ALIBABA CLOUD

阿里云

日志服务

实时消费

文档版本：20220629

阿里云

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
危险	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。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会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，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	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。	单击**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。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

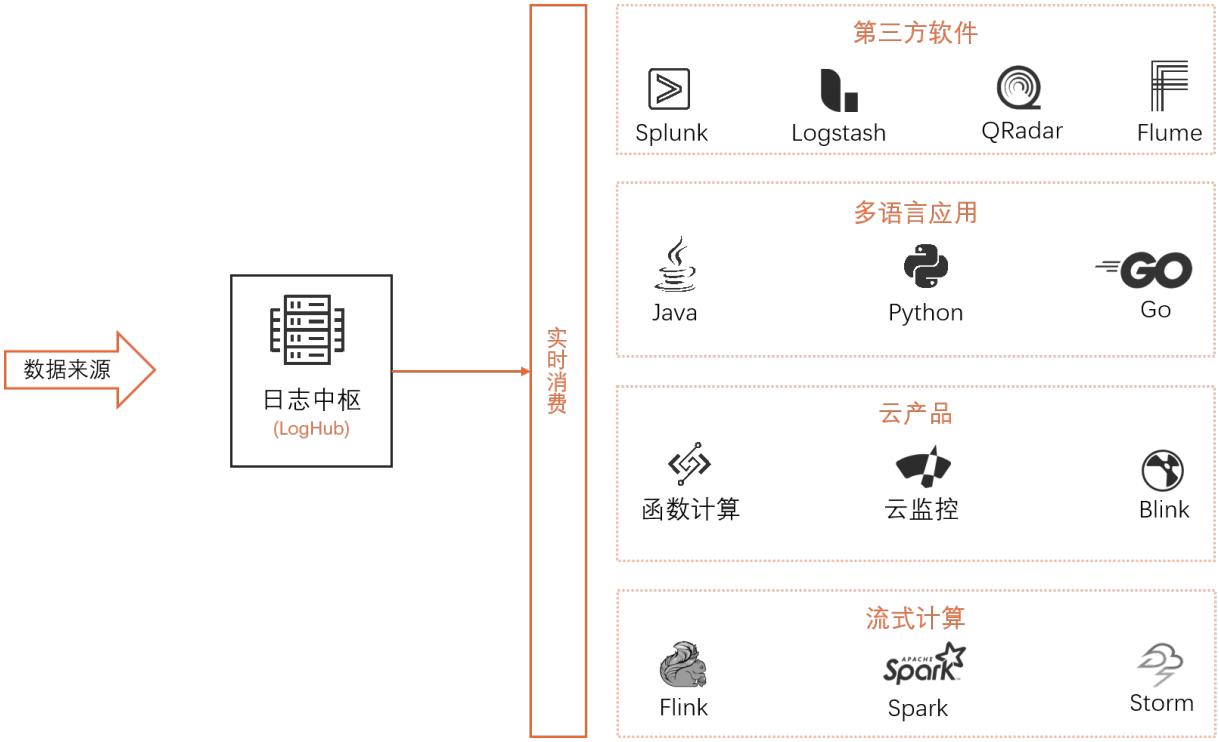
目录

1.实时消费概述

日志服务提供数据实时消费功能，支持通过SDK实时消费数据。本文介绍实时消费功能的概念、功能优势、应用场景、计费规则、消费目标等信息。

实时消费

实时消费是指第三方软件、多语言应用、云产品、流式计算框架等通过SDK实时消费日志服务的数据。实时消费是对全量数据的顺序读写，类似于消息中间件Kafka的功能。



> **说明** 实时消费和查询与分析都是读取数据。关于两者的区别，请参见日志消费与查询区别。

应用场景

实时消费适用于流计算、实时计算等场景。实时消费的实时性较强，通常为秒级。您可以自定义存储时间。

功能优势

实时消费具有以下优势：

- 数据集中

  日志服务已完成不同机器上的数据集中化，您只需通过SDK实时消费采集到日志服务的数据。

- 分类管理

  您可以利用日志服务的数据分类管理功能，实现不同的应用和产品实时消费不同项目、不同类型的数据。

计费规则

实时消费涉及多个计费项，包括读写流量、请求费用等。更多信息，请参见计费项。

消费目标

日志服务支持的实时消费目标如下表所示。

类型	目标	说明
第三方软件	Splunk	您可以通过Splunk实时消费日志服务的数据。具体操作，请参见阿里云日志服务Splunk Add-on。
	Flume	您可以通过Flume实时消费日志服务的数据。具体操作，请参见Flume消费。
	Logstash	您可以通过Logstash实时消费日志服务的数据。具体操作，请参见Logstash消费。
	QRadar	QRadar等安全信息与事件管理系统可以通过HTTPS协议或Syslog协议实时消费日志服务的数据。具体操作，请参见通过HTTPS投递日志到SIEM和通过Syslog投递日志到SIEM。
多语言应用	多语言应用	Java、Python、Go等语言的应用作为消费者或消费组消费日志服务的数据。具体操作，请参见普通消费和通过消费组消费日志数据。
云产品	函数计算	您可以通过函数计算实时消费日志服务的数据。具体操作，请参见通过函数计算消费日志数据。
	Blink	您可以通过实时计算实时消费日志服务的数据。具体操作，请参见实时计算（Blink）消费。
	云监控	您可以通过云监控实时消费日志服务的数据。具体操作，请参见云监控消费。
流式计算	Storm	您可以通过流式计算框架Storm实时消费日志服务的数据。具体操作，请参见Storm消费。
	Flink	您可以通过流式计算框架Flink实时消费日志服务的数据。具体操作，请参见Flink消费。
	Spark	您可以通过流式计算框架Spark实时消费日志服务的数据。具体操作，请参见Spark Streaming消费。

