了。

##### Pull模式

Pull模式的数据系统主动从数据源拉取数据，可以拉取到任意时间点处的一段数据：

data = **pull**(start\_time, stop\_time).

通常采用定时拉取的方式。数据拉取模式的适用有一个前提，源数据系统需要保存一定时间内所有数据（甚至所有数据），并且提供数据的远程访问接口。

Pull模式下，数据拉取组件能够根据自己的消费性能（吞吐量等），在一定的延迟容忍度下控制数据拉取的节奏。如果组件出现异常，待异常修复重启后从异常发生点开始继续拉取数据，数据获取出现间断但不会丢失数据。

这里我们不去讨论远程数据访问的可靠性，我们将忽略任何数据源获取接口异常。这类数据源常见的包括：

###### 日志文件

日志文件都采用append方式写入日志记录文件，在获取日志文件数据时都会记录当前读取的文件的offset。数据获取组件每次拉取一段数据后记录当前offset，如果数据获取组件出现异常后重启，则从记录的offset处开始重新拉取数据。

日志文件的拉取相比其它有一些不同之处，日志文件通常存储在业务系统，所有我们需要在日志文件所在的服务器上部署数据拉取组件。从部署架构角度上看这些组件已经不在数据系统的部署环境，但是我们仍然将其划分为数据系统的功能部分，因为该组件是数据系统引入的，业务系统不需要关心它的存在。

###### 数据库表

从源数据库增量获取数据需要数据表记录每次新增/删除/更新操作的详细时间戳。新增和更新操作比较容易做到，即在每次操作时记录create\_time和update\_time列。这也是数据表模型设计中常用的设计原则。但是对删除操作常常不适用，如果直接将数据删除，我们无法直接查询到表中不存在的记录是在何时被删除的。针对删除操作业务模型在设计是通常也会考虑逻辑删除，即在表模型中增加布尔类型的删除标志列deleted，逻辑删除时设置删除标志并更新update\_time。

有了create\_time、update\_time和deleted列我们可以很方便地进行增量数据拉取，并且在每次拉取后记录当前时间戳。如果数据获取组件出现异常后重启，则从记录的时间戳开始重新拉取数据。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Record\_id | … | Created\_time | Update\_time | Deleted |

###### Kafka等消息中间件

Kafka等消息中间件在大数据系统中应用非常广泛，我们可以发现在大多系统中都采用Kafka作为数据缓存组件。例如，我们可以在业务系统节点部署日志定时读取程序，将日志数据推送到Kafka。

数据获取组件从Kafka中定时拉取数据，并维护当前数据在Kafka中的offset。如果数据获取组件出现异常后重启，则从异常退出时的offset处开始重新拉取数据。数据获取组件可以自己维护Kafka的offset，也可以依赖Kafka服务端维护offset，详细的介绍请参见第x章。

##### Push模式

另一类数据获取模式是被动接受源数据推送过来的数据。如果数据接收组件不可用，我们将丢失未成功接受的数据，所以组件需要更高的鲁棒性。Push模式下组件无法预测源数据的吞吐量，无法控制数据消费的频率，所有需要更高性能和更高吞吐量。

例如我们部署一个Web服务，业务系统调用服务结果口向Web服务post数据。例如在获取App埋点数据时，我们无法从服务端主动拉取移动端的数据，也无法在用户手机上安装数据获取的程序，只能是App主动将数据上传到Web服务。

#### 数据内容划分模式

组件获取的数据内容可分为原始数据和增量日志数据。从日志文件、kafka和web服务获取的数据通常是原始数据，不需要做太多的转换。但数据库数据通常需要是增量日志数据，例如数据库修改和删除记录需要下游组件将修改操作作用在当前数据上才能的得到当前状态。

另一类增量日志数据是数据库CDC( change data capture) 日志。数据库每次发生数据增删改，都会产生一条操作日志，这些日志可以看成是对数据的操作指令。

