

阿里云 日志服务

实时消费

文档版本：20190916

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 禁止： 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	 警告： 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 说明： 您也可以通过按Ctrl + A选中全部文件。
>	多级菜单递进。	设置 > 网络 > 设置网络类型
粗体	表示按键、菜单、页面名称等UI元素。	单击 确定 。
<code>courier</code> 字体	命令。	执行 <code>cd /d C:/windows</code> 命令，进入Windows系统文件夹。
<code>##</code>	表示参数、变量。	<code>bae log list --instanceid Instance_ID</code>
<code>[]</code> 或者 <code>[a b]</code>	表示可选项，至多选择一个。	<code>ipconfig [-all -t]</code>
<code>{ }</code> 或者 <code>{a b}</code>	表示必选项，至多选择一个。	<code>swich {stand slave}</code>

目录

法律声明.....	I
通用约定.....	I
1 简介.....	1
2 普通消费.....	3
3 消费组消费.....	5
3.1 通过消费组消费日志.....	5
3.2 消费组状态.....	12
3.3 消费组监控与报警.....	14
4 函数计算消费.....	17
4.1 开发指南.....	17
4.2 配置函数计算消费日志.....	20
5 Storm消费.....	29
6 Flume消费.....	33
7 ARMS消费.....	37
8 开源Flink消费.....	38
9 Logstash消费.....	45
10 Spark Streaming消费.....	46
11 实时计算（Flink）消费.....	47
12 CloudMonitor消费.....	50
13 Golang Consumer Group 消费.....	51
14 最佳实践.....	52
14.1 消费-通过函数计算清洗数据.....	52
14.2 消费-搭建监控系统.....	60
14.3 消费-计量计费日志.....	62
14.4 消费-通过Consumer Library实现高可靠消费.....	67
14.5 消费-通过ETL清洗数据.....	76

1 简介

数据收集至日志服务 LogHub 后，有三种方法可以处理日志：

方式	场景	实时性	存储时间
实时消费（LogHub）	流计算、实时计算等	实时	自定义
索引查询（LogSearch）	适合最近热数据的在线查询	实时（99.9%的情况为1秒，最大3秒。）	自定义
投递存储（LogShipper）	适合全量存储日志，进行离线分析	5分钟~30分钟	依赖于存储系统

实时消费

日志服务LogHub功能提供Pull接口，支持日志数据实时消费。对于一个 Shard 中的日志，消费过程如下：

1. 根据时间、Begin、End 等条件获得游标。
2. 通过游标、步长参数读取日志，同时返回下一个位置游标。
3. 不断移动游标进行日志消费。



说明：

消费日志与查询日志都意味着“读取”日志，两者区别请参见[#unique_4](#)。

SDK消费

日志服务提供多语言（Java、Python、Go 等）SDK，且这些语言的 SDK 都支持日志消费接口。关于SDK的更多信息请参考[日志服务 SDK](#)。

消费组消费

[消费组](#)是日志服务对 LogHub 消费者提供的高级模式。消费组提供了一个轻量级计算框架，解决多个消费者并行消费 Logstore 的能力，消费组提供自动分配 Shard、支持保序、断点续传等功能，详情请参考[Consumer Library](#)。目前 Go、Python、Java 等语言均提供消费组 SDK。

流计算系统消费

- [Spark Streaming Client](#)。
- [Storm Spout](#)。
- [Flink Connector](#)：包括 Flink Consumer & Producer。

云产品消费

- 使用 [CloudMonitor 云监控消费](#)：监控场景。
- 使用 [ARMS消费](#)：业务实时监控场景。
- 使用 [StreamCompute 消费日志](#)：自定义监控场景。
- 使用 [FunctionCompute 消费日志](#)
- 使用 E-MapReduce 消费日志：请参见 [Storm](#)，[Spark Streaming](#)。
- 使用 [命令行工具CLI](#)消费日志。

开源产品消费

[#unique_14](#)：使用 Flume 消费日志，并将日志导入到 HDFS 实例。

查询分析

参见[实时查询分析简介](#)：

- 使用日志服务控制台查询日志：参见[实时查询分析简介](#)。
- 使用日志服务 SDK/API 查询日志：日志服务提供 REST 风格的 API，基于 HTTP 协议实现。日志服务的 API 同样提供全功能的日志查询接口。详细内容请参考[日志服务 API](#)。

投递存储

- [投递日志到OSS](#)：长时间存储或使用 E-MapReduce 分析日志。
- [投递至表格存储（Table Store）](#)：使用表格存储（NoSQL）存储日志。
- [#unique_18](#)：使用 MaxCompute 分析日志。
- [使用函数计算进行自定义投递](#)。
- [#unique_20](#)：将日志数据通过 DataWorks 的数据集成（Data Integration）功能投递至 MaxCompute，已进行大数据分析。

其他

安全日志服务：日志服务与安全云产品对接，可通过 ISV 消费云产品日志。

2 普通消费

日志服务提供多语言（Java、Python、Go等）的 SDK，且这些语言的 SDK 都支持日志消费接口。

SDK消费

以 [Java SDK](#) 为例，以下代码展示如何消费 ShardId 中的数据。关于 SDK 的更多信息请参考[日志服务 SDK](#)。

```
Client client = new Client(host, accessId, accessKey);

String cursor = client.GetCursor(project, logStore, shardId,
CursorMode.END).GetCursor();
System.out.println("cursor = " + cursor);
try {
    while (true) {
        PullLogsRequest request = new PullLogsRequest(project,
logStore, shardId, 1000, cursor);
        PullLogsResponse response = client.pullLogs(request);
        System.out.println(response.getCount());
        System.out.println("cursor = " + cursor + " next_cursor = " +
response.getNextCursor());
        if (cursor.equals(response.getNextCursor())) {
            break;
        }
        cursor = response.getNextCursor();
        Thread.sleep(200);
    }
} catch (LogException e) {
    System.out.println(e.GetRequestId() + e.GetErrorMessage());
}
```

控制台预览

日志预览也是一种日志消费。日志服务控制台提供专门的预览页面帮助您在浏览器内直接预览日志库中的部分日志。

1. 登录[日志服务控制台](#)，单击Project名称。
2. 在日志库列表中，单击日志库名称后的图标，然后选择消费预览。

3. 在消费预览页面，指定预览的Shard，时间段，然后单击预览。

预览页面向您展示指定时间区间开始的10个数据包的日志数据。

消费预览

config-operation-log

Shard : 0

15分钟

预览

日志预览仅供调试日志数据是否上传成功，如果需要通过关键词查询请创建日志索引

时间/来源	内容
2019-07-29 08:47:25 192.168.0.148	message: 2019-07-29 00:47:25 [INF] [aliyunlog_timer_syner.go:27] [run] flush all start: time: 2019-07-29T00:47:25.418903687Z level:info version: 0.1.0
2019-07-29 08:53:00 192.168.0.148	message: 2019-07-29 00:53:00 [INF] [aliyunlog_timer_syner.go:27] [run] flush all start: time: 2019-07-29T00:53:00.819072168Z level:info version: 0.1.0
2019-07-29 08:58:21 192.168.0.148	message: 2019-07-29 00:58:21 [INF] [aliyunlog_timer_syner.go:27] [run] flush all start: time: 2019-07-29T00:58:21.819244171Z level:info version: 0.1.0

3 消费组消费

3.1 通过消费组消费日志

与使用SDK消费数据相比，使用消费组（ConsumerGroup）消费日志数据的优点在于，用户无需关心日志服务的实现细节和消费者之间的负载均衡、failover等，只需要专注于业务逻辑即可。

基本概念

使用消费组消费日志数据之前，请阅读以下日志服务消费组基本概念，包括消费组（ConsumerGroup）和消费者（Consumer）。

概念	说明
消费组	一个消费组由多个消费者构成，同一个消费组下面的消费者共同消费一个Logstore中的数据，消费者之间不会重复消费数据。
消费者	消费组的构成单元，实际承担消费任务，同一个消费组下面的消费者名称必须不同。

在日志服务中，一个Logstore下面会有多个Shard，协同消费库的功能就是将Shard分配给一个消费组下面的消费者，分配方式遵循以下原则：

- 每个Shard只会分配到一个消费者。
- 一个消费者可以同时拥有多个Shard。

新的消费者加入一个消费组，这个消费组下面的Shard从属关系会调整，以达到消费负载均衡的目的，但是仍遵循分配原则，分配过程对用户透明。

协同消费库的另一个功能是保存Checkpoint，方便程序故障恢复时能接着从断点继续消费，从而保证数据不会被重复消费。

操作步骤

日志服务消费组通过JAVA语言和Python语言实现，本文档以JAVA为例。更多内容，请参考GitHub: [JAVA](#)、[Python](#)、[Golang](#)。

1. 添加maven 依赖。

```
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.5.0</version>
</dependency>
<dependency>
  <groupId>com.aliyun.openservices</groupId>
```

```
<artifactId>loghub-client-lib</artifactId>
<version>0.6.16</version>
</dependency>
```

2. 创建main .java文件。

```
import com.aliyun.openservices.loghub.client.ClientWorker;
import com.aliyun.openservices.loghub.client.config.LogHubConfig;
import com.aliyun.openservices.loghub.client.exceptions.LogHubClientWorkerException;

public class Main {
    // 日志服务域名, 根据实际情况填写
    private static String sEndpoint = "cn-hangzhou.log.aliyuncs.com";

    // 日志服务项目名称, 根据实际情况填写
    private static String sProject = "ali-cn-hangzhou-sls-admin";
    // 日志库名称, 根据实际情况填写
    private static String sLogstore = "sls_operation_log";
    // 消费组名称, 根据实际情况填写
    private static String sConsumerGroup = "consumerGroupX";
    // 消费数据的ak, 根据实际情况填写
    private static String sAccessKeyId = "";
    private static String sAccessKey = "";

    public static void main(String[] args) throws LogHubClientWorkerException, InterruptedException {
        // 第二个参数是消费者名称, 同一个消费组下面的消费者名称必须不同, 可以使用相同的消费组名称, 不同的消费者名称在多台机器上启动多个进程, 来均衡消费一个Logstore, 这个时候消费者名称可以使用机器ip来区分。第9个参数 (maxFetchLogGroupSize) 是每次从服务端获取的LogGroup数目, 使用默认值即可, 如有调整请注意取值范围(0,1000]
        LogHubConfig config = new LogHubConfig(sConsumerGroup, "consumer_1", sEndpoint, sProject, sLogstore, sAccessKeyId, sAccessKey, LogHubConfig.ConsumePosition.BEGIN_CURSOR);
        ClientWorker worker = new ClientWorker(new SampleLogHubProcessorFactory(), config);
        Thread thread = new Thread(worker);
        //Thread运行之后, Client Worker会自动运行, ClientWorker扩展了Runnable接口。
        thread.start();
        Thread.sleep(60 * 60 * 1000);
        //调用worker的Shutdown函数, 退出消费实例, 关联的线程也会自动停止。
        worker.shutdown();
        //ClientWorker运行过程中会生成多个异步的Task, Shutdown之后最好等待还在执行的Task安全退出, 建议sleep 30s。
        Thread.sleep(30 * 1000);
    }
}
```

3. 创建SampleLogHubProcessor.java文件。

```
import com.aliyun.openservices.log.common.FastLog;
import com.aliyun.openservices.log.common.FastLogContent;
import com.aliyun.openservices.log.common.FastLogGroup;
import com.aliyun.openservices.log.common.FastLogTag;
import com.aliyun.openservices.log.common.LogGroupData;
import com.aliyun.openservices.loghub.client.ILogHubCheckPointTracker;
import com.aliyun.openservices.loghub.client.exceptions.LogHubCheckPointException;
import com.aliyun.openservices.loghub.client.interfaces.ILogHubProcessor;
```

```

import com.aliyun.openservices.loghub.client.interfaces.ILogHubProcessorFactory;

import java.util.List;

public class SampleLogHubProcessor implements ILogHubProcessor {
    private int shardId;
    // 记录上次持久化 checkpoint 的时间
    private long mLastCheckTime = 0;

    public void initialize(int shardId) {
        this.shardId = shardId;
    }

    // 消费数据的主逻辑，这里面的所有异常都需要捕获，不能抛出去。
    public String process(List<LogGroupData> logGroups,
        ILogHubCheckPointTracker checkPointTracker
    ) {
        // 这里简单的将获取到的数据打印出来
        for (LogGroupData logGroup : logGroups) {
            FastLogGroup flg = logGroup.GetFastLogGroup();
            System.out.println(String.format("\tcategory\t:\t%s\n\tsource\t:\t%s\n\ttopic\t:\t%s\n\tmachineUUID\t:\t%s",
                flg.getCategory(), flg.getSource(), flg.getTopic(), flg.getMachineUUID()));
            System.out.println("Tags");
            for (int tagIdx = 0; tagIdx < flg.getLogTagsCount(); ++tagIdx) {
                FastLogTag logtag = flg.getLogTags(tagIdx);
                System.out.println(String.format("\t%s\t:\t%s", logtag.getKey(), logtag.getValue()));
            }
            for (int lIdx = 0; lIdx < flg.getLogCount(); ++lIdx) {
                FastLog log = flg.getLog(lIdx);
                System.out.println("-----\nLog: " + lIdx + ",
time: " + log.getTime() + ", GetContentCount: " + log.getContentCount());
                for (int cIdx = 0; cIdx < log.getContentsCount(); ++cIdx) {
                    FastLogContent content = log.getContents(cIdx);
                    System.out.println(content.getKey() + "\t:\t" + content.getValue());
                }
            }
            long curTime = System.currentTimeMillis();
            // 每隔 30 秒，写一次 checkpoint 到服务端，如果 30 秒内，worker
            // 新启动的 worker 会从上一个 checkpoint 取消费数据，有可能有少量的
            // 重复数据
            if (curTime - mLastCheckTime > 30 * 1000) {
                try {
                    //参数true表示立即将checkpoint更新到服务端，为false会将
                    //checkpoint缓存在本地，后台默认隔60s会将checkpoint刷新到服务端。
                    checkPointTracker.saveCheckPoint(true);
                } catch (LogHubCheckPointException e) {
                    e.printStackTrace();
                }
                mLastCheckTime = curTime;
            }
            return null;
        }

        // 当 worker 退出的时候，会调用该函数，用户可以在此处做些清理工作。
    }
}

```

```
public void shutdown(ILogHubCheckPointTracker checkPointTracker)
{
    //将消费断点保存到服务端。
    try {
        checkPointTracker.saveCheckPoint(true);
    } catch (LogHubCheckPointException e) {
        e.printStackTrace();
    }
}

class SampleLogHubProcessorFactory implements ILogHubProcessorFactory {
    public ILogHubProcessor generatorProcessor() {
        // 生成一个消费实例
        return new SampleLogHubProcessor();
    }
}
```

**说明:**

运行以上代码，可以将一个Logstore下的所有数据打印出来，如果需要多个消费者共同消费一个Logstore，可以按程序注释修改程序，用同样的消费组名称加不同的消费者名称，启动另外的消费进程。

限制与异常诊断

每个Logstore创建消费组个数的上限为10。超出时将报错ConsumerGroupQuotaExceed。

建议为消费者程序配置log4j，将消费组内部遇到的错误信息打印出来，方便定位异常。例如，放置log4j.properties文件到resources目录下执行程序可以看到类似如下异常：

```
[WARN ] 2018-03-14 12:01:52,747 method:com.aliyun.openservices.loghub.
client.LogHubConsumer.sampleLogError(LogHubConsumer.java:159)
com.aliyun.openservices.log.exception.LogException: Invalid loggroup
count, (0,1000]
```

以下配置为log4j.properties典型配置：

```
log4j.rootLogger = info,stdout
log4j.appender.stdout = org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target = System.out
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd
HH:mm:ss,SSS} method:%l%n%m%n
```

高阶操作

普通用户使用上述程序即可消费数据，高阶用户还可以参考以下内容完成高阶操作。

- 消费某个时间开始的数据

上面代码中的LoghubConfig有两个构造函数：

```
// consumerStartTimeInSeconds参数表示1970之后的秒数，含义是消费这个时间点之后的数据。
public LogHubConfig(String consumerGroupName,
                    String consumerName,
                    String loghubEndPoint,
                    String project, String logStore,
                    String accessId, String accessKey,
                    int consumerStartTimeInSeconds);
// position是个枚举变量，LogHubConfig.ConsumePosition.BEGIN_CURSOR表示从最老的数据开始消费，LogHubConfig.ConsumePosition.END_CURSOR表示从最新的数据开始消费。
public LogHubConfig(String consumerGroupName,
                    String consumerName,
                    String loghubEndPoint,
                    String project, String logStore,
                    String accessId, String accessKey,
                    ConsumePosition position);
```

可以按照消费需求，使用不同的构造方法，但是注意，如果服务端保存有checkpoint，那么开始消费位置以服务端保存的checkpoint为准。

- 重置Checkpoint

在补数据或重复计算等场景中，需要将某个ConsumerGroup点位设置为某一个时间点，使当前消费组能够从新位置开始消费。可通过以下两种方式完成。

- 删除消费组。
 - 停止消费程序，并在控制台删除消费组。
 - 修改代码，使用指定时间点消费，重新启动程序。
- 通过SDK将当前消费组重置到某一个时间点。
 - 停止消费程序。
 - 使用SDK修改位点，重新启动消费程序。

```
public static void updateCheckpoint() throws Exception {
    Client client = new Client(host, accessId, accessKey);
    long timestamp = Timestamp.valueOf("2017-11-15 00:00:00").
getTime() / 1000;
    ListShardResponse response = client.ListShard(new ListShardRequest(project, logStore));
    for (Shard shard : response.GetShards()) {
        int shardId = shard.GetShardId();
        String cursor = client.GetCursor(project, logStore,
shardId, timestamp).GetCursor();
        client.UpdateCheckPoint(project, logStore, consumerGroup
, shardId, cursor);
    }
}
```

```
}

```

状态与报警

- 如果您开启了[服务日志功能](#)，可以：
 - 通过[#unique_26/unique_26_Connect_42_section_qt5_kwt_ngb](#)仪表盘查看消费组状态。
 - 对[#unique_26/unique_26_Connect_42_section_qt5_kwt_ngb](#)中的图表[#unique_27](#)任务。
- 如果您未开启服务日志功能，可以：
 - [在控制台查看消费组状态](#)
 - [通过云监控查看消费组延迟，并配置报警](#)

使用RAM子账号进行访问

子账号访问需要设置消费组相关的RAM权限，设置方法请参考[RAM文档](#)。

可以设置的相关Action如下：

Action	Resource
log:GetCursorOrData	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}
log:CreateConsumerGroup	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/*
log:ListConsumerGroup	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/*
log:ConsumerGroupUpdateCheckPoint	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/\${consumerGroupName}
log:ConsumerGroupHeartBeat	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/\${consumerGroupName}

Action	Resource
log:UpdateConsumerGroup	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/\${consumerGroupName}
log:GetConsumerGroupCheckPoint	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/\${consumerGroupName}

例如，假设Project所在地域为cn-hangzhou，Project名称为project-test，消费的Logstore名称为logstore-test，Project的owner的aliuid为1234567，消费组名称为consumergroup-test，则需要为RAM子账号设置以下权限。

```
{
  "Version": "1",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "log:GetCursorOrData"
      ],
      "Resource": "acs:log:cn-hangzhou:1234567:project/project-test/logstore/logstore-test"
    },
    {
      "Effect": "Allow",
      "Action": [
        "log:CreateConsumerGroup",
        "log:ListConsumerGroup"
      ],
      "Resource": "acs:log:cn-hangzhou:1234567:project/project-test/logstore/logstore-test/consumergroup/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "log:ConsumerGroupUpdateCheckPoint",
        "log:ConsumerGroupHeartBeat",
        "log:UpdateConsumerGroup",
        "log:GetConsumerGroupCheckPoint"
      ],
      "Resource": "acs:log:cn-hangzhou:1234567:project/project-test/logstore/logstore-test/consumergroup/consumergroup-test"
    }
  ]
}
```

```
}
```

3.2 消费组状态

协同消费组（ConsumerGroup） 是实时消费数据高级模式，能够提供多个消费实例对日志库消费自动负载均衡。Spark Streaming、Storm 都以 ConsumerGroup 作为基础模式。

通过控制台查看消费进度

1. 登录[日志服务控制台](#)，单击Project名称。
2. 单击  依次展开日志库名称 > 数据消费节点。
3. 单击指定的ConsumerGroup之后，即可查看每个shard消费数据的进度。

图 3-1: 消费状态

Consumer Group状态 (etl-c7c6d06a08678cd01f148f05a12f48)		
输入shard Q		
shard ↓↑	最近消费数据时间 ↓↑	消费数据客户端
0	2019-06-17 20:47:37	etl-c7c6d06a08678cd01f148f05a12f48-5107715
1	2019-06-17 20:48:09	etl-c7c6d06a08678cd01f148f05a12f48-5107715
2	2019-06-17 20:47:58	etl-c7c6d06a08678cd01f148f05a12f48-5107715
3	2019-06-17 20:48:06	etl-c7c6d06a08678cd01f148f05a12f48-5107715

如上图所示，页面上展示该日志库包含 4 个 Shard，对应 4 个消费者，第二列为每个消费者最近消费的数据时间。通过消费数据时间可以判断出目前数据处理是否能满足数据产生速度，如果已经严重落后于当前时间（即数据消费速率小于数据产生速率），可以考虑增加消费者数目。

通过 API/SDK 查看消费进度

以 Java SDK 作为例子，演示如何通过 API 获得消费状态。

```
package test;
import java.util.ArrayList;
import com.aliyun.openservices.log.Client;
import com.aliyun.openservices.log.common.Consts.CursorMode;
import com.aliyun.openservices.log.common.ConsumerGroup;
import com.aliyun.openservices.log.common.ConsumerGroupShardCheckPoint;
import com.aliyun.openservices.log.exception.LogException;
public class ConsumerGroupTest {
    static String endpoint = "";
    static String project = "";
    static String logstore = "";
    static String accessKeyId = "";
    static String accesskey = "";
```

```

    public static void main(String[] args) throws LogException {
        Client client = new Client(endpoint, accessKeyId, accesskey);
        //获取这个 logstore 下的所有 consumer group, 可能不存在, 此时
        consumerGroups 的长度是 0
        ArrayList<ConsumerGroup> consumerGroups;
        try{
            consumerGroups = client.ListConsumerGroup(project,
logstore).GetConsumerGroups();
        }
        catch(LogException e){
            if(e.GetErrorCode() == "LogStoreNotExist")
                System.out.println("this logstore does not have any
consumer group");
            else{
                //internal server error branch
            }
            return;
        }
        for(ConsumerGroup c: consumerGroups){
            //打印 consumer group 的属性, 包括名称、心跳超时时间、是否按序消费
            System.out.println("名称: " + c.getConsumerGroupName());
            System.out.println("心跳超时时间: " + c.getTimeout());
            System.out.println("按序消费: " + c.isInOrder());
            for(ConsumerGroupShardCheckPoint cp: client.GetCheckPoint(
project, logstore, c.getConsumerGroupName()).GetCheckPoints()){
                System.out.println("shard: " + cp.getShard());
                //请格式化下, 这个时间返回精确到毫秒的时间, 长整型
                System.out.println("最后一次消费数据的时间: " + cp.
getUpdateTime());
                System.out.println("消费者名称: " + cp.getConsumer());
                String consumerPrg = "";
                if(cp.getCheckPoint().isEmpty())
                    consumerPrg = "尚未开始消费";
                else{
                    //unix 时间戳, 单位是秒, 输出的时候请注意格式化
                    try{
                        int prg = client.GetPrevCursorTime(project,
logstore, cp.getShard(), cp.getCheckPoint()).GetCursorTime();
                        consumerPrg = "" + prg;
                    }
                    catch(LogException e){
                        if(e.GetErrorCode() == "InvalidCursor")
                            consumerPrg = "非法, 前一次消费时刻已经超出了
logstore中数据的生命周期";
                        else{
                            //internal server error
                            throw e;
                        }
                    }
                }
                System.out.println("消费进度: " + consumerPrg);
                String endCursor = client.GetCursor(project, logstore
, cp.getShard(), CursorMode.END).GetCursor();
                int endPrg = 0;
                try{
                    endPrg = client.GetPrevCursorTime(project,
logstore, cp.getShard(), endCursor).GetCursorTime();
                }
                catch(LogException e){
                    //do nothing
                }
                //unix 时间戳, 单位是秒, 输出的时候请注意格式化
                System.out.println("最后一条数据到达时刻: " + endPrg);
            }
        }
    }

```

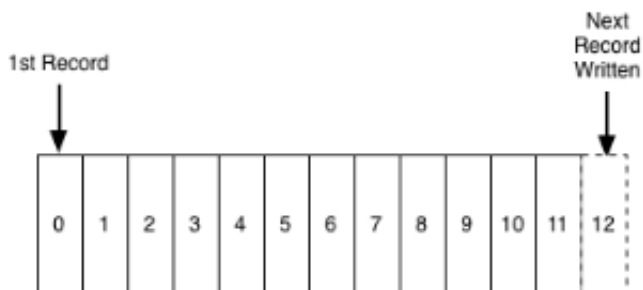
```
}  
}  
}
```

3.3 消费组监控与报警

ConsumerGroup是一个消费者组，包含多个Consumer，每个Consumer消费Logstore中的一部分Shard。

Shard的数据模型可以简单理解成一个队列，新写入的数据被加到队尾，队列中的每条数据都会对应一个数据写入时间，下图是Shard的数据模型。

图 3-2: Shard数据模型



协同消费延迟报警中的基本概念：

- 消费过程：消费者从队头开始顺序读取数据的过程。
- 消费进度：消费者当前读取的数据对应的写入时间。
- 消费落后时长：当前消费进度和队列中最新的数据写入时间的差值，单位为秒。

ConsumerGroup的消费落后时长取其包含的所有Shard的消费落后时长的最大值，当超过用户预设阈值时，就认为消费落后太多，触发报警。

操作步骤

1. 登录日志服务控制台，单击需要监控的Logstore后的图标，选择监控进入云监控控制台。

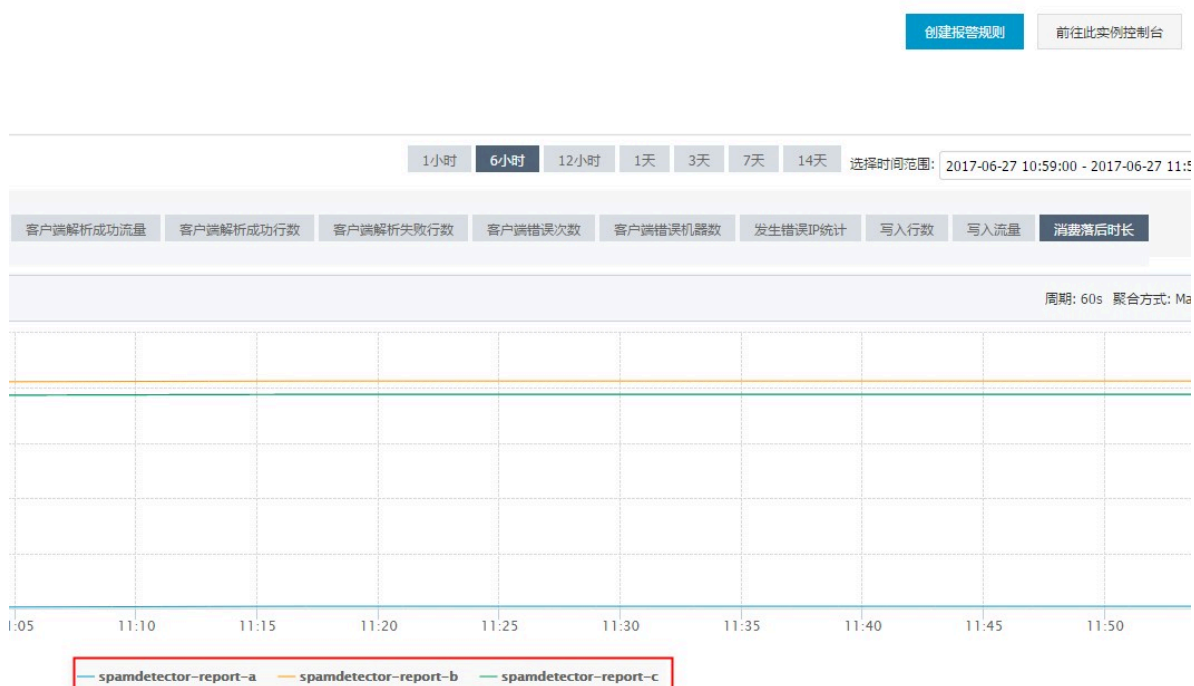
2. 单击消费落后时长图表，查看消费落后时长。

图 3-3: 消费落后时长图表



下图展示了Logstore下所有ConsumerGroup的消费落后时长，单位为秒。框中图例是所有的ConsumerGroup，

图 3-3: 查看消费落后时长



3. 单击右上角创建报警规则进入规则创建页面。
4. 创建针对 ConsumerGroup spamdetector-report-c 的报警规则，5分钟内只要有一次延迟大于等于600秒就会报警。设置生效时间和报警通知联系人，保存规则。

图 3-5: 设置报警规则

2

设置报警规则

规则名称:

xxxxxx

规则描述:

消费落后时长

5分钟周期

连续1周期

只要有一次

>=

600

Second

consumerGroup:

任意consumerGroup

All

+添加报警规则

通道沉默周期:

24 小时

生效时间:

00:00

至

23:59

3

通知方式

通知对象:

联系人通知组

搜索

快速创建联系人组

已选组 1 个

云账号报警联系人

报警级别:

电话+短信+邮件+钉钉机器人

短信+邮件+钉钉机器人

邮件+钉钉机器人

☐ 弹性伸缩 (选择伸缩规则后, 会在报警发生时触发相应的伸缩规则)

邮件主题:

邮件主题默认为产品名称+监控项名称+实例ID

上面的操作完成后便成功创建了报警规则。有关报警规则配置的任何问题，请提工单到云监控。

4 函数计算消费

4.1 开发指南

日志服务[自定义ETL功能](#)的数据消费端运行在阿里云函数计算服务上，根据不同的ETL用途，您可以选择使用[日志服务提供的函数模板](#)，或是自定义实现个性化函数。

本文介绍如何实现一个自定义日志服务ETL函数。

函数Event

函数Event即运行函数的输入参数，格式为一个JSON Object序列化后字符串。

字段说明

- **jobName**字段

日志服务ETL Job名字，函数计算服务上的日志服务触发器对应一个日志服务的ETL Job。

- **taskId**字段

对于一个ETL Job，taskId是某一次确定性的函数调用的标识。

- **cursorTime**字段

本次函数调用包括的数据中，最后一条日志到达日志服务服务端的unix_timestamp。

- **source**字段

该字段由日志服务生成，日志服务根据ETL Job定义的任务间隔定时触发函数执行，source字段是函数Event中很重要的一部分，定义了本次函数调用应该消费哪些数据。

该数据源范围由如下字段（相关字段定义请参考[日志服务概念介绍](#)）组成：

字段	说明
endpoint	Project所属日志服务Region的 #unique_36 。
projectName	Project名称。
logstoreName	Logstore名称。
shardId	Logstore下的某一个确定Shard。
beginCursor	需要从Shard的什么位置开始消费数据。

字段	说明
endCursor	需要消费Shard数据到什么位置。



说明:

Shard对应的[beginCursor, endCursor)是一个左闭右开区间。

· parameter字段

JSON Object类型，由用户在创建ETL Job（函数计算的LOG触发器）时设定。自定义函数运行时解析该字段，可以获取到函数所需要的运行参数。

在函数计算控制台创建日志服务触发器时，通过填写函数配置进行设置，如下图：

图 4-1: 函数配置

触发器类型

日志服务触发器

[帮助文档](#) [ETL函数开发指南](#)

* 触发器名称

logstore-replication-job

1. 只能包含字母、数字、下划线和中划线

2. 不能以数字、中划线开头

3. 长度限制在1-128之间

* 日志项目名称

wdproject

?

* 日志仓库名称

wdproject

?

* 触发器日志

internal-alert-history

?

* 触发间隔

60

秒

注意事项:

» 1. 取值范围：[3,600]，单位：秒

» 2. 该参数定义日志服务触发函数执行的间隔，例如每60秒将logstore的每个shard最近60秒数据位置读出作为函数event调用函数执行，在计算。

» 3. 如果logstore的shard流量较大（超过1MB/s或者更高），建议缩短函数的触发间隔，使得每次函数运行所处理的数据量是合理大小。

* 重试次数

3

次

注意事项:

» 1. 取值范围：[0,100]

» 2. 日志服务根据触发间隔每次触发函数执行时，如果遇到错误（例如权限不足、网络失败、函数执行异常返回等），该参数定义本次触发

» 3. 对于本次触发，如果超过最大重试次数仍无法成功的。需要等到下一次触发间隔到来时，由日志服务再次触发函数执行。

* 函数配置

1

2

3

4

5

```
{
  "source": {
    "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
    "projectName": "fc-*****",
    "logstoreName": "demo",
```

函数Event示例

```
{
  "source": {
    "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
    "projectName": "fc-*****",
    "logstoreName": "demo",
```

```
    "shardId": 0,
    "beginCursor": "MTUwNTM5MDI3NTY1ODcwNzU2Ng==",
    "endCursor": "MTUwNTM5MDI3NTY1ODcwNzU2OA=="
  },
  "parameter": {
    ...
  },
  "jobName": "fedad35f51a2a97b466da57fd71f315f539d2234",
  "taskId": "9bc06c96-e364-4f41-85eb-b6e579214ae4",
  "cursorTime": 1511429883
}
```

在函数调试时候，你也可以根据[#unique_37](#)接口获取cursor并按上述格式自行组装一个函数Event用于测试。

函数开发

您可以使用Java/Python/Node.js等多种语言实现函数功能，日志服务提供了相应runtime的[各种语言SDK](#)以便您在函数中进行集成。

本节以Java8 runtime为例，介绍如何开发日志服务ETL函数。由于涉及Java8函数编程细节，请先阅读[函数计算服务Java编程指南](#)。

Java函数模板

目前，日志服务提供了基于Java8执行环境的[自定义ETL函数模板](#)，您可以在这基础上完成自定义需求的实现。

模板已实现以下功能：

- 函数Event中source、taskId、jobName字段的解析。
- 根据source中定义的数据源，通过[日志服务Java SDK](#)拉取数据，并对每一批数据调用processData接口进行处理。

在模板中，您还需要实现以下功能：

- 函数Event中parameter字段的解析，`UserDefinedFunctionParameter.java`实现。
- 函数内针对数据的自定义业务逻辑，`UserDefinedFunction.java`的processData接口实现。
- 为您的函数取一个可以描述功能的名字，替换`UserDefinedFunction`。

processData方法实现

您需要在processData内完成对一批数据的消费、加工、投递，视具体需求而定。

可以参考[LogstoreReplication](#)，它实现了将数据从一个Logstore中读出后写到另一个日志服务Logstore。

note



说明:

1. processData处理数据成功请返回true，处理数据遇到异常无法重试成功的请返回false，但此时函数还会继续运行下去，且日志服务判定这是一次成功的ETL任务，只是忽略了一些处理异常的数据。
2. 如果遇到致命错误或业务逻辑上认为有异常需要提前终止函数执行，请通过throw Exception方式跳出函数运行，日志服务可以据此判断函数运行异常，并会按照ETL Job设置的规则重新调用函数执行。

注意事项

- 如果Shard流量较大，请注意为函数配置足够的内存规格，以避免函数OOM导致异常终止。
- 如果函数内执行耗时操作，或者Shard流量较大，请注意设置较短的函数触发间隔和较长的函数运行超时时间。
- 请为函数服务配置足够的权限，例如在函数内写OSS就需要配置OSS写权限。

ETL日志

- ETL调度日志

调度日志只记录ETL任务开始的时间、结束时间，任务是否成功以及成功返回的信息。如果ETL任务出错了，不仅要生成ETL出错日志，而且要向系统管理员发送报警邮件或短信。在创建触发器时设置触发器日志Logstore，并为该Logstore开通索引查询功能。

对于函数执行的统计，可以通过函数进行返回，例如Java8函数的outputStream。日志服务默认提供的模板会写出一个JSON Object序列化后的字符串，该字符串将一并记录在ETL任务调度日志中，方便您进行统计、查询。

- ETL过程日志

这一部分日志是在ETL执行过程中每执行一步记录的关键点和错误，包括某一步骤的开始、结束时间、初始化动作完成情况，模块出错信息等。ETL过程日志的意义是随时可以感知ETL运行情况，如果发生错误，可以及时通过过程日志查找原因。

您可以通过context.getLogger()记录过程日志到日志服务指定的Project、Logstore，建议为该Logstore开启索引查询功能。

4.2 配置函数计算消费日志

依托于阿里云函数计算服务，日志服务提供流式的全托管数据加工服务。

您可以配置一个ETL Job，日志服务将定时获取数据更新并触发函数执行：增量消费日志服务Logstore的数据，在函数里完成自定义加工任务。用于数据加工的函数可以是日志服务提供的模板或者用户自定义函数。

适用场景

数据清洗、加工场景

通过日志服务，快速完成日志采集、加工、查询、分析。

图 4-2: 数据清洗、加工场景



数据投递场景

为数据的目的端落地提供支撑，构建云上大数据产品间的数据管道。

图 4-3: 数据投递场景



工作原理

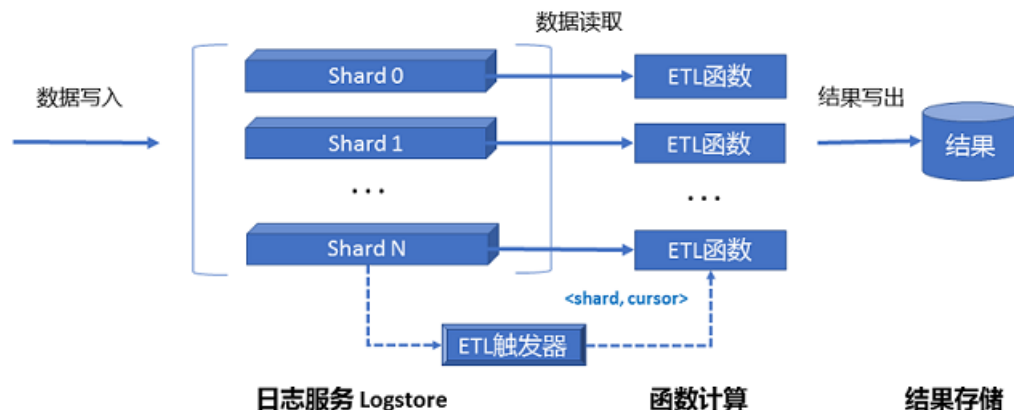
触发机制

日志服务的ETL Job对应于函数计算的一个触发器，创建ETL Job后，日志服务将根据该Job配置启动定时器，定时器轮询Logstore的shard信息，当发现有新日志写入时，生成的< shard_id, begin_cursor, end_cursor >三元组信息作为函数event，触发函数执行。

日志服务的ETL任务触发机制是时间触发，例如：ETL Job触发间隔为60秒，Logstore的Shard 0一直有数据写入，那么对于Shard 0每1分钟就会触发一次函数执行（如果Shard没有新的数

据写入则不会触发函数执行），函数执行的输入为最近60秒的cursor区间。在函数内，可以根据cursor读取Shard 0数据进行处理。

图 4-4: 触发机制



ETL函数

您可以选择使用函数模板或者自定义函数，开始前建议先了解函数计算服务的基本概念。

- 日志服务维护的函数模板

函数模板在GitHub上维护，请单击[aliyun-log-fc-functions](#)访问。

- 用户自定义函数

由您自行实现，函数配置的格式与函数的具体实现有关，请参考[ETL函数开发指南](#)。

使用指南

步骤1 授权与准备资源

1. 在[快捷授权](#)页面单击同意授权，为日志服务触发函数执行授权。
2. 创建日志服务Project和函数过程日志Logstore。

如您之前从未创建过Project和Logstore，请参考[#unique_41](#)创建。



说明:

日志服务Project和函数计算Service所属区域必须相同。

步骤2 创建服务

1. 在函数计算控制台右上角单击新建服务。

2. 填写服务名称和功能描述，并展开高级配置。

配置项	说明
服务名称	为您新建的函数计算服务命名。命名规则： · 由英文字符（a-z）或（A-Z），数字（0-9），下划线（_）和中划线（-）组成 · 首字母必须为英文字母（a-z）、（A-Z）或下划线（_） · 大小写敏感，且长度为1-128字符
功能描述	新建服务的描述内容。
日志项目	日志服务Project名称。必须与您新建函数计算服务的项目位于同一Region。
日志仓库	日志服务Logstore名称。必须与您新建函数计算服务的项目位于同一Region。
角色创建方式	创建服务角色，根据选择的系统模板创建对应的权限。授予函数计算推送日志到指定的仓库。您可以新建角色或是选择已有角色。如您选择使用已有角色，需要选定一个已经存在的角色。

配置项	说明
系统模板授权	选择系统授权策略。日志服务支持 AliyunLogFullAccess 和 AliyunLogReadOnlyAccess 两个系统授权策略。

图 4-5: 创建服务

新建服务 ×

高级配置

日志项目

etl-test-2

?