CQRS？

### 分布式数据获取组件

#### 单点监控

通过脚本定时获取数据获取进程状态，在进程异常退出时自动重启程序。

#### 主备模式

类似单点监控模式，主备模式预先启动多个进程，但只有其中一个进程会获取数据，其它进程“待命”(standby)。一旦主进程异常退出，待命的进程会竞选出一个进程为主进程去获取数据。这种模式需要借助集群管理服务，例如zookeeper。

#### 消费者组

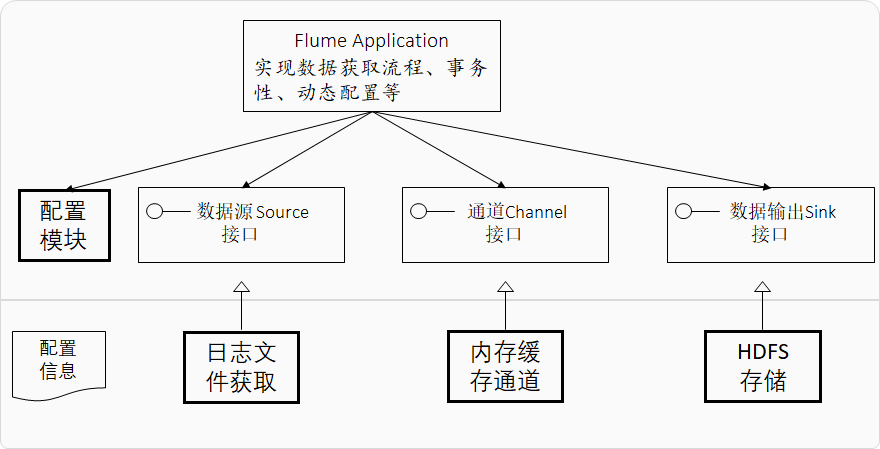
消费者组(consumer group)是目前很多消息中间件的消费者API的功能，可以同时启动多个数据获取组件，每个组件获取部分数据，能确保数据完整且不重复。例如Kafka的consumer group中，启动的每个消费进程消费一个数据分区(partition)，同一个partition被有且只有一个消费进程消费，但有可能一个消费者消费多个partition。当某个消费进程异常退出后，它读取的partition会被Kafka自动分配给另一个消费者进程。

### 基于Flume的数据获取方案

Apache flume (http://flume.apache.org) 是一种分布式的、可靠的、可用的系统，主要用于高效地收集、聚合和移动大量来自不同来源的日志数据到一个集中的数据存储区。

#### Flume的核心设计

Flume Application主程序实现了数据收集的流程，能够保证端到端数据传输的完整性、事务性。Application依赖 “配置模块” 以及 “三个核心模块” 的接口：数据源(source)、数据输出(sink)和数据通道(channel)。在运行时，通过配置模块负责解析用户配置的具体(concrete)核心模块，一般可以通过文件等方式实现配置。



配置模块和三个核心模块接口

两端是可插拔的数据源(source)和数据输出(sink)，用户根据实际的数据类型选择Flume提供的source/sink，或者基于Flume定义的接口协议开发自己的source/sink。

数据通道(channel)可以缓存数据从而提高数据完整性。更高级别的数据完整性必然带来额外的性能代价。通道也暴露给用户，用户可根据对数据完整性的要求，灵活选择通道类型甚至定义特定的通道模块，从而权衡数据完整性和性能。

基于配置语言，用户可以配置具体所需的source/sink。所以Flume留给用户的开发g工作一般包括：

* 根据实际数据存储选择或者自定义开发source/sink模块；
* 选择或自定义开发通道channel模块，权衡数据完整性和性能；
* 创建配置信息，如配置文件，配置Application agent的各个模块、数据库连接信息等。