日志仓库

etltest1

?

① 选择已经存在的角色，可以授予函数计算操作对应的云资源。
授予函数计算推送日志到指定的仓库(etl-test-2/etltest1)

角色创建方式

选择已有角色

已经存在角色

etl-fuction-log-mode

系统模版授权

AliyunLogFullAccess ×

所选角色对应的权限策略

AliyunLogFullAccess

FC-post-log-to-etl-test-2-etltest1

点击授权

确定

取消

确定系统授权策略之后，单击点击授权。此时会跳出角色快捷创建弹窗，您需要在弹窗中确认角色信息和权限信息，包括策略名称、策略描述和策略详情。如果是新建策略，您还需要确认角色名称和角色描述。在策略详情中，您可以细化授权策略，为该角色自定义一个合适的授权策略。

服务授权成功后，单击确定，进入服务概览界面。

24

文档版本：20190916

步骤3 创建函数和触发器

1. 在服务概览界面，单击新建函数。

选择函数模板。

在函数创建向导中，您可以选择与您业务模型相似的业务模板进行修改并创建函数，也可以选择空白函数模板自定义函数。

- 日志服务模板。logstore_replication和loss-shipper-csv是日志服务提供的业务模板，您可以基于此模板创建函数及触发器。
- 空白模板。空白函数模板会创建一个空白函数，通过引导页面进行触发器配置、函数参数配置和代码开发，完成函数的创建。

2. 填写触发器配置，并单击下一步。

如果您选择了日志服务提供的模板，您可以直接配置触发器；如果您选择了空白模板，请先选择触发器类型，再配置触发器。

请填写触发器名称、Project名称、Logstore名称必选项，配置触发器。一个函数计算的LOG类型触发器对应一个日志服务的ETL Job。

配置项	说明	取值
触发器名称	为您新建的触发器命名。	触发器的名称只能包含字母、数字、下划线和中划线，不能以数字、中划线开头，且长度范围为1~256字节。
Project名称	日志服务Project名称。	必须是已经存在的Project。该Project必须与您的服务处于同一地域。
Logstore名称	日志服务Logstore名称（数据源）。本触发器会定时从该Logstore订阅数据到函数计算进行自定义加工，该参数在ETL Job创建后不允许修改。	必须是已存在的Logstore，且属于Project名称中已配置的Project。
触发器日志	日志服务会定时触发函数计算的函数执行，在触发过程中发生的异常、函数执行统计信息会记录到该Logstore，您可以为这个Logstore创建索引以备查看。	必须是已存在的Logstore，且属于Project名称中已配置的Project。

配置项	说明	取值
触发间隔	日志服务触发函数运行的间隔，定义日志服务触发函数执行的间隔。例如每60秒将logstore的每个shard最近60秒数据位置读出作为函数event调用函数执行，在函数内有用户逻辑读取shard数据做计算。如果logstore的shard流量较大（超过1MB/s或者更高），建议缩短函数的触发间隔，使得每次函数运行所处理的数据量是合理大小。	取值范围为3~600，单位为秒。
重试次数	日志服务根据触发间隔每次触发函数执行时，如果遇到错误（例如权限不足、网络失败、函数执行异常返回等），该参数定义本次触发所允许的最大重试次数。对于本次触发，如果超过最大重试次数仍无法成功的，需要等到下一次触发间隔到来时，由日志服务再次触发函数执行。重试对业务造成的影响，因具体的函数代码实现逻辑而异。	取值范围为0~100，单位为次。

配置项	说明	取值
函数配置	日志服务将该配置内容作为函数event一部分传入函数，如何使用该函数由函数自定义逻辑决定。每一种函数实现所要求的函数配置可能是不同的，绝大部分默认提供的函数模板也需要参考说明填写您的参数。默认不传入任何参数时请填写：{ }。	该配置内容必须是JSON Object格式字符串。

图 4-6: 触发器配置

触发器类型

日志服务触发器

帮助文档 ETL函数开发指南

* 触发器名称

logstore-replication-job

1. 只能包含字母、数字、下划线和中划线
 2. 不能以数字、中划线开头
 3. 长度限制在1-128之间

* 日志项目名称

wdproject

?

* 日志仓库名称

wdproject

?

* 触发器日志

internal-alert-history

?

* 触发间隔

60

秒

注意事项:

- » 1. 取值范围: [3,600], 单位: 秒
- » 2. 该参数定义日志服务触发函数执行的间隔, 例如每60秒将logstore的每个shard最近60秒数据位置读出作为函数event调用函数执行, 在计算。
- » 3. 如果logstore的shard流量较大(超过1MB/s或者更高), 建议缩短函数的触发间隔, 使得每次函数运行所处理的数据量是合理大小。

* 重试次数

3

次

注意事项:

- » 1. 取值范围: [0,100]
- » 2. 日志服务根据触发间隔每次触发函数执行时, 如果遇到错误(例如权限不足、网络失败、函数执行异常返回等), 该参数定义本次触发
- » 3. 对于本次触发, 如果超过最大重试次数仍无法成功的。需要等到下一次触发间隔到来时, 由日志服务再次触发函数执行。

* 函数配置

1

2

3

4

5

```

"source": {
  "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
  "projectName": "fc-*****",
  "logstoreName": "demo",

```



说明:

您已经拥有了让日志服务调用您的函数和读写Logstore的的权限。

3. 填写基础管理配置。

填写函数名称、函数入口等基础管理配置信息，并单击下一步。

4. 填写权限配置。

确认模版授权和触发器角色授权，并单击下一步。

5. 核对您的函数信息和触发器信息，并单击完成。

查看触发器日志

登录日志服务控制台，为Job配置的触发器日志Logstore创建索引，查看任务执行统计结果。

查看函数运行日志

登录日志服务控制台，查看函数执行过程的详细信息，详细信息请参考[日志记录](#)。

常见问题

创建触发器后未触发函数执行

1. 确认已经使用[快捷授权](#)为日志服务触发函数执行授权。
2. 确认Job配置的Logstore是否有数据增量修改，当Shard数据有变化时会触发函数执行。
3. 登录日志服务控制台查看触发器日志、函数运行日志查看是否有异常。

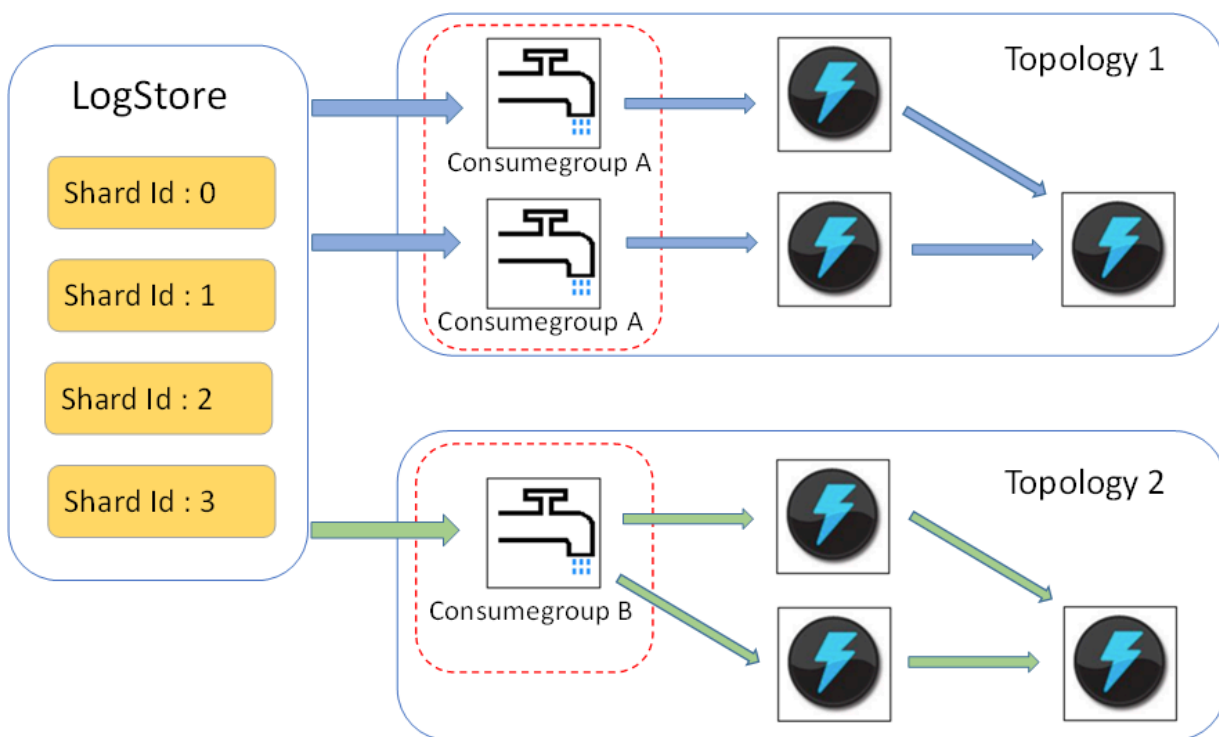
5 Storm消费

日志服务的LogHub提供了高效、可靠的日志通道功能，您可以通过Logtail、SDK等多种方式来实时收集日志数据。收集日志之后，可以通过Spark Stream、Storm 等各实时系统来消费写入到LogHub中的数据。

为了降低Storm用户消费LogHub的代价，日志服务提供了LogHub Storm Spout来实时读取LogHub的数据。

基本结构和流程

图 5-1: 基本结构和流程



- 上图中红色虚线框中就是LogHub Storm Spout，每个Storm Topology会有一组Spout，同组内的Spout共同负责读取Logstore中全部数据。不同Topology中的Spout相互不干扰。
- 每个Topology需要选择唯一的LogHub Consume Group名字来相互标识，同一 Topology内的Spout通过 [Consumer Library](#) 来完成负载均衡和自动failover。
- Spout从LogHub中实时读取数据之后，发送至Topology中的Bolt节点，定期保存消费完成位置作为checkpoint到LogHub服务端。

使用限制

- 为了防止滥用，每个Logstore最多支持 10 个Consumer Group，对于不再使用的Consumer Group，可以使用Java SDK中的DeleteConsumerGroup接口进行删除。

- Spout的个数最好和Shard个数相同，否则可能会导致单个Spout处理数据量过多而处理不过来。
- 如果单个Shard 的数据量太大，超过一个Spout处理极限，则可以使用Shard split接口分裂Shard，来降低每个Shard的数据量。
- 在Loghub Spout中，强制依赖Storm的ACK机制，用于确认Spout将消息正确发送至Bolt，所以在Bolt中一定要调用ACK进行确认。

使用样例

- Spout 使用示例（用于构建 Topology）

```

public static void main( String[] args )
{
    String mode = "Local"; // 使用本地测试模式
    String conumser_group_name = ""; // 每个Topology 需要设定
    唯一的 consumer group 名字，不能为空，支持 [a-z][0-9] 和 '-', '.', 长度在 [3
    -63] 字符，只能以小写字母和数字开头结尾
    String project = ""; // 日志服务的Project
    String logstore = ""; // 日志服务的Logstore
    String endpoint = ""; // 日志服务访问域名
    String access_id = ""; // 用户 ak 信息
    String access_key = "";
    // 构建一个 Loghub Storm Spout 需要使用的配置
    LogHubSpoutConfig config = new LogHubSpoutConfig(conumser_g
    roup_name,
        endpoint, project, logstore, access_id,
        access_key, LogHubCursorPosition.END_CURSOR);
    TopologyBuilder builder = new TopologyBuilder();
    // 构建 loghub storm spout
    LogHubSpout spout = new LogHubSpout(config);
    // 在实际场景中，Spout的个数可以和Logstore Shard 个数相同
    builder.setSpout("spout", spout, 1);
    builder.setBolt("exclaim", new SampleBolt()).shuffleGrouping
    ("spout");
    Config conf = new Config();
    conf.setDebug(false);
    conf.setMaxSpoutPending(1);
    // 如果使用Kryo进行数据的序列化和反序列化，则需要显示设置 LogGroupDa
    ta 的序列化方法 LogGroupDataSerializSerializer
    Config.registerSerialization(conf, LogGroupData.class,
    LogGroupDataSerializSerializer.class);
    if (mode.equals("Local")) {
        logger.info("Local mode...");
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("test-jstorm-spout", conf,
        builder.createTopology());
        try {
            Thread.sleep(6000 * 1000); //waiting for several
            minutes
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        cluster.killTopology("test-jstorm-spout");
        cluster.shutdown();
    } else if (mode.equals("Remote")) {
        logger.info("Remote mode...");
        conf.setNumWorkers(2);
    }
}

```

```

        try {
            StormSubmitter.submitTopology("stt-jstorm-spout-4",
            conf, builder.createTopology());
        } catch (AlreadyAliveException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InvalidTopologyException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    } else {
        logger.error("invalid mode: " + mode);
    }
}
}

```

- 消费数据的 bolt 代码样例，只打印每条日志的内容

```

public class SampleBolt extends BaseRichBolt {
    private static final long serialVersionUID = 4752656887
    774402264L;
    private static final Logger logger = Logger.getLogger(BaseBasicB
    olt.class);
    private OutputCollector mCollector;
    @Override
    public void prepare(@SuppressWarnings("rawtypes") Map stormConf
    , TopologyContext context,
        OutputCollector collector) {
        mCollector = collector;
    }
    @Override
    public void execute(Tuple tuple) {
        String shardId = (String) tuple
            .getValueByField(LogHubSpout.FIELD_SHARD_ID);
        @SuppressWarnings("unchecked")
        List<LogGroupData> logGroupDatas = (ArrayList<LogGroupData
        >) tuple.getValueByField(LogHubSpout.FIELD_LOGGROUPS);
        for (LogGroupData groupData : logGroupDatas) {
            // 每个 logGroup 由一条或多条日志组成
            LogGroup logGroup = groupData.getLogGroup();
            for (Log log : logGroup.getLogsList()) {
                StringBuilder sb = new StringBuilder();
                // 每条日志，有一个时间字段，以及多个 Key:Value 对，
                int log_time = log.getTime();
                sb.append("LogTime:").append(log_time);
                for (Content content : log.getContentsList()) {
                    sb.append("\t").append(content.getKey()).append
                    (":")
                        .append(content.getValue());
                }
                logger.info(sb.toString());
            }
        }
        // 在 loghub spout 中，强制依赖 storm 的 ack 机制，用于确认 spout
        将消息正确
        // 发送至 bolt，所以在 bolt 中一定要调用 ack
        mCollector.ack(tuple);
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        //do nothing
    }
}

```

```
}
```

Maven

storm 1.0 之前版本（如 0.9.6），请使用：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>loghub-storm-spout</artifactId>
  <version>0.6.6</version>
</dependency>
```

storm 1.0 版本及以后，请使用：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>loghub-storm-1.0-spout</artifactId>
  <version>0.1.3</version>
</dependency>
```

6 Flume消费

日志服务LogHub支持通过aliyun-log-flume插件与Flume进行对接，实现日志数据的写入和消费数据。

aliyun-log-flume是一个实现日志服务（Loghub）对接Flume的插件，可以通过Flume将日志服务与其它数据系统如HDFS、Kafka等打通。目前Flume官方支持的插件除了HDFS、Kafka之外，还有Hive、HBase、ElasticSearch等。除此之外，对于常见的数据源在社区也都能找到对应的插件支持。aliyun-log-flume为Loghub提供Sink和Source插件与Flume对接。

- Sink：Flume读取其他数据源的数据然后写入Loghub。
- Source：Flume消费Loghub的数据然后写入其他系统。

Github地址请单击：[Github](#)。

Loghub Sink

通过Sink的方式将其他数据源的数据通过Flume接入Loghub。目前支持两种解析格式：

- SIMPLE：将整个Flume Event作为一个字段写入Loghub。
- DELIMITED：将整个Flume Event作为分隔符分隔的数据，根据配置的列名解析成对应的字段写入Loghub。

支持的配置如下：

配置项	是否必选	说明
type	必选	固定为com.aliyun.loghub.flume.sink.LoghubSink。
endpoint	必选	服务入口。
project	必选	Project名称。
logstore	必选	Logstore名称。
accessKeyId	必选	阿里云访问秘钥accessKeyId。
accessKey	必选	阿里云访问秘钥accessKey。
batchSize	可选	每批次写入Loghub的数据条数。默认1000条。
maxBufferSize	可选	缓存队列的大小。默认1000条。

配置项	是否必选	说明
serializer	可选	Event序列化格式，支持模式如下： · DELIMITED：设置解析格式为分隔符模式。如果设置为该模式，则columns参数为必选。 · SIMPLE：设置解析格式为单行模式。默认为该模式。 · 自定义serializer：设置解析格式为自定义的序列化器模式，设置为该模式时需要填写完整类名称。
columns	可选	serializer为DELIMITED时，必须指定该字段列表，用逗号分隔，顺序与实际数据中的字段顺序一致。
separatorChar	可选	serializer为DELIMITED时，用于指定数据的分隔符，必须为单个字符。默认为,。
quoteChar	可选	serializer为DELIMITED时，用于指定Quote字符。默认为"。
escapeChar	可选	serializer为DELIMITED时，用于指定转义字符。默认为\。
useRecordTime	可选	是否使用数据中的timestamp字段作为日志时间。如果为false则使用当前时间。默认为false。

Loghub Source

通过Source的方式将Loghub的数据通过Flume投递到其他的数据源。目前支持两种输出格式：

- DELIMITED：数据以分隔符日志的形式写入Flume。
- JSON：数据以JSON日志的形式写入Flume。

支持的配置如下：

配置项	是否必选	说明
type	必选	固定为com.aliyun.loghub.flume.source.LoghubSource。
endpoint	必选	服务入口。
project	必选	Project名称。
logstore	必选	Logstore名称。

配置项	是否必选	说明
accessKeyId	必选	阿里云访问秘钥accessKeyId。
accessKey	必选	阿里云访问秘钥accessKey。
heartbeatIntervalMs	可选	客户端和Loghub的心跳间隔，默认30000，单位毫秒。
fetchIntervalMs	可选	Loghub数据拉取间隔，默认100，单位毫秒。
fetchInOrder	可选	是否按顺序消费。默认false。
batchSize	可选	每批次读取的数据条数，默认100。
consumerGroup	可选	读取的消费组名称（随机产生）。
initialPosition	可选	读取数据的起点位置，支持begin, end, timestamp。默认为begin。  说明： 如果服务端已经存在checkpoint，会优先使用服务端的checkpoint。
timestamp	可选	当initialPosition为timestamp时，必须指定时间戳，Unix时间戳格式。
deserializer	必选	Event反序列化格式，支持的模式如下： · DELIMITED：设置解析格式为分隔符模式。默认为该模式。如果设置为该模式，则columns参数为必选。 · JSON：设置解析格式为JSON模式。 · 自定义serializer：设置解析格式为自定义的序列化器模式，设置为该模式时需要填写完整类名称。
columns	可选	deserializer为DELIMITED时，必须指定字段列表，用逗号分隔，顺序与实际数据中的字段顺序一致。
separatorChar	可选	deserializer为DELIMITED时，用于指定数据的分隔符，必须为单个字符。默认为,。
quoteChar	可选	deserializer为DELIMITED时，用于指定Quote字符。默认为"。
escapeChar	可选	deserializer为DELIMITED时，用于指定转义字符。默认为\。

配置项	是否必选	说明
appendTimestamp	可选	deserializer为DELIMITED时，是否将时间戳作为一个字段自动添加到每行末尾。默认为false。
sourceAsField	可选	deserializer为JSON时，是否将日志Source作为一个字段，字段名称为__source__。默认为false。
tagAsField	可选	deserializer为JSON时，是否将日志Tag作为字段，字段名称为__tag__: {tag名称}。默认为false。
timeAsField	可选	deserializer为JSON时，是否将日志时间作为一个字段，字段名称为__time__。默认为false。
useRecordTime	可选	是否使用日志的时间，如果为false则使用当前时间。默认为false。

7 ARMS消费

业务实时监控服务 ARMS 是一款端到端一体化实时监控解决方案的PaaS级阿里云产品，如果您已经开通了阿里云日志服务与ARMS云产品，即可实现ARMS与LogHub无缝对接。

操作步骤：

1. 准备数据源。配置日志服务采集日志，将LogHub作为ARMS的数据源。
2. 为ARMS配置[RAM授权](#)，允许ARMS读取LogHub的日志数据。
3. 配置[ARMS接入LogHub数据源](#)。

完成以上步骤后，ARMS已成功同步您的LogHub日志数据，您可以参考[快速接入自定义监控](#)，为您的日志数据创建ARMS自定义监控任务。

8 开源Flink消费

Flink Log Connector是阿里云日志服务提供的，用于对接Flink的工具。本文档主要介绍如何对接Flink工具对日志数据进行消费。

前提条件

- 已启用Access Key，并创建了Project和Logstore。详细步骤请参见[#unique_46](#)。
- 若您选择使用子账号操作日志服务，请确认已正确设置了Logstore的RAM授权策略。详细内容请参见[#unique_47](#)。

背景信息

Flink Log Connector包括两部分，消费者（Consumer）和生产者（Producer）。

- 消费者用于从日志服务中读取数据，支持exactly once语义，支持Shard负载均衡。
- 生产者用于将数据写入日志服务，使用Connector时，需要在项目中添加maven依赖：

```
<dependency>
  <groupId>com.aliyun.openservices</groupId>
  <artifactId>flink-log-connector</artifactId>
  <version>0.1.13</version>
</dependency>
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>2.5.0</version>
</dependency>
```

Github地址请单击：[Github](#)。

Log Consumer

在Connector中，类Flink Log Consumer提供了订阅日志服务中某一个Logstore的能力，实现了exactly once语义，在使用时，用户无需关心Logstore中Shard数量的变化，Consumer会自动感知。

Flink中每一个子任务负责消费Logstore中部分Shard，如果Logstore中Shard发生split或者merge，子任务消费的Shard也会随之改变。

Flink Log Consumer 会用到的阿里云日志服务接口如下：

- GetCursorOrData

用于从Shard中拉数据，注意频繁的调用该接口可能会导致数据超过日志服务的Shard quota，可以通

过ConfigConstants.LOG_FETCH_DATA_INTERVAL_MILLIS和ConfigConstants.LOG_MAX_NUM制接口调用的时间间隔和每次调用拉取的日志数量，Shard的quota参考文章[#unique_48](#)。

```
configProps.put(ConfigConstants.LOG_FETCH_DATA_INTERVAL_MILLIS, "100");
configProps.put(ConfigConstants.LOG_MAX_NUMBER_PER_FETCH, "100");
```

- ListShards

用于获取Logstore中所有的Shard列表，获取Shard状态等。如果您的Shard经常发生分裂合并，可以通过调整接口的调用周期来及时发现Shard的变化。

```
// 设置每30s调用一次ListShards
configProps.put(ConfigConstants.LOG_SHARDS_DISCOVERY_INTERVAL_MILLIS, "30000");
```

- CreateConsumerGroup

该接口调用只有当设置消费进度监控时才会发生，功能是创建ConsumerGroup，用于同步checkpoint。

- ConsumerGroupUpdateCheckPoint

该接口用户将Flink的snapshot同步到日志服务的ConsumerGroup中。

当RAM子账号使用Flink Log Consumer时，需要对以上几个API进行授权：

接口	资源
GetCursorOrData	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}
ListShards	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}
CreateConsumerGroup	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/*
ConsumerGroupUpdateCheckPoint	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}/consumergroup/\${consumerGroupName}

关于子账号及其授权操作请参见[#unique_49](#)。

1. 配置启动参数。

下面是一个简单的消费示例，我们使用`java.util.Properties`作为配置工具，所有Consumer的配置都可以在`ConfigConstants`中找到。

```
Properties configProps = new Properties();
// 设置访问日志服务的域名
configProps.put(ConfigConstants.LOG_ENDPOINT, "cn-hangzhou.log.aliyuncs.com");
// 设置访问ak
configProps.put(ConfigConstants.LOG_ACCESSKEYID, "");
configProps.put(ConfigConstants.LOG_ACCESSKEY, "");
// 设置日志服务的project
configProps.put(ConfigConstants.LOG_PROJECT, "ali-cn-hangzhou-sls-admin");
// 设置日志服务的Logstore
configProps.put(ConfigConstants.LOG_LOGSTORE, "sls-consumergroup_log");
// 设置消费日志服务起始位置
configProps.put(ConfigConstants.LOG_CONSUMER_BEGIN_POSITION, Consts.LOG_END_CURSOR);
// 设置日志服务的消息反序列化方法
RawLogGroupListDeserializer deserializer = new RawLogGroupListDeserializer();
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<RawLogGroupList> logTestStream = env.addSource(
    new FlinkLogConsumer<RawLogGroupList>(deserializer, configProps));
```



说明:

Flink Stream的子任务数量和日志服务Logstore中的Shard数量是独立的，如果Shard数量多于子任务数量，每个子任务不重复的消费多个Shard，如果少于子任务数量，那么部分子任务就会空闲，等到新的Shard产生。

2. 设置消费起始位置。

Flink Log Consumer支持设置Shard的消费起始位置，通过设置属性`ConfigConstants.LOG_CONSUMER_BEGIN_POSITION`，就可以定制消费从Shard的头

尾或者某个特定时间开始消费，另外，Connector也支持从某个具体的ConsumerGroup中恢复消费。具体取值如下：

- `Consts.LOG_BEGIN_CURSOR`：表示从Shard的头开始消费，也就是从Shard中最旧的数据开始消费。
- `Consts.LOG_END_CURSOR`：表示从Shard的尾开始，也就是从Shard中最新的数据开始消费。
- `Consts.LOG_FROM_CHECKPOINT`：表示从某个特定的ConsumerGroup中保存的Checkpoint开始消费，通过`ConfigConstants.LOG_CONSUMERGROUP`指定具体的ConsumerGroup。
- `UnixTimestamp`：一个整型数值的字符串，用1970-01-01到现在的秒数表示，含义是消费Shard中这个时间点之后的数据。

四种取值示例如下：

```
configProps.put(ConfigConstants.LOG_CONSUMER_BEGIN_POSITION, Consts.LOG_BEGIN_CURSOR);
configProps.put(ConfigConstants.LOG_CONSUMER_BEGIN_POSITION, Consts.LOG_END_CURSOR);
configProps.put(ConfigConstants.LOG_CONSUMER_BEGIN_POSITION, "1512439000");
configProps.put(ConfigConstants.LOG_CONSUMER_BEGIN_POSITION, Consts.LOG_FROM_CHECKPOINT);
```



说明：

如果在启动Flink任务时，设置了从Flink自身的StateBackend中恢复，那么Connector会忽略上面的配置，使用StateBackend中保存的Checkpoint。

3. （可选）设置消费进度监控。

Flink Log Consumer支持设置消费进度监控，所谓消费进度就是获取每一个Shard实时的消费位置，这个位置使用时间戳表示，详细概念可以参考文档[消费组状态](#)，[消费组监控与报警](#)。

```
configProps.put(ConfigConstants.LOG_CONSUMERGROUP, "your consumer group name");
```



说明：

该设置为可选配置项，如果设置，Consumer会首先创建ConsumerGroup，如果ConsumerGroup已经存在，则不执行任何操作，Consumer中的snapshot会自动同步到日志服务的ConsumerGroup中，用户可以在日志服务的控制台查看Consumer的消费进度。

4. 设置容灾和exactly once语义支持。

当打开Flink的Checkpointing功能时，Flink Log Consumer会周期性的将每个Shard的消费进度保存起来，当作业失败时，Flink会恢复Log Consumer，并从保存的最新的Checkpoint开始消费。

写Checkpoint的周期定义了当发生失败时，最多多少的数据会被回溯，即重新消费，使用代码如下：

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.  
getExecutionEnvironment();  
// 开启flink exactly once语义  
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.  
EXACTLY_ONCE);  
// 每5s保存一次checkpoint  
env.enableCheckpointing(5000);
```

更多Flink Checkpoint的细节请参考Flink官方文档[Checkpoints](#)。

Log Producer

Flink Log Producer 用于将数据写到阿里云日志服务中。



说明：

Producer只支持Flink at-least-once语义，在发生作业失败的情况下，写入日志服务中的数据有可能会重复，但是不会丢失。

Producer会用到的日志服务API接口如下：

- PostLogStoreLogs
- ListShards

当RAM子账号使用Producer时，需要对以上两个API进行授权：

接口	资源
PostLogStoreLogs	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}
ListShards	acs:log:\${regionName}:\${projectOwnerAliUid}:project/\${projectName}/logstore/\${logstoreName}

关于子账号及其授权操作请参见[#unique_49](#)。


```

class SimpleLogSerializer implements LogSerializationSchema<String>
{
    public RawLogGroup serialize(String element) {
        RawLogGroup rlg = new RawLogGroup();
        RawLog rl = new RawLog();
        rl.setTime((int)(System.currentTimeMillis() / 1000));
        rl.addContent("message", element);
        rlg.addLog(rl);
        return rlg;
    }
}

public class ProducerSample {
    public static String sEndpoint = "cn-hangzhou.log.aliyuncs.com";
    public static String sAccessKeyId = "";
    public static String sAccessKey = "";
    public static String sProject = "ali-cn-hangzhou-sls-admin";
    public static String sLogstore = "test-flink-producer";
    private static final Logger LOG = LoggerFactory.getLogger(
ConsumerSample.class);
    public static void main(String[] args) throws Exception {
        final ParameterTool params = ParameterTool.fromArgs(args);
        final StreamExecutionEnvironment env = StreamExecutionEnvir
onment.getExecutionEnvironment();
        env.getConfig().setGlobalJobParameters(params);
        env.setParallelism(3);
        DataStream<String> simpleStringStream = env.addSource(new
EventsGenerator());
        Properties configProps = new Properties();
        // 设置访问日志服务的域名
        configProps.put(ConfigConstants.LOG_ENDPOINT, sEndpoint);
        // 设置访问日志服务的ak
        configProps.put(ConfigConstants.LOG_ACCESSKEYID,
sAccessKeyId);
        configProps.put(ConfigConstants.LOG_ACCESSKEY, sAccessKey);
        // 设置日志写入的日志服务project
        configProps.put(ConfigConstants.LOG_PROJECT, sProject);
        // 设置日志写入的日志服务Logstore
        configProps.put(ConfigConstants.LOG_LOGSTORE, sLogstore);
        FlinkLogProducer<String> logProducer = new FlinkLogProducer<
String>(new SimpleLogSerializer(), configProps);
        simpleStringStream.addSink(logProducer);
        env.execute("flink log producer");
    }
    // 模拟产生日志
    public static class EventsGenerator implements SourceFunction<
String> {
        private boolean running = true;
        @Override
        public void run(SourceContext<String> ctx) throws Exception
        {
            long seq = 0;
            while (running) {
                Thread.sleep(10);
                ctx.collect((seq++) + "-" + RandomStringUtils.
randomAlphabetic(12));
            }
        }
        @Override
        public void cancel() {
            running = false;
        }
    }
}

```

9 Logstash消费

日志服务除了支持各种语言、流计算消费方式外，还支持通过Logstash消费日志数据，您可以直接配置日志服务的[Logstash输入源插件](#)来获取日志服务中数据，写入到其他系统中，例如Kafka、HDFS等。

功能特性

- 支持分布式协同消费：可配置多台服务器同时消费某一Logstore。
- 高性能：基于Java ConsumerGroup实现，单核消费速度可达20MB/s。
- 高可靠：消费进度保存到服务端，异常恢复后会从上一次消费的checkpoint处自动恢复消费。
- 自动负载均衡：根据消费者数量自动分配Shard，消费者增加或退出后会自动Rebalance。

10 Spark Streaming消费

E-MapReduce 实现了一套通用的Spark Streaming实时消费LogHub的接口，SDK请转至[GitHub](#)下载。

11 实时计算 (Flink) 消费

阿里云实时计算 (Realtime Compute/Flink) 可以通过创建日志服务源表的方式, 直接消费日志服务中的数据。

日志服务本身是流数据存储, 流计算能将其作为流式数据输入。对于日志服务而言, 每条日志包含一组key-value形式的字段, 假设有如下的日志内容:

```
__source__: 11.85.123.199
__tag__:__receive_time__: 1562125591
__topic__: test-topic
a: 1234
b: 0
c: hello
```

StreamCompute可以定义如下的DDL:

```
create table sls_stream(
  a int,
  b int,
  c varchar
) with (
  type = 'sls',
  endPoint = '<your endpoint>',
  accessId = '<your access key id>',
  accessKey = '<your access key>',
  startTime = '2017-07-05 00:00:00',
  project = 'ali-cloud-streamtest',
  logStore = 'stream-test',
  consumerGroup = 'consumerGroupTest1'
);
```

除日志字段外, 支持提取如下三个属性字段和tag中的自定义字段, 如__receive_time__:

表 11-1: 属性字段

字段名	说明
__source__	日志来源。
__topic__	日志主题。
__timestamp__	日志时间。

属性字段的提取需要添加HEADER声明, 如:

```
create table sls_stream(
  __timestamp__ bigint HEADER,
  __receive_time__ bigint HEADER
  b int,
  c varchar
) with (
  type = 'sls',
```