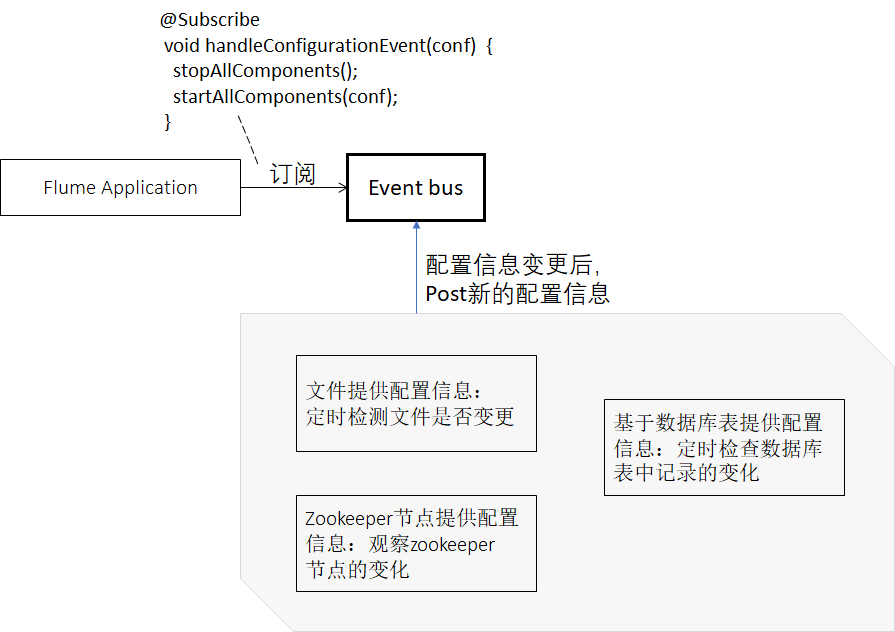
鲁棒性

单点监控+File Channel确保数据完整性

Consumer Group模式

#### 动态更新配置

Flume Application主程序实现了配置信息热更新特性。配置信息提供者(Configuration Provider) 负责配置信息读取、解析并监听配置信息的变更。当发生变更时，配置信息提供者会将新的配置信息推送(post)到事件总线 (event bus) 。主程序中订阅了有关配置信息的事件，一旦收到相关事件则重新启动所有组件。Flume依赖Guava中的eventbus实现配置信息变更事件的推送和订阅(github.com/google/guava/wiki/EventBusExplained).



目前Flume实现了两大类配置信息提供者：基于文件的配置提供和基于Zookeeper node存储的配置提供。基于文件的配置机制，启动后台线程定时检测文件的变更，默认30秒检测一次。

Zookeeper本身是非常优秀的分布式应用协调程序，基于zookeeper node存储配置信息，并绑定一个监听器(listener)到存储节点，一旦存储节点信息发生改变则会触发监听器的回调函数，将新的配置信息推送给事件总线。

如果我们希望通过别的方式存储Flume的配置信息，比如将配置信息存储到数据库，并提供用户界面更方便地修改这些配置，那么我们必须创建新的“配置提供方” (ConfigurationProvider)，并修改Application代码使新的配置提供逻辑生效。

事实上我们可以进一步将“配置提供方”作为Application的初始化参数，实现类似Source/Sink/Channel那样插拔式地配置具体的“配置提供方”组件。示例代码请参见：

状态监控

Flume的广义用法

NATs物联网数据消费。

## 解决存储问题

HBase的优势

HBase的架构

HBase的使用（Phoenix）

性能问题

## 解决离线计算问题

secondary sort

客户数据清洗、整理、分析流程

用户行为数据离线计算（设计模式。。。）

Hive给谁用？

跟MR的比较

## 解决实时计算问题

风控实时统计

用户行为数据实时计算

系统性能监控（sparkstreaming的跨batch血缘关系）

## 解决查询问题

phoenix

Drill

## 解决高级数据分析问题

sql on hadoop: impala, presto, druid

OLAP

Kylin+Cubes

Superset

## 解决机器学习算法

异常检测模型

## 关于算法

大数据处理更需要关心算法

基本算法的力量（网络流）

## 关于架构

Martin大叔的架构书

3种编程模式