```

endPoint = '<your endpoint>',
accessId = '<your access key id>',
accessKey = '<your access key>',
startTime = '2017-07-05 00:00:00',
project = 'ali-cloud-streamtest',
logStore = 'stream-test',
consumerGroup = 'consumerGroupTest1'
);

```

WITH参数说明

参数	是否必选	说明
endPoint	是	日志服务endpoint。
accessId	是	访问日志服务的accessKey ID。
accessKey	是	访问日志服务的密钥accessKey。
project	是	日志服务Project名称。
logStore	是	日志服务Logstore名称。
consumerGroup	否	消费组名称。
startTime	否	消费日志开始的时间点。
heartBeatIntervalMills	否	消费客户端的心跳间隔时间。默认为10秒。
maxRetryTimes	否	读取最大尝试次数，默认5次。
batchGetSize	否	单次读取logGroup条数，默认为10条。如果flink的版本是1.4.2+，则默认为100条，最大1000条。
columnErrorDebug	否	是否打开调试开关，如果打开，会把解析异常的日志打印出来。默认为false。

类型映射

日志服务中的字段都是字符串类型，和流计算字段类型对应关系如下，强烈建议用户使用该对应关系进行DDL声明：

日志服务字段类型	流计算字段类型
STRING	VARCHAR

如果使用其他类型也会尝试自动转换，比如“1000”或者“2018-01-12 12:00:00”也可以定义为bigint和timestamp类型。



说明:

- Blink 2.2.0 以下的版本不支持Shard的扩容和缩容，如果分裂或者缩容了某个正在流计算读取的logStore，会导致任务持续出错，不可恢复，该情况下需要重新启动任务来使任务恢复正常。
- 所有blink版本均不支持对正在消费的logStore进行删除 重建。
- 1.6.0及之前版本，在shards数目很多的情况下指定consumerGroup可能会影响读取性能。
- 日志服务暂不支持MAP类型的数据。
- 对于不存在的字段会置为null。
- 字段顺序支持无序（建议字段顺序和表中定义一致）。
- batchGetSize指明的是logGroup获取的数量，如果单条日志的大小和batchGetSize都很大，很有可能会导致频繁的GC。

FAQ

- 如果一个Shard没有新数据写入，会导致job的整体延迟增加，或有些窗口聚合的job无输出，这个要多注意。遇到这种情况，只需要把并发数调整为RW的partition数量即可。
- 并发数建议和Shard个数一致，否则当两个Shard读取速度差异较大时，理论上在追数据场景存在数据被过滤掉的风险。
- 针对__tag__:__hostname__、__tag__:__path__等字段，去掉前面的tag，按照获取属性字段的方式获取即可。



说明:

调试时无法抽取到该类型数据，建议用户用本地DEBUG方法，上线运行print到日志中查看。

12 CloudMonitor消费

[云监控\(Cloud Monitor\)](#)可以直接消费LogHub下Logstore数据，提供监控功能。

主要功能：

- 对日志中的关键字报警。
- 统计单位时间内的QPS、RT。
- 统计单位时间内的PV、UV等。

操作步骤请参见：

- [日志监控概览](#)
- [云监控管理日志监控](#)

13 Golang Consumer Group 消费

Aliyun LOG Go Consumer Library 是用 Go 语言编写的协同消费库，支持多个消费者同时消费 Logstore。使用 Consumer Library 消费数据可以为您带来多种优势，包括：性能优良，资源可控制，自动重连等，用户只需专注自己业务逻辑，无需关心 Shard 分配、CheckPoint、Failover 等事宜。

Github项目地址以及更多详细说明参见：[Golang Consumer Library](#)。

14 最佳实践

14.1 消费-通过函数计算清洗数据

日志服务的LogHub是流式的数据中心，日志写入后可实时消费。日志服务ETL面向的正是这些流式写入的数据，提供秒级实时的ETL作业。

背景信息

ETL（Extract-Transform-Load）用来描述将数据从来源端经过抽取（Extract）、转换（Transform）、加载（Load）至目的端的过程。传统ETL是构建数据仓库的重要一环，用户从数据源抽取所需的数据，经过数据清洗，最终按照预先定义好的数据仓库模型，将数据加载到数据仓库中去。随着业务需求的日益增加，不同系统的相互大批量数据交互也已成为常态。数据在不同系统中流动起来，有助于充分发掘日志大数据的价值。

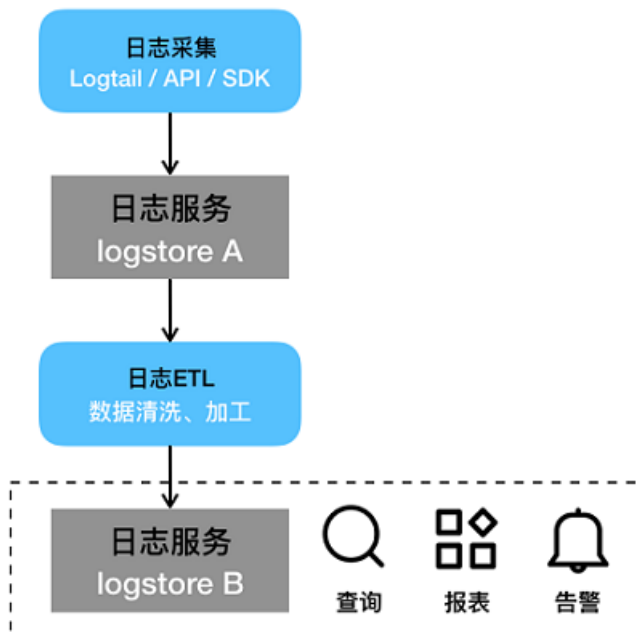
“日志服务+函数计算” ETL的优势

- 一站式采集、存储、加工、分析
- 全托管加工任务，按时间触发，自动重试
- 资源按shard水平扩展，满足大数据需求
- 基于函数计算提供数据加工，弹性资源，按需付费
- ETL对用户透明，提供日志、报警功能
- 持续增加内置函数模板，降低主流需求下的函数开发代价

应用场景

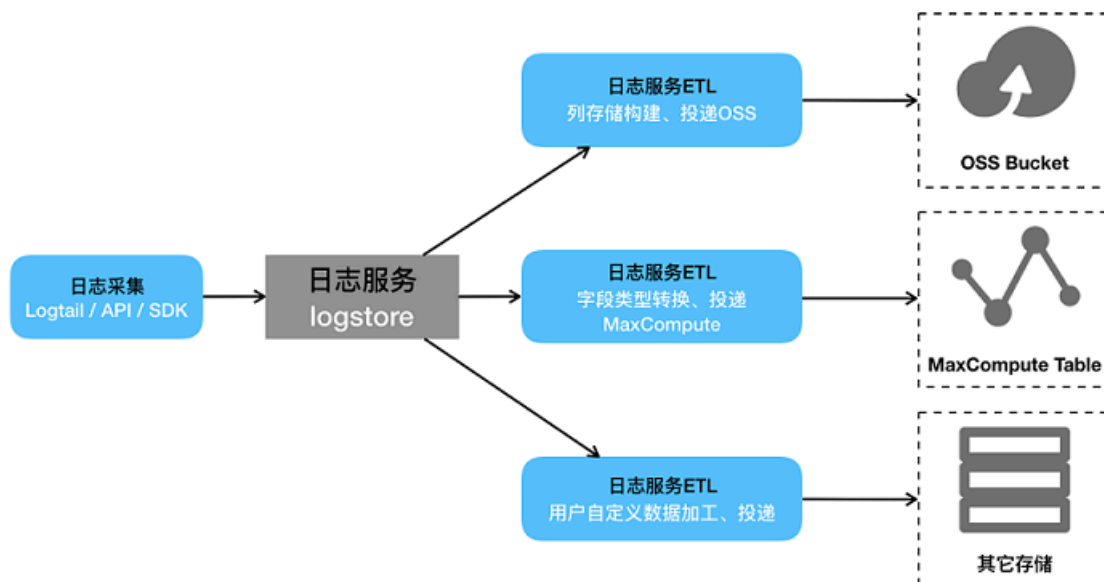
· 数据清洗、加工场景

通过日志服务，快速完成日志采集、加工、查询、分析。



· 数据投递场景

为数据的目的端落地提供支撑，构建云上大数据产品间的数据管道。

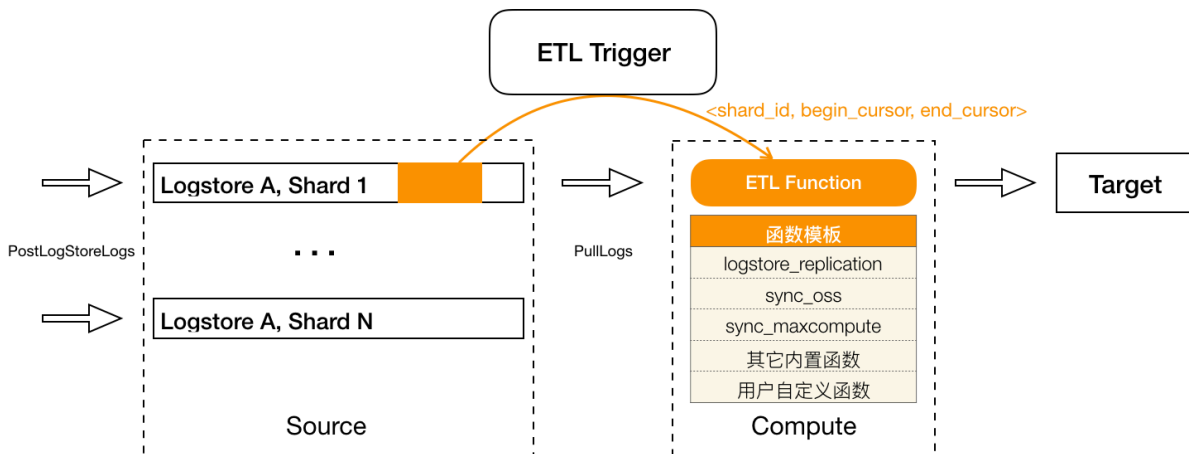


ETL模型

实时数据流处理，基于流的模型。ETL Trigger轮询源logstore下各shard的写入位置，并定时生成三元组信息触发函数执行，该三元组用于标识本次ETL任务所应该处理的数据范围。

通过shard的并发做到水平扩展，shard弹性伸缩保证了ETL的动态伸缩，通过定时器触发作业完成持续的数据加载。

在ETL任务执行层面，考虑UDF的灵活性，加工逻辑会跑在函数计算的函数上，而函数计算提供了按需付费、弹性伸缩能力以及自定义代码执行功能，正是很多云上用户所需要的。



ETL日志

· ETL过程日志

这是一类是执行过程日志，这一部分日志是在ETL执行过程中每执行一步的记录关键点和错误，包括某一步骤的开始、结束时间、初始化动作完成情况，模块出错信息等。记录日志的目的是随时可以知道ETL运行情况，如果出错了，可以知道哪里出错。函数运行产生的日志记录了数据加工过程中关键点、异常。

· ETL调度日志

调度日志只记录ETL任务开始的时间、结束时间，任务是否成功以及成功返回的信息。如果ETL任务出错了，不仅要形成ETL出错日志，而且要向系统管理员发送报警邮件或短信。在调度日志的基础上，可以构建出报表统计ETL的总体运行状况。

应用实例

通过Nginx、Apache等HTTP服务器构建的软件，可以记录每一个用户访问日志。通过日志服务+函数计算ETL，可以分析您服务了哪些地区的用户，这些用户通过什么链路访问您的服务。

对于数据分析工程师而言，ETL过程往往占据整个项目工作60%~70%的工作量。日志服务的目标是使用内置的函数模板的前提下，将构建ETL的时间缩短到15分钟内。

操作步骤

1. 日志集中化存储。

您可以使用日志服务的Logtail客户端快速接入机器上的日志文件。

客户端采集Nginx访问日志将会集中存储到日志服务的一个Logstore中，如下图，forward字段的IP记录了用户请求的来源。

时间 ↓	内容 ▼	⬆ ⬇
1	<pre>__source__: 10.177.162.203 __topic__: forward: 117.136.90.160 host: [REDACTED] ip: 172.29.94.109 latency: 0.083 method: GET outflow: 232594 port: 36245 protocol: HTTP/1.1 readtime: http_x_readtime ref: - site: [REDACTED] status: 200 time: 16/Oct/2017:11:38:21 +0800 ua: Mozilla/5.0 (Linux; Android 6.0.1; X800+ Build/BEXCNFN5902605111S; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/53.0.2785.49 Mobile MQQBrowser/6.2 TBS/043520 Safari/537.36 MicroMessenger/6.5.16.1120 NetType/4G Language/zh_CN upstream_time: 0.071 uri: / url: /?utm_content=se_919625&from=timeline&isappinstalled=0</pre>	

2. 云端数据加工。

a) 登录函数计算控制台创建Service。

在高级配置中，建议为ETL Function配置加工过程中的日志记录的存储Logstore，方便通过日志来定位加工过程中的异常行为。为函数授予日志服务AliyunLogFullAccess权限，函数在运行过程中会读源Logstore数据，数据处理后再写到目标Logstore。

 服务 > log_etl

帮助文档 服务实时监控

基本信息

修改

▼

服务名称: log_etl	所在区域: 华东 2
创建时间: 2017-08-09 16:19:35	修改时间: 2017-10-16 10:51:25
描述信息: log service etl	

高级配置

日志解耦

修改

▼

LogProject: etl-test	LogStore: etl-function-log
服务角色: acs:ram::1654218965343050:role/aliyunlogfullaccess	

函数管理

刷新

新建函数

输入函数名称模糊查询

搜索

函数名称	函数描述	创建时间	修改时间	执行语言	基本操作
ip-lookup		2017-10-13 14:58:28	2017-10-13 14:58:28	java8	删除

b) 通过内置模板创建函数。

 服务 > log_etl > 函数

帮助文档 服务实时监控

函数模板

触发器配置

基础管理配置

信息核对

业务模板提供了示例配置和代码，为您创建函数时涉及到的函数参数配置、触发器参数配置和代码实现提供参考。您可以选择与您业务模型相似的业务模板进行修改并创建函数，也可以选择空白函数模板自定义函数。

搜索模板

java

共有3条, 每页显示: 9条

«

1

»

GO

空白函数

python2.7/java8/nodejs6

空白函数模板会创建一个空白函数。通过引导页面进行触发器配置、函数参数配置和代码开发，完成函数的创建。

ip-lookup

java8 - log

查看详情

该函数订阅日志服务logstore的实时数据，根据日志字段中的ip值，查找ip数据库获得ip归属的国家、省、市、ISP信息。最终在原始数据基础上添加ip归属信息后写入另一个logstore。

logstore-replication

java8 - log

查看详情

该函数订阅日志服务logstore的实时数据，并复制数据到另一个logstore。

默认的函数配置如下：

 服务 > log_etl > 函数 > ip-lookup

帮助文档 服务实时监控

基本信息

点击下载代码

修改

▼

函数名称: ip-lookup	所在区域: 华东 2
代码大小: 4697394 字节	创建时间: 2017-10-13 14:58:28
修改时间: 2017-10-13 14:58:28	函数入口: com.aliyun.log_etl_function.IpLookup:handleRequest
运行环境: java8	执行内存: 768MB
超时时间: 120秒	描述信息:

触发器管理

刷新

新建触发器

触发器名称	事件源	事件类型	触发规则	操作
add-ip	acs:log:cn-shanghai:1654218965343050:project/etl-test	log:PostLogStoreLogs	时间触发	删除

c) 在函数上新建日志服务触发器。

日志服务触发器配置如下。

* 触发器名称: add-ip

* Project名称: etl-test

* LogStore名称: nginx_access_log

* 触发器日志: ?

* 触发间隔: 秒

注意事项:
» 1. 日志服务触发函数运行的间隔, 比如每隔120秒将 logstore 在最近120秒内的数据取出到函数服务, 以执行自定义计算。

* 重试次数: 次

注意事项:
» 1. 当日志服务触发函数执行出错时(授权失败、网络错误等原因), 日志服务调度函数进行重试的最大次数。重试的影响, 因具体的函数代码实现逻辑而异。

* 函数配置:

```
1 {
2   "source": {},
3   "target": {
4     "endpoint": "http://cn-shanghai-intranet.log.aliyuncs.com",
5     "logstoreName": "nginx_access_log_etl",
6     "projectName": "etl-test"
7   },
8   "transform": {
9     "cityKeyName": "city",
10    "countryKeyName": "country",
11    "ipKeyName": "forward",
12    "ispKeyName": "isp",
13    "ossBucketOfIpData": "log-etl-resources",
14    "ossEndpointOfIpData": "http://oss-cn-hangzhou-internal.aliyuncs.com",
15    "ossObjectOfIpData": "ipdata/ipdata_geo_isp_code.txt.utf8.gz",
16    "provinceKeyName": "province"
17  }
18 }
```

指定数据源为第一步中采集到中心化Nginx日志Logstore, 例如本例子的Projectetl-test/logstore:nginx_access_log。

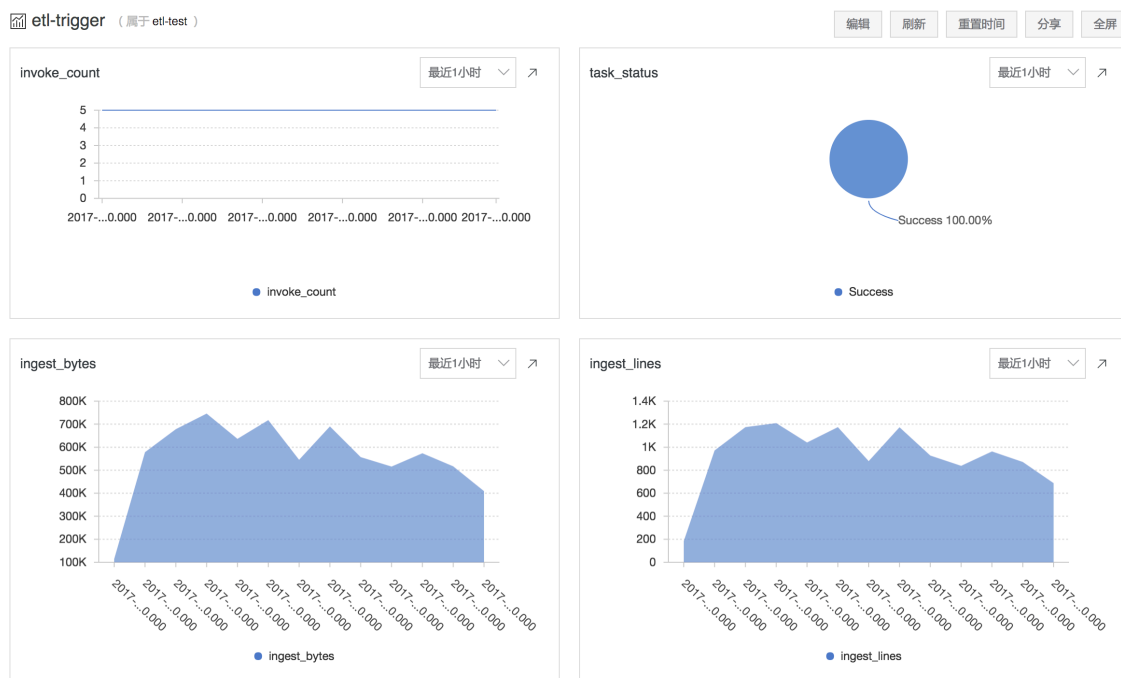
日志服务将轮询Logstore的数据, 当数据持续产生时, 每60秒创建一次ETL任务, 并调用函数执行。触发函数执行以及函数执行结果将会记录到触发器日志logstore: etl-trigger-log中。

不同函数的实现和功能不同, 其配置也不相同, ip-lookup的详细配置项说明请参考README。

d) 保存配置, 等待1分钟后ETL任务开始执行。

可以关注一下ETL过程日志、调度日志, 按如上配置, 分别在logstore: etl-function-log、etl-trigger-log。

可以通过查询语句构建出如本文日志部分所示的报表：



- 截图左上角是每分钟调度函数执行的触发次数，构建自查询语句。

```
project_name : etl-test and job_name : ceff019ca3d077f85aca
ad35bb6b9bba65da6717 | select from_unixtime(__time__ - time
% 60) as t, count(1) as invoke_count group by from_unixtime(
__time__ - time % 60) order by t asc limit 1000
```

- 截图右上角是ETL任务成功、失败的比例，构建自查询语句。

```
project_name : etl-test and job_name : ceff019ca3d077f85aca
ad35bb6b9bba65da6717 | select task_status, count(1) group by
task_status
```

- 截图左下角是每5分钟的摄入的日志字节数，构建自查询语句。

```
project_name : etl-test and job_name : ceff019ca3d077f85aca
ad35bb6b9bba65da6717 and task_status : Success | select
from_unixtime(__time__ - time % 300) as t, sum(ingest_bytes)
as ingest_bytes group by from_unixtime(__time__ - time % 300)
order by t asc limit 1000
```

- 截图右下角则是每5分钟摄入处理的日志行数，构建自查询语句。

```
project_name : etl-test and job_name : ceff019ca3d077f85aca
ad35bb6b9bba65da6717 and task_status : Success | select
from_unixtime(__time__ - time % 300) as t, sum(ingest_lines)
as ingest_lines group by from_unixtime(__time__ - time % 300)
order by t asc limit 1000
```

3. 加工后数据建模。

机器上的Nginx日志经由Logtail实时采集到源logstore，再由ETL准实时加工后写出到目标Logstore。经函数处理后带IP信息数据如下：

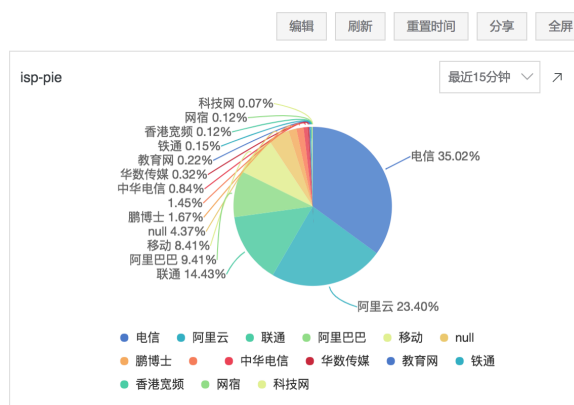
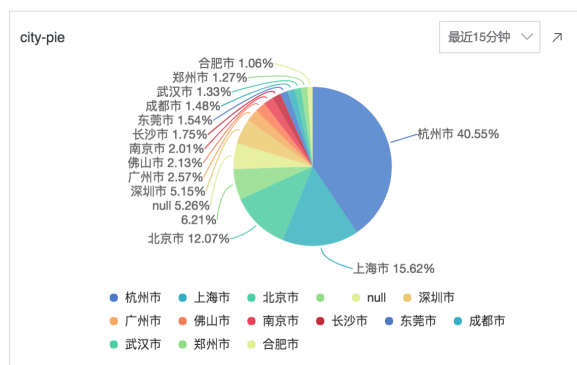
时间	内容
10-16 11:38:21	<pre>__source__: 10.177.162.203 __topic__: city: country: 中国 forward: 10.136.90.160 host: 10.136.90.160 ip: 172.29.94.109 isp: 移动 latency: 0.083 method: GET outflow: 232594 port: 36245 protocol: HTTP/1.1 province: 山西省 readtime: http_x_readtime ref: - site: http_x_readtime status: 200 time: 16/Oct/2017:11:38:21 +0800 ua: Mozilla/5.0 (Linux; Android 6.0.1; X800+ Build/BEXCNFN5902605111S; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/53.0.2785.49 Mobile MQQBrowser/6.2 TBS/043520 Safari/537.36 MicroMessenger/6.5.16.1120 NetType/4G Language/zh_CN upstream_time: 0.071 uri: / url: /?utm_content=se_919625&from=timeline&isappinstalled=0</pre>

对比加工前后，可以发现，新的数据增加了四个字段（country、province、city、isp），可以知道：IP源117.136.90.160的请求来自中国山西太原，运营商是中国移动。

接下来，使用日志服务的日志分析功能查询一个时间段内请求IP的城市和ISP分布。通过如下两个查询语句构建报表：

- * | select city, count(1) as c group by city order by c desc limit 15
- * | select isp, count(1) as c group by isp order by c desc limit 15

access-ip-statistic (属于 etl-test)



14.2 消费-搭建监控系统

日志服务是阿里云一个重要的基础设施，支撑着阿里云所有集群日志数据的收集和分发。众多应用比如OTS、ODPS、CNZZ等利用日志服务logtail收集日志数据，利用API消费数据，导入下游实时统计系统或者离线系统做分析统计。作为一个基础设施，日志服务具备：

- 可靠性：经过多年阿里集团内部用户的检验，经历多年双十一考验，保证数据的可靠、不丢失。
- 可扩展性：数据流量上升，通过增加shard个数快速动态扩容处理能力。
- 便捷性：一键式管理包括上万台机器的日志收集。

日志服务帮用户完成了日志收集，统一了日志格式，提供API用于下游消费。下游系统可以接入多重系统重复消费，比如导入Spark、Storm做实时计算，也可以导入elastic search做搜索，真正做到了一次收集，多次消费。在众多数据消费场景中，监控是最常见的一种场景。本文介绍阿里云基于日志服务的监控系统。

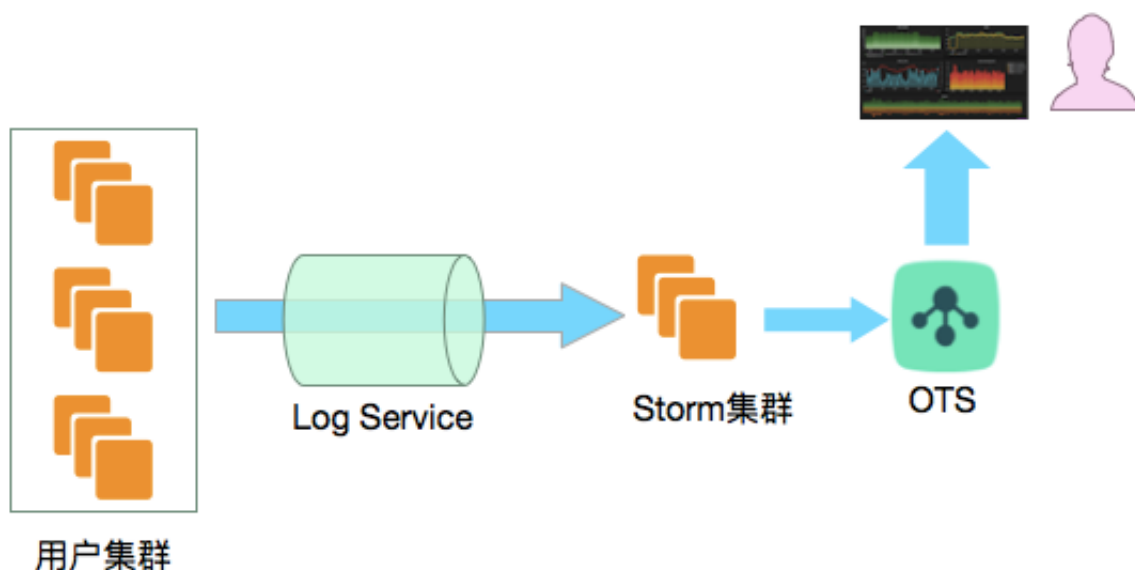
日志服务把所有集群的监控数据作为日志收集到服务端，解决了多集群管理和异构系统日志收集的问题，监控数据统一成格式化的日志发送到日志服务。

日志服务为监控系统提供了：

- 统一的机器管理：安装一次logtail，所有的后续操作在日志服务端进行。
- 统一的配置管理：需要收集哪些日志文件，只要在服务端配置一次，配置会自动下发到所有机器。
- 结构化的数据：所有数据格式化成日志服务的数据模型，方便下游消费。

- 弹性的服务能力：处理大规模数据写入和读取的能力。

图 14-1: 监控系统架构



如何搭建监控系统

1. 收集监控数据

配置SLS的日志收集，确保日志收集到了日志服务。

2. 中间件使用API消费数据

通过SDK的PullLog接口从日志服务批量消费日志数据，并且把数据同步到下游实时计算系统。

3. 搭建storm实时计算系统

选择storm或者其他的类型的实时计算系统，配置计算规则，选择要计算的监控指标，计算结果写入到OTS中。

4. 展示监控信息

通过读取保存在OTS中的监控数据，在前端展示；或者读取数据，根据数据结果做报警。

14.3 消费-计量计费日志

使用云服务最大好处是按量付费，无需预留资源，因此各云产品都有计量计费需求。这里我们介绍一种基于[日志服务](#)计量计费方案，该方案每天处理千亿级计量日志，被众多云产品使用。

计量日志生成计费结果过程

计量日志记录了用户涉及计费的项目，后台计费模块根据计费项和规则进行运算，产生最后账单。例如如下原始访问日志记录了项目（Project）使用情况：

```
microtime:1457517269818107 Method:PostLogStoreLogs Status:200 Source:
10.145.6.81 ClientIP:112.124.143.241 Latency:1968 InFlow:1409 NetFlow
:474 OutFlow:0 UserId:44 AliUid:1264425845***** ProjectName:app-
myapplication ProjectId:573 LogStore:perf UserAgent:ali-sls-logtail
APIVersion:0.5.0 RequestId:56DFF2D58B3D939D691323C7
```

计量计费程序读取原始日志，根据规则生成用户在各维度使用数据（包括流量、使用次数、出流量等）：

C	D	E	F	G	H	I	J	K	L
uid	project	region	inflowsize	writecount	readcount	outflowsize	network_out	shard_size	index_size
1.47543E+15	aquilapreproductionenviron	cn-hangzhou	437	0	0	0	0	48	790
1.76991E+15	ali-tbosstest-log	cn-hangzhou-corp	0	0	0	0	0	92	0
1.72535E+15	ali-icbu-janus-log	cn-shanghai-corp	229879259	0	0	0	0	96	#####
1.62572E+15	corp-scmg-admin	cn-hangzhou-stg	15709	0	0	0	0	16	82344
1.19214E+15	dtboost	cn-hangzhou	0	0	0	0	0	48	0
1.63404E+15	md-oa	cn-beijing	0	0	0	0	0	240	0
1.85233E+15	ots_e2e_test_2nd	cn-hangzhou-stg	0	0	0	0	0	8	0
1.2358E+15	wxb-log	cn-hangzhou	466323	0	0	0	0	240	1394118
1.26443E+15	portal-b8568f751df75214dc	cn-hangzhou-stg	0	73811	0	500	73811	600	0
1.26854E+15	ali-pangumaster-log-hangz	cn-hangzhou-stg	98041	0	0	0	0	4	633933
1.85386E+15	daily	cn-hangzhou	205159	0	0	0	0	96	0
1.59853E+15	ali-alipay-siteprobe-test	cn-hangzhou-corp	0	0	0	0	0	184	0
34762362	1111111	cn-qingdao	0	0	0	0	0	96	0

典型计量日志计费场景

- 电力公司：每10秒会收到一条日志，记录该10秒内每个用户ID下该周期内功耗、峰值、均值等，每天、每小时和每月给用户账单。
- 运营商：每隔10秒从基站收到时间段内某个手机号码的动作（上网、电话、短信、VoIP），使用量（流量），时长等信息，后台计费服务统计出该区间内消耗资费。
- 天气预测API服务：根据用户调用接口类型、城市、查询类型、结果大小等对用户请求进行收费。

要求与挑战

既要算对，又要算准是一件要求很高的事情，系统要求如下：

- 准确可靠：既不可多算，也不能少算。
- 灵活：支持补数据等场景，例如一部分数据没有推送过来，当需要修正时可以重新计算。
- 实时性强：能够做到秒级计费，对于欠费场景快速切断。

其他需求 (Plus)：

- 账单修正功能：在实时计费失败时，我们可以通过理想计费进行对账。
- 查询明细：支持用户查看自己消费明细。

现实中还有两类挑战：

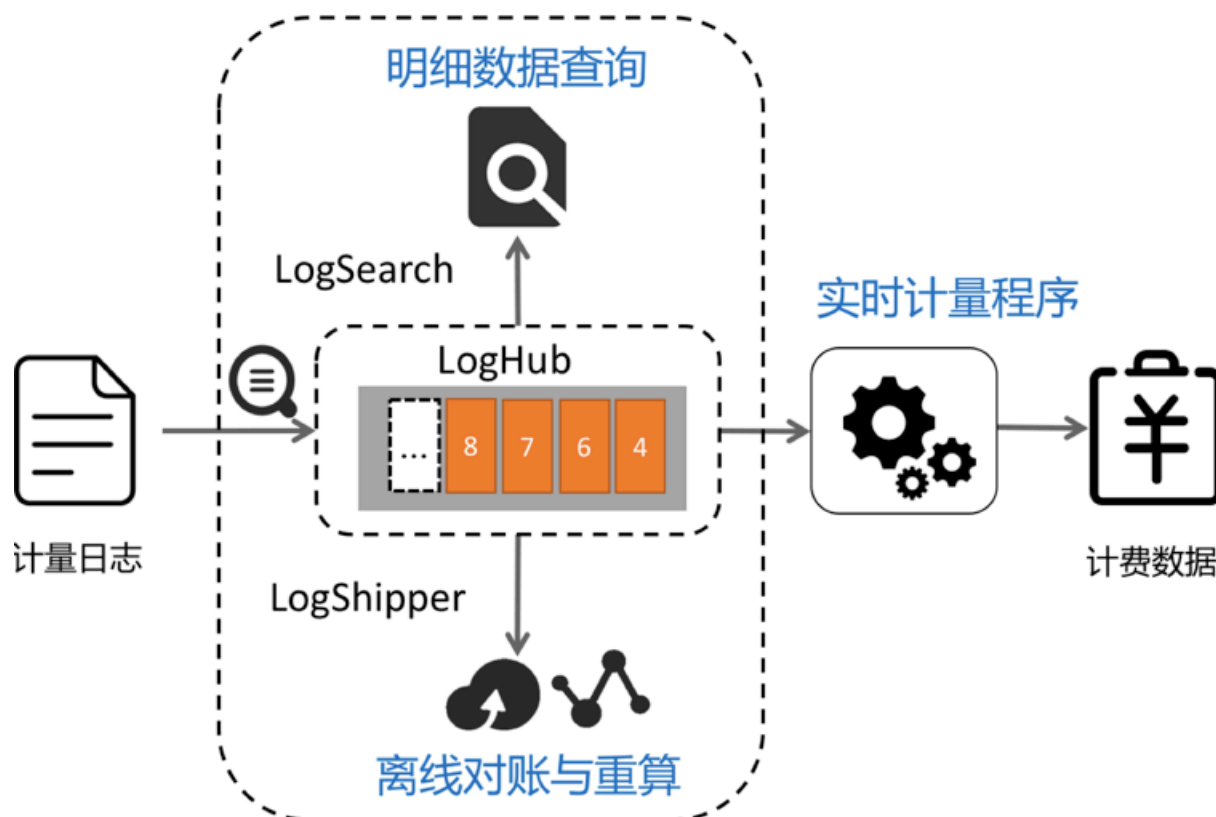
- 不断增长的数据量：随着用户以及调用上升，数据规模会越来越大，如何保持架构的弹性伸缩。
- 容错处理：计费程序可能有Bug，如何确保计量数据与计费程序独立。

本文档主要介绍一种阿里云基于日志服务开发计量计费方案，该方案已在线上稳定运行多年，从未出现过一例算错、延迟等情况，供单价参考。

系统架构

以[阿里云日志服务](#)的LogHub功能为例：

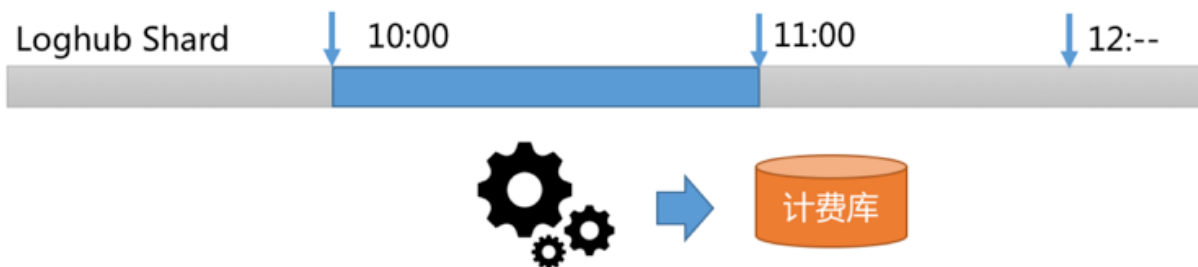
1. 使用LogHub进行计量日志实时采集与计量程序对接：LogHub 支持的30+种API和接入手段，接入计量日志非常容易。
2. 计量程序每隔固定时间消费LogHub中步长数据，在内存中计算结果生成计费数据。
3. （附加）对明细数据查询需求，可以将计量日志配置索引查询。
4. （附加）将计量日志推送至OSS、MaxCompute进行离线存储，进行T+1等对账与统计。



实时计量程序内部结构：

1. 根据LogHub读取接口GetCursor功能，选定某个时间段日志（例如10:00-11:00）Cursor。
2. 通过PullLogs接口消费该时间段内数据。
3. 在内存中进行数据统计与计算，拿到结果，生成计费数据。

我们可以以此类推，把选择时间计算逻辑修改为1分钟，10秒钟等。



性能分析：

- 假设有10亿条/天计量日志，每条长度为200字节，数据量为200GB。
- LogHub 默认SDK或Agent都带压缩功能，实际存储数据量为40GB（一般至少有5倍压缩率），一个小时数据量为 $40/24 = 1.6\text{GB}$ 。
- LogHub读取接口一次最大支持读1000个包（每个包最大为5MB），在千兆网条件下2秒内即可读完。
- 加上内存中数据累计与计算时间，对1小时计量日志进行汇总，不超过5秒。

数据量大应如何解决

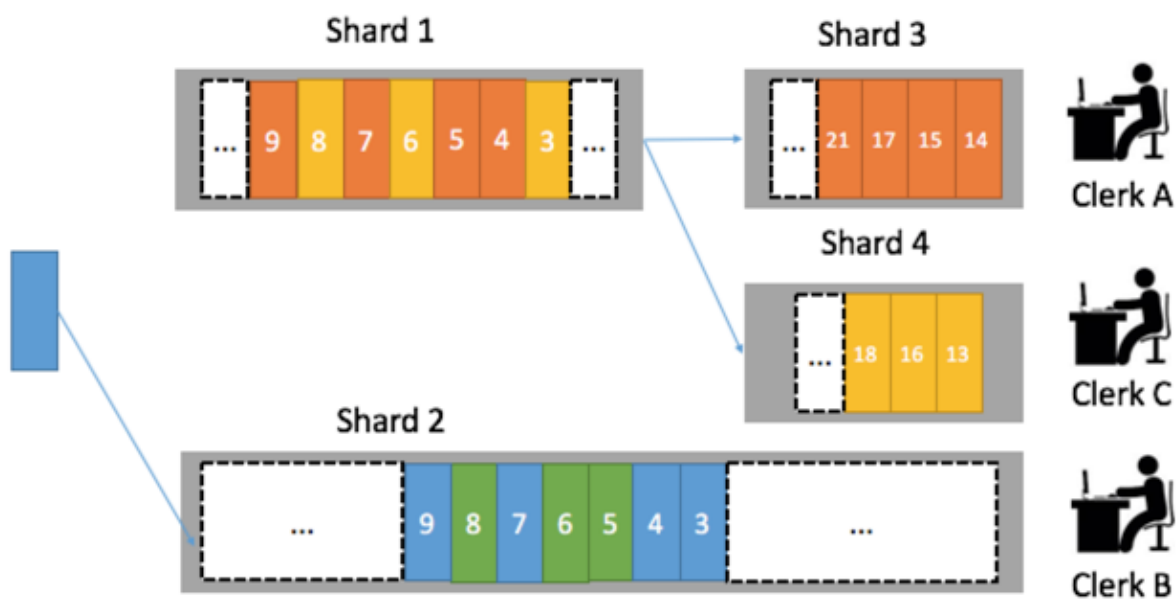
在一些计费场景下（例如运营商、IoT等）计量日志量会很大（例如十万亿，数据量为2PB/Day），折算压缩数据后一小时有16 TB，以万兆网络读取需要1600秒，已不能满足快速出账单需求。

1. 控制产生的计费数据量

我们对于产生计量日志程序进行改造（例如Nginx），先在内存中做了聚合，每隔1分钟Dump一次该时间段聚合的汇总计量日志结果。这样数据量就和总体的用户数相关了：假设Nginx该时间段内有1000个用户，一个小时数据点也才 $1000 * 200 * 60 = 12\text{ GB}$ （压缩后为240 MB）。

2. 将计量日志处理并行化

LogHub下每个日志库可以分配不同Shard（分区），我们可以分配3个分区，3个计量消费程序。为了保证一个用户计量数据总是由一个消费程序处理，我们可以根据用户ID Hash到固定Shard中。例如杭州市西湖区用户写在1号Shard，杭州上城区用户数据写在2号Shard，这样后台计量程序就可水平扩展。



其他问题

- 补数据怎么办？

LogHub 下每个Logstore可以设置生命周期（1-365天），如果计费程序需要重新消费数据，在生命周期内可以任意根据时间段进行计算。

- 计量日志散落在很多服务器（前端机）怎么办

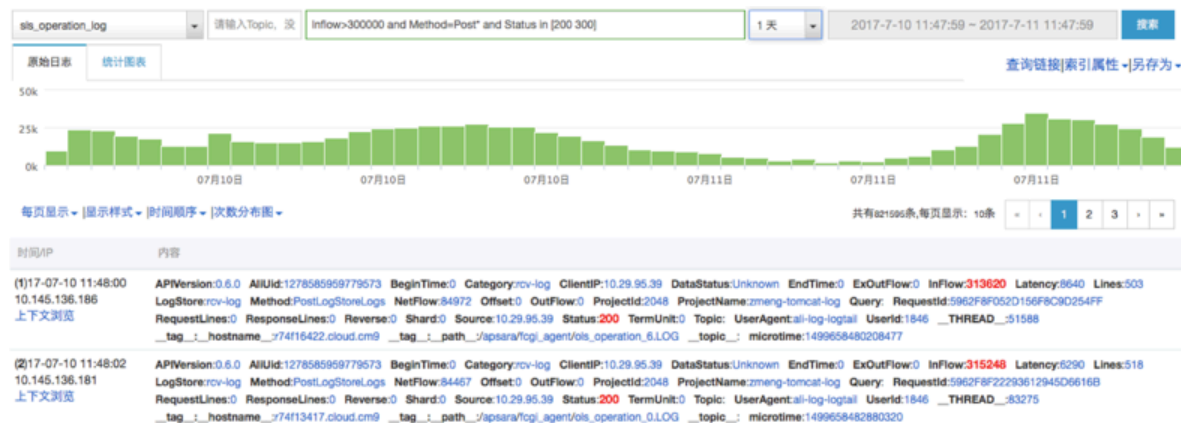
1. 使用Logtail Agent实时采集
2. 使用机器标示定义一套动态机器组弹性伸缩

· 查询明细需求如何满足

对LogHub中数据可以创建索引，支持实时查询与统计分析，例如我们想调查有一些特别大的计量日志：

```
Inflow>300000 and Method=Post* and Status in [200 300]
```

在对Loghub中数据打开索引后，即可实时查询与分析。



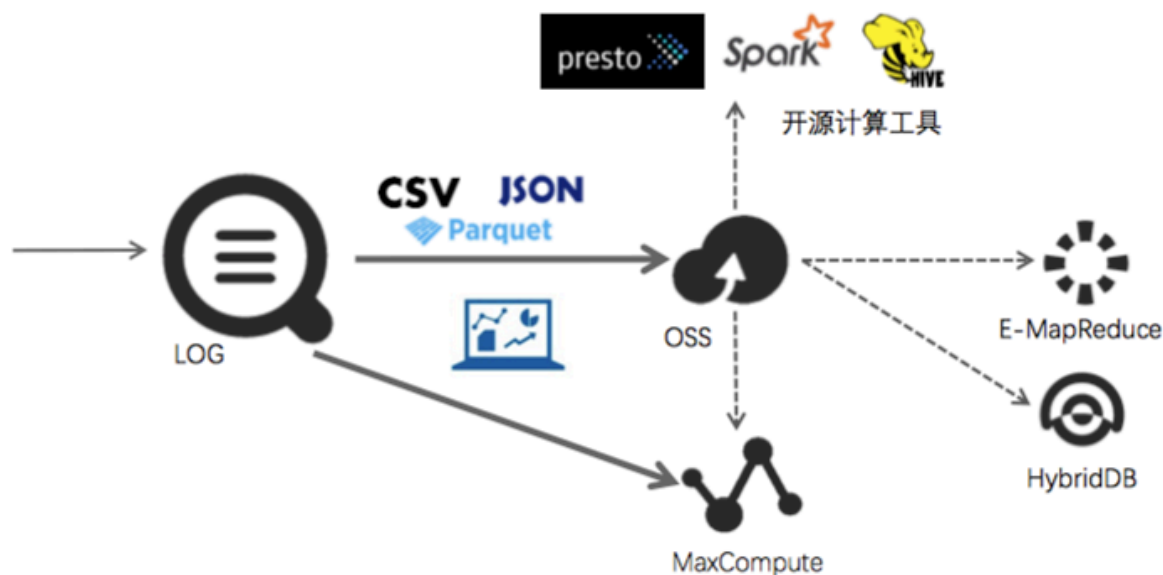
也可以在查询后加上统计分析：

```
Inflow>300000 and Method=Post* and Status in [200 300] | select max(Inflow) as s, ProjectName group by ProjectName order by s desc
```

ProjectName	s
rel-stat-staff-benefits	1207132
rel-acticle-user-action	816968
tteduhaproxy	688385
noc	583006
xiaobinggd	539798
ludashi-stat	534489
sis-weloop	526956
ykf-testalipay	524523
fc-monitor-ol	524515
syt-accesslog	486647

- 存储日志并进行T+1对账

日志服务提供LogHub中数据投递功能，支持自定义分区、自定义存储格式等将日志存储在OSS/MaxCompute上，利用E-MapReduce、MaxCompute、HybridDB、Hadoop、Hive、Presto、Spark等进行计算。



14.4 消费-通过Consumer Library实现高可靠消费

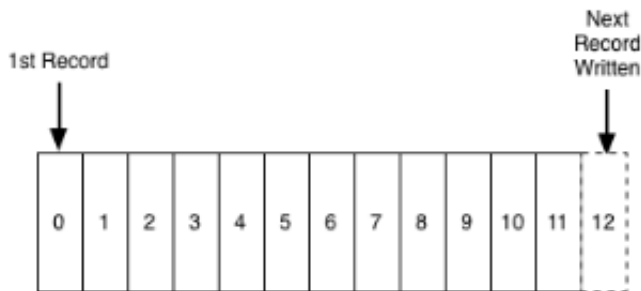
日志处理是一个很大范畴，其中包括实时计算、数据仓库、离线计算等众多点。这篇文章主要介绍在实时计算场景中，如何能做到日志处理保序、不丢失、不重复，并且在上下游业务系统不可靠（存在故障）、业务流量剧烈波动情况下，如何保持这三点。

为方便理解，本文使用《银行的一天》作为例子将概念解释清楚。在文档末尾，介绍日志服务LogHub功能，如何与Spark Streaming、Storm Spout等配合，完成日志数据的处理过程。

问题定义

什么是日志数据？

原LinkedIn员工Jay Kreps在《[The Log: What every software engineer should know about real-time data's unifying abstraction](#)》描述中提到：“append-only, totally-ordered sequence of records ordered by time”。



- **Append Only**: 日志是一种追加模式，一旦产生过后就无法修改。
- **Totally Ordered By Time**: 严格有序，每条日志有一个确定时间点。不同日志在秒级时间维度上可能有重复，比如有2个操作GET、SET发生在同一秒钟，但对于计算机而言这两个操作也是有顺序的。

什么样的数据可以抽象成日志？

半世纪前说起日志，想到的是船长、操作员手里厚厚的笔记。如今计算机诞生使得日志产生与消费无处不在：服务器、路由器、传感器、GPS、订单、及各种设备通过不同角度描述着我们生活的世界。从船长日志中我们可以发现，日志除了带一个记录的时间戳外，可以包含几乎任意的内容，例如：一段记录文字、一张图片、天气状况、船行方向等。半个世纪过去了，“船长日志”的方式已经扩展到一笔订单、一项付款记录、一次用户访问、一次数据库操作等多样的领域。

在计算机世界中，常用的日志有：Metric, Binlog (Database, NoSQL), Event, Auditing, Access Log 等。

在我们今天的演示例子中，我们把用户到银行的一次操作作为一条日志数据。其中包括用户、账号名、操作时间、操作类型、操作金额等。

例如：

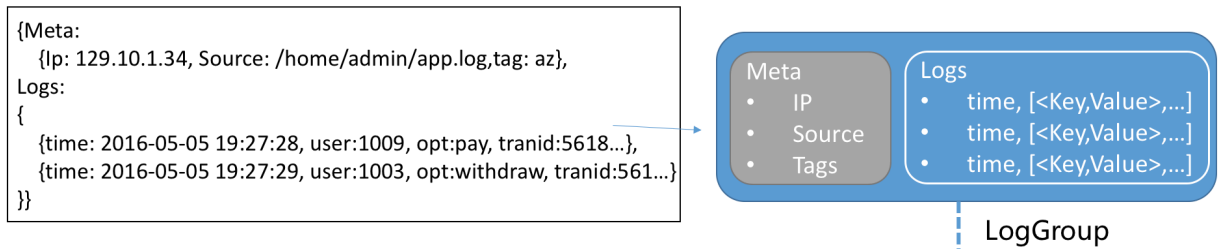
```
2016-06-28 08:00:00 张三 存款 1000元
2016-06-27 09:00:00 李四 取款 20000元
```

LogHub数据模型

为了能抽象问题，这里以[阿里云日志服务](#)下LogHub作为演示模型，详细可以参见日志服务下[基本概念](#)。

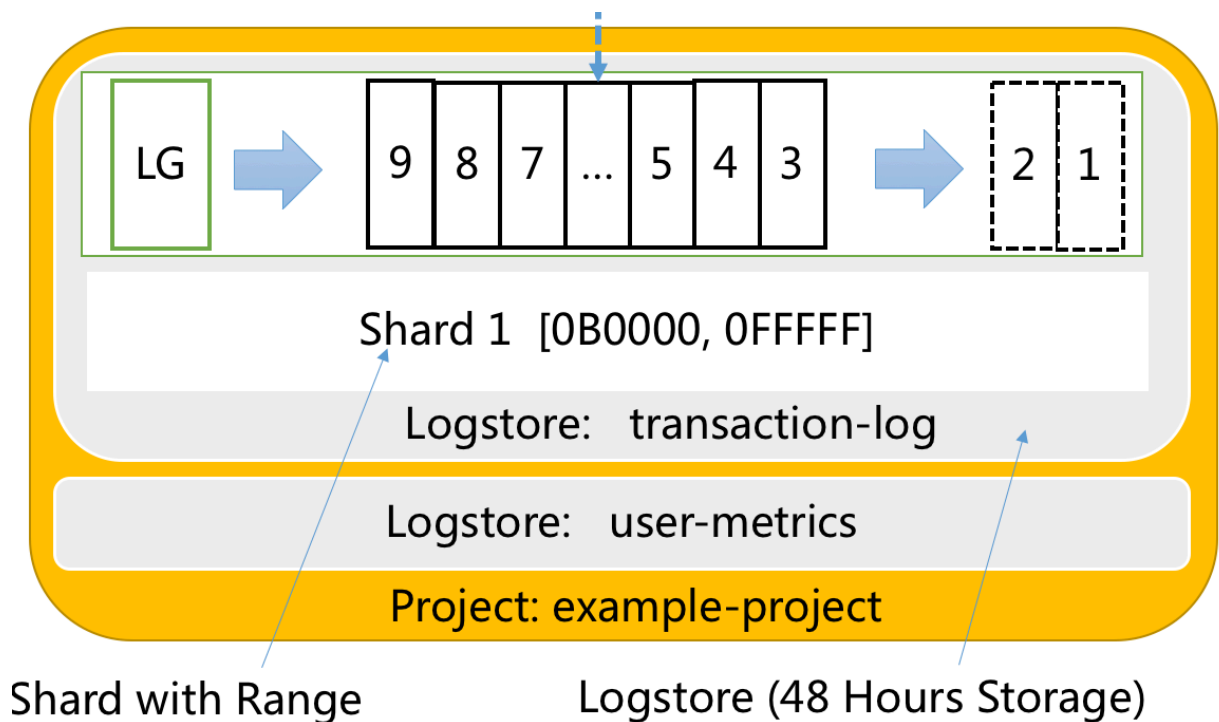
- **Log**: 由时间、及一组Key-Value对组成。
- **LogGroup**: 一组日志的集合，包含相同Meta (IP, Source) 等。

两者关系如下：



- **Shard**: 分区，LogGroup读写基本单元，可以理解为48小时为周期的FIFO队列。每个Shard提供 5 MB/S Write, 10 MB/S Read能力。Shard 有逻辑区间（BeginKey, EndKey）用以归纳不同类型数据。
- **Logstore**: 日志库，用以存放同一类日志数据。Logstore是一个载体，通过由[0000, FFFF ..)区间Shard组合构建而成，Logstore会包含1个或多个Shard。
- **Project**: Logstore存放容器。

这些概念相互关系如下：



银行的一天

以19世纪银行为例。某个城市有若干用户（Producer），到银行去存取钱（User Operation），银行有若干个柜员（Consumer）。因为19世纪还没有电脑可以实时同步，因此每个柜员都有一个小账本能够记录对应信息，每天晚上把钱和账本拿到公司去对账。

在分布式世界里，我们可以把柜员认为是固定内存和计算能力单机。用户是来自各个数据源的请求，Bank大厅是处理用户存取数据的日志库（Logstore）。

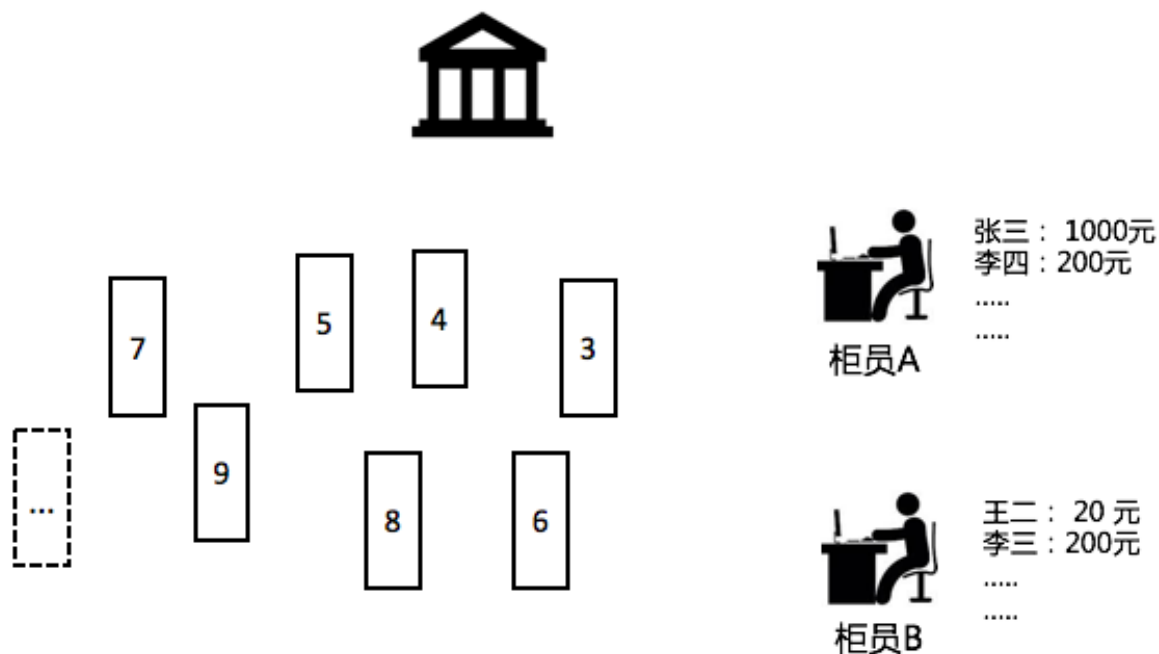


- Log/LogGroup：用户发出的存取款等操作。
- 用户（User）：Log/LogGroup生产者。
- 柜员（Clerk）：银行处理用户请求的员工。
- 银行大厅（Logstore）：用户产生的操作请求先进入银行大厅，再交给柜员处理。
- 分区（Shard）：银行大厅用以安排用户请求的组织方式。

问题1：保序（Ordering）

银行有2个柜员（A，B），张三进了银行，在柜台A上存了1000元，A把张三1000元存在自己的账本上。张三到了下午觉得手头紧到B柜台取钱，B柜员一看账本，发现不对，张三并没有在这里存钱。

从这个例子可以看到，存取款是一个严格有序的操作，需要同一个柜员（处理器）来处理同一个用户的操作，这样才能保持状态一致性。

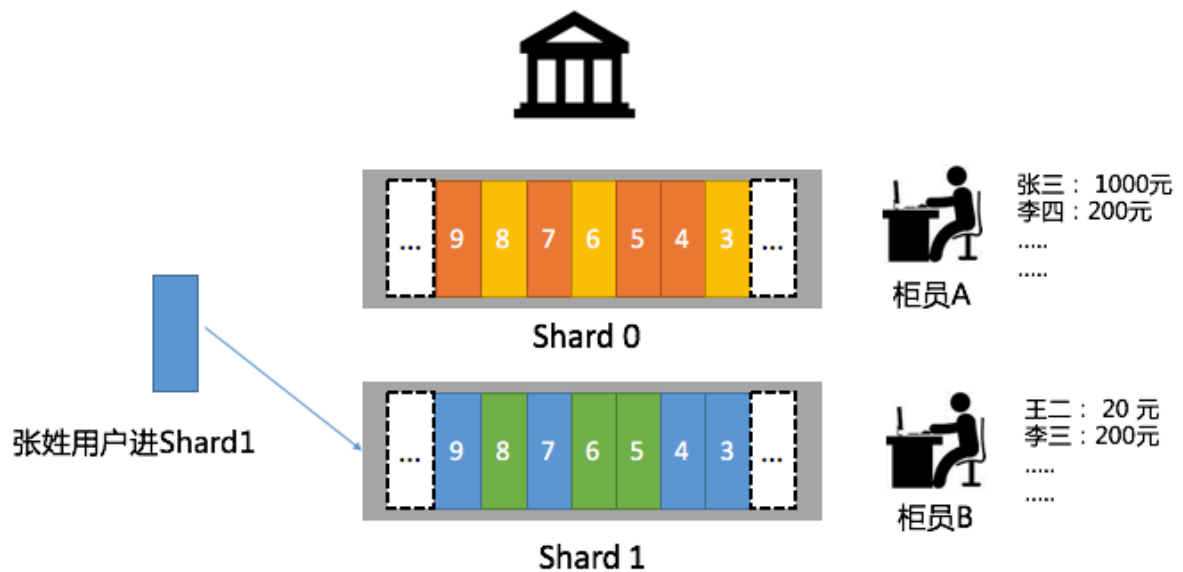


实现保序的方法很简单：排队，创建一个Shard，终端只有一个柜员A来处理。用户请求先进出，一点问题都没有。但带来的问题是效率低下，假设有1000个用户来进行操作，即使有10个柜员也无济于事。

这种场景怎么办？

1. 假设有10个柜员，我们可以创建10个Shard。
2. 如何保证对于同一个账户的操作是有序的？可以根据一致性Hash方式将用户进行映射。例如我们开10个队伍（Shard），每个柜员处理一个Shard，把不同银行账号或用户姓名，映射到特定Shard中。在这种情况下张三 Hash（Zhang）= Z 永远落在一个特定Shard中（区间包含Z），处理端面对的永远是柜员A。

当然如果张姓用户比较多，也可以换其他策略。例如根据用户AccountID、ZipCode进行Hash，这样就可以使得每个Shard中操作请求更均匀。



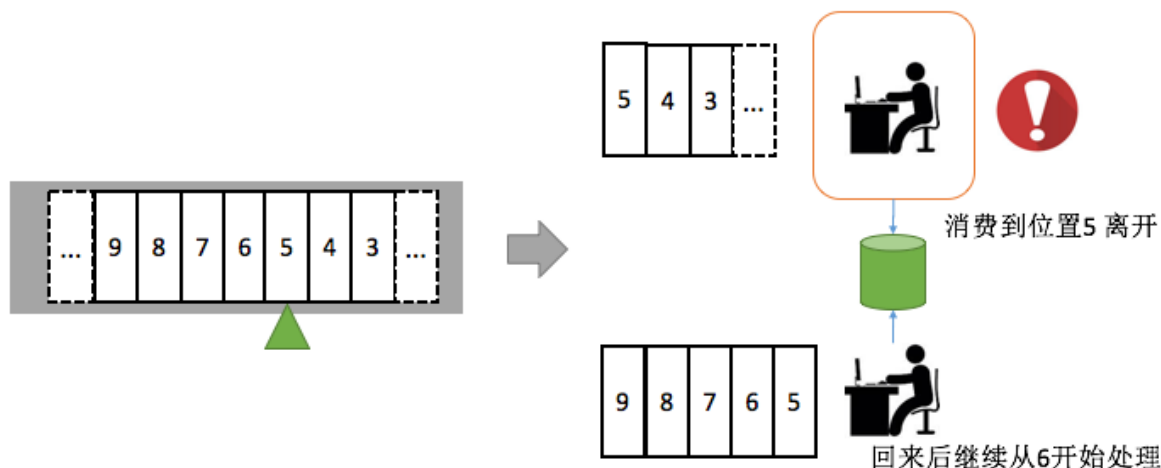
问题2：不丢失（At-Least Once）

张三拿着存款在柜台A处理，柜员A处理到一半去接了个电话，等回来后以为业务已经办理好了，于是开始处理下一个用户的请求，张三的存款请求因此被丢失。

虽然机器不会人为犯错，在线时间和可靠性要比柜员高。但难免也会遇到电脑故障、或因负载高导致的处理中断，因为这样的场景丢失用户的存款，这是万万不行的。

这种情况怎么办呢？

A可以在自己日记本上（非账本）记录一个项目：当前已处理到Shard哪个位置，只有当张三的这个存款请求被完全确认后，柜员A才能叫下一个。



带来问题是什么？可能会重复。比如A已经处理完张三请求（更新账本），准备在日记本上记录处理到哪个位置之时，突然被叫开了，当他回来后，发现张三请求没有记录下来，他会把张三请求再次处理一遍，这就会造成重复。

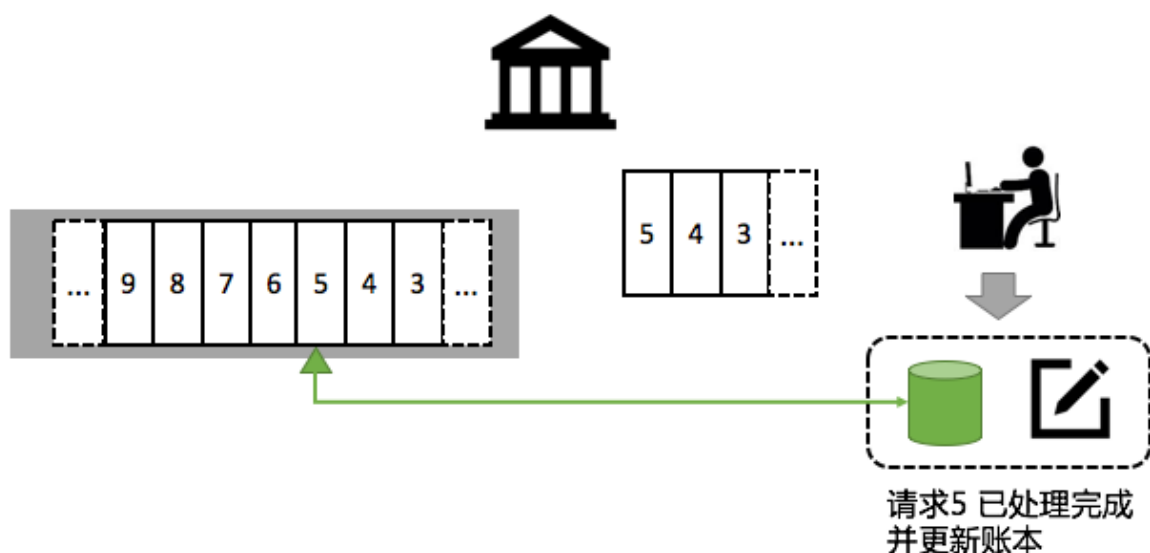
问题3：不重复（Exactly Once）

重复一定会带来问题吗？不一定。

在幂等情况下，重复虽然会有浪费，但对结果没有影响。什么叫幂等：重复消费不对结果产生影响的操作叫做幂等。例如用户有一个操作“查询余额”，该操作是一个只读操作，重复做不影响结果。对于非只读操作，例如注销用户这类操作，可以连续做两次。

但现实生活中大部分操作不是幂等的，例如存款、取款等，重复进行计算会对结果带来致命的影响。解决的方式是什么呢？柜员（A）需要把账本完成 + 日记本标记Shard中处理完成作为一个事物合并操作，并记录下来（CheckPoint）。

如果A暂时离开或永久离开，其他柜员只要使用相同的规范：记录中已操作则处理下一个即可，如果没有则重复做，过程中需要保证原子性。



Checkpoint可以将Shard 中的元素位置（或时间）作为Key，放入一个可以持久化的对象中。代表当前元素已经被处理完成。

业务挑战

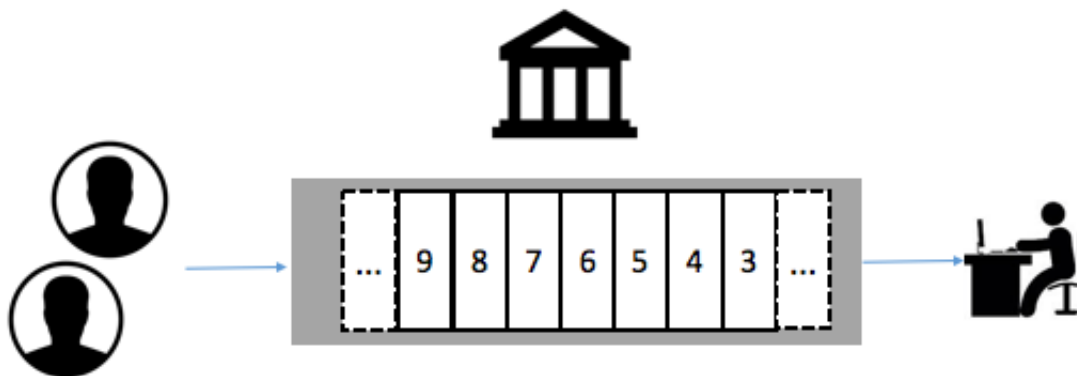
以上三个概念解释完成后，原理并不复杂。但在现实世界中，规模的变化与不确定性会使得以上三个问题便得更复杂。例如：

1. 遇到发工资日子，用户数会大涨。
2. 柜员（Clerk）毕竟不是机器人，他们需要休假，需要吃午饭。
3. 银行经理为了整体服务体验，需要加快柜员，以什么作为判断标准？Shard中处理速度？
4. 柜员在交接过程中，能否非常容易地传递账本与记录？

现实中的一天

8点银行开门

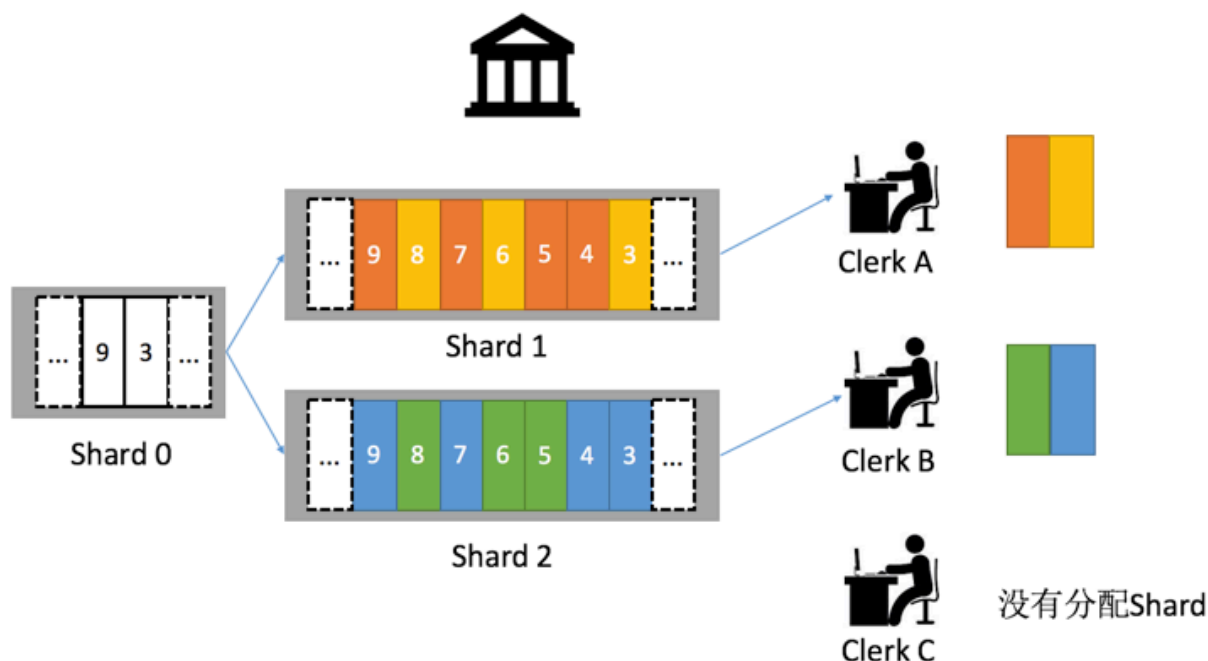
只有一个Shard0，用户请求全部排在Shard0下，柜员A也正好可以处理。



10点进入高峰期间

银行经理决定把10点后Shard0分裂成2个新Shard（Shard1，Shard2），并且给了如下规定，姓名是[A-W]用户到Shard1中排队，姓名是[X, Y, Z] 到Shard 2 中排队等待处理，为什么这两个Shard区间不均匀？因为用户的姓氏本身就是不均匀的，通过这种映射方式可以保证柜员处理的均衡。

10-12点请求消费状态：

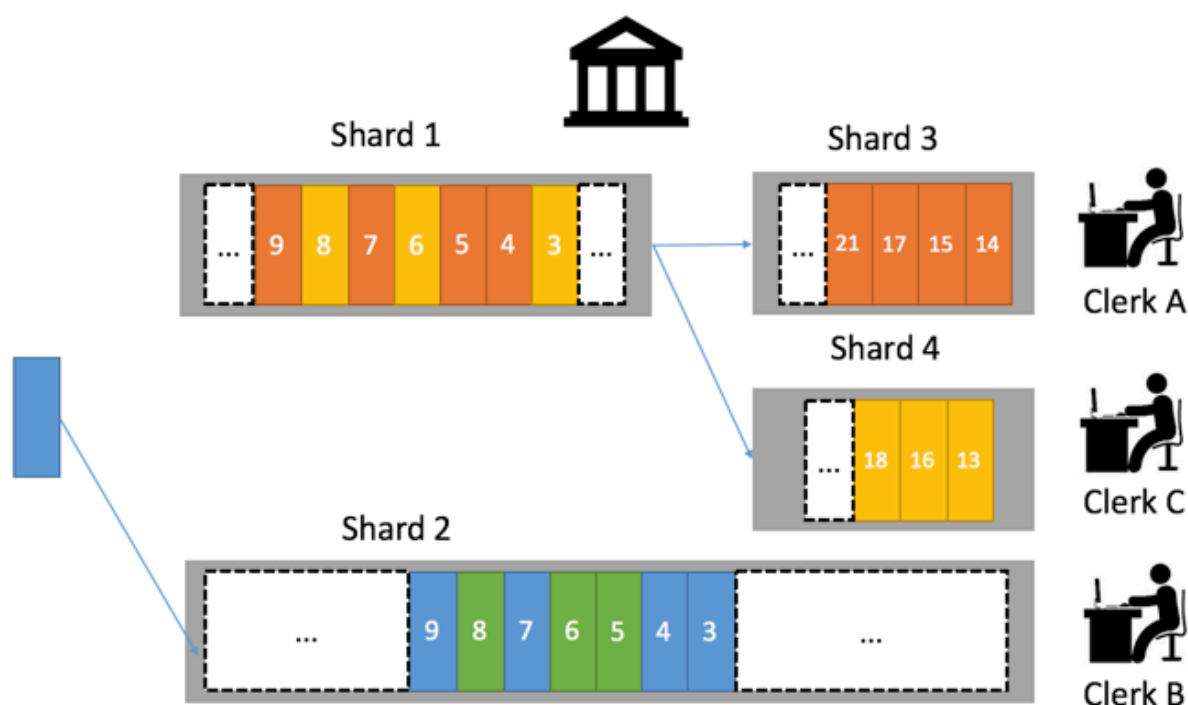


柜员A处理2个Shard非常吃力，于是经理派出柜员B、C出厂。因为只有2个Shard，B开始接管A负责一个Shard，C处于闲置状态。

中午12点人越来越多

银行经理觉得Shard1下柜员A压力太大，因此从Shard1中拆分出（Shard3，Shard4）两个新的Shard，Shard3由柜员A处理、Shard4由柜员C处理。在12点后原来排在Shard 1中的请求，分别到Shard3，Shard4中。

12点后请求消费状态：



流量持续到下午4点后，开始逐渐减少

因此银行经理让柜员A、B休息，让C同事处理Shard2，Shard3，Shard4中的请求。并逐步将Shard2与Shard3合并成Shard5，最后将Shard5和Shard4合并成一个Shard，当处理完成Shard中所有请求后银行关门。

现实中的日志处理

上述过程可以抽象成日志处理的经典场景，如果要解决银行的业务需求，我们要提供弹性伸缩、并且灵活适配的日志基础框架，包括：

1. 对Shard进行弹性伸缩。
2. 消费者上线与下线能够对Shard自动适配，过程中数据不丢失。过程中支持保序。
3. 过程中不重复（需要消费者配合）。
4. 观察到消费进度，以便合理调配计算资源。
5. 支持更多渠道日志接入（对银行而言开通网上银行、手机银行、支票等渠道，可以接入更多的用户请求）。

通过LogHub + LogHub Consumer Library 能够帮助您解决日志实时处理中的这些经典问题，只需把精力放在业务逻辑上，而不用去担心流量扩容、Failover等琐事。

另外，Storm、Spark Streaming已经通过Consumer Library实现了对应的接口，欢迎使用。日志服务的更多技术资讯和讨论，请查看[日志服务的主页](#)和[日志处理圈子](#)。

14.5 消费-通过ETL清洗数据

日志处理过程中的一个假设是：数据并不是完美的。在原始数据与最终结果之间有 Gap，需要通过 ETL（Extract Transformation Load）等手段进行清洗、转换与整理。

案例

“我要点外卖”是一个平台型电商网站，涉及用户、餐厅、配送员等。用户可以在网页、App、微信、支付宝等进行下单点菜；商家拿到订单后开始加工，并自动通知周围的外卖送货员；快递员将外卖送到用户手中。



运营小组有两个任务：

- 掌握外卖送货员的位置，定点调度。
- 掌握优惠券、现金的使用情况，定点投放优惠券进行互动运营。

送货员位置信息（GPS）数据加工

GPS 数据（X, Y）通过送货员 App 每分钟汇报一次，格式如下：

```
2016-06-26 19:00:15 ID:10015 DeviceID:EXX12345678 Network:4G GPS-X:10.30.339 GPS-Y:17.38.224.5 Status:Delivering
```

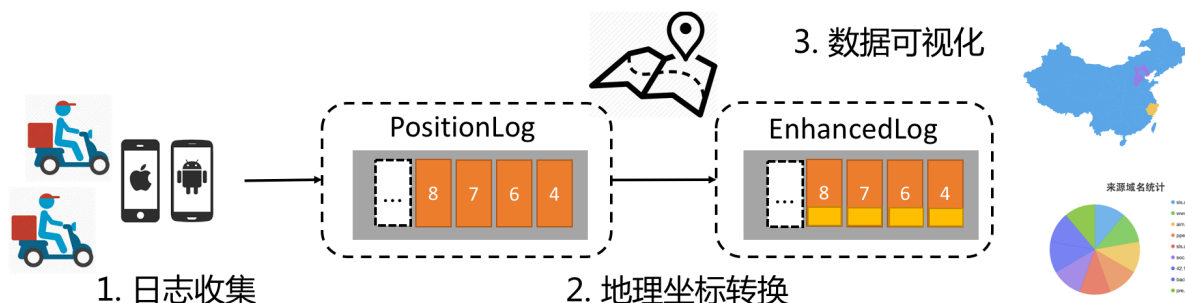
其中记录了上报时间，送货员 ID，使用网络，设备号，坐标位置 GPS-X，GPS-Y。GPS 给出的经纬度非常准确，但运营小组需要统计每个区域当前的状态，并不需要这么细的数据。因此，需要对原始数据做一次转换（Transformation），将坐标转换为可读的城市、区域、邮政编码等字段。

```
(GPS-X,GPS-Y) --> (GPS-X, GPS-Y, City, District, ZipCode)
```

这就是一个典型的 ETL 需求。使用 loghub 功能创建两个 logstore（PositionLog），以及转换后 logstore（EnhancedLog）。通过运行 ETL 程序（例如 Spark Streaming、Storm、或容器中启动 Consumer Library），实时订阅 PositionLog，对坐标进行转化，写入 EnhancedLog。可以对 EnhancedLog 数据机型仓库，连接实时计算进行可视化，或建立索引进行查询。

整个过程推荐架构如下：

1. 快递员 App 上埋点，每分钟汇报当前 GPS 位置一次，写入第一个 logstore（PositionLog）
 - 推荐使用 LogHub Android/iOS SDK、MAN（移动数据分析）将移动设备日志接入。
2. 通过实时程序实时订阅 PositionLog 数据，处理后实时写入 EnhancedLog Logstore。
 - 推荐使用 Spark Streaming、Storm 或 Consumer Lib（一种自动负载均衡的编程模式）或 SDK 订阅。
3. 对 Enhanced Log 进行处理，例如计算后进行数据可视化。推荐：
 - LogHub 对接流计算
 - LogShipper 投递（OSS, E-MapReduce, Table Store、MaxCompute）
 - LogSearch：订单查询等



支付订单脱敏与分析

支付服务接受支付请求，包括支付的账号、方式、金额、优惠券等。

- 其中包含部分敏感信息，需要进行脱敏。
- 需要对支付信息中优惠券、现金两个部分进行剥离。

整个过程如下：

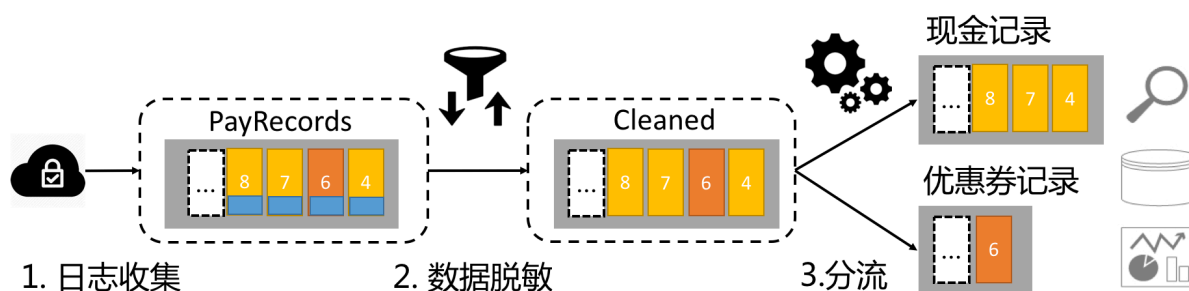
1. 创建 4 个 logstore，原始数据（PayRecords），脱敏后数据（Cleaned），现金订单（Cash），代金券（Coupon）。应用程序通过 Log4J Appender 向原数据 logstore（PayRecords）写入订单数据。

推荐使用 Log4J Appender，敏感数据不落盘。

2. 脱敏程序实时消费 PayRecords，将账号相关信息剥离后，写入 Cleaned Logstore。

分流程程序实时消费 Cleaned Logstore，通过业务逻辑把优惠券、现金两个部分分别存入对账相关的 logstore（Cash、Coupon）进行后续处理。

实时脱敏、分流程程序推荐使用 Spark Streaming、Storm 或 Consumer Lib（一种自动负载均衡的编程模式）或 SDK 订阅。



其他

- loghub 功能下每个 logstore 可以通过 RAM 进行账号级权限控制。
- loghub 当前读写可以通过监控获得，消费进度可以通过控制台查看。