2.多语言应用
2.1. 普通消费

日志服务提供多语言SDK，且都支持日志服务消费接口。本文介绍普通消费日志的SDK示例及控制台的消费预览功能。

SDK消费

本示例中，创建一个PullLogsDemo.java文件，并调用接口读取日志数据，完成普通消费的演示。示例如下：

```
import com.aliyun.openservices.log.Client;
import com.aliyun.openservices.log.common.Consts;
import com.aliyun.openservices.log.common.LogGroupData;
import com.aliyun.openservices.log.exception.LogException;
import com.aliyun.openservices.log.request.PullLogsRequest;
import com.aliyun.openservices.log.response.PullLogsResponse;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.TimeUnit;
public class PullLogsDemo {
    //日志服务的服务入口。更多信息，请参见服务入口。此处以杭州为例，其它地域请根据实际情况填写。
    private static final String endpoint = "cn-hangzhou.log.aliyuncs.com";
    //阿里云访问密钥AccessKey。更多信息，请参见访问密钥。阿里云账号AccessKey拥有所有API的访问权限，风险很高。强烈建议您创建并使用RAM用户进行API访问或日常运维。
    private static final String accessKeyId = "your_access_id";
    private static final String accessKeySecret = "your_access_key";
    //Project名称。
    private static final String project = "your_project";
    //Logstore名称。
    private static final String logStore = "your_logstore";
    public static void main(String[] args) throws Exception{
        //创建日志服务Client。
        Client client = new Client(endpoint, accessKeyId, accessKeySecret);
        //查询Logstore的Shard数量。
        int shardCount = client.GetLogStore(project, logStore).GetLogStore().GetShardCount();
        System.out.println(String.format("%s has %d shards", logStore, shardCount));
        // 从头开始消费，获取游标。（如果是从尾部开始消费，使用Consts.CursorMode.END）
        List<String> cursors = new ArrayList<String>(shardCount);
        for (int i = 0; i < shardCount; i++) {
            cursors.add(i, client.GetCursor(project, logStore, i, Consts.CursorMode.BEGIN).GetC
ursor());
        }
        // 按照Shard进行日志消费。
        try{
            do {
                for (int i = 0; i < shardCount; i++) {
                    // 从每个Shard中获取日志。
                    PullLogsRequest request = new PullLogsRequest(project, logStore, i, 1000, c
ursors.get(i));
                    PullLogsResponse response = client.pullLogs(request);
                    // 日志都在日志组（LogGroup）中，按照逻辑拆分即可。
                    List<LogGroupData> logGroups = response.getLogGroups();
                    System.out.println(String.format("Get %d logGroup from logStore:%s:\tShard:
%d", logGroups.size(), logStore, i));
                    // 打印完成后，移动游标。
```

```
                    // 打印完成后，移动游标。
                    cursors.set(i, response.getNextCursor());
                }
                TimeUnit.SECONDS.sleep(3);
            } while (true);
        }catch (LogException e) {
            System.out.println("error code :" + e.GetErrorCode());
            System.out.println("error message :" + e.GetErrorMessage());
            throw e;
        }
    }
}
```

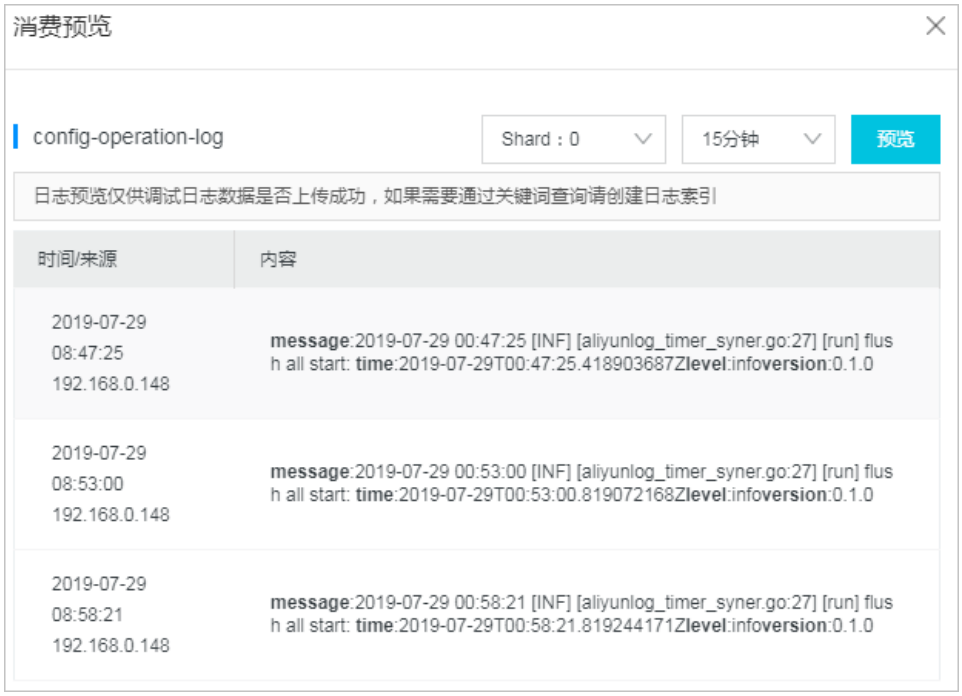
更多关于日志服务SDK的信息请参见日志服务SDK参考。

消费预览

消费预览也是一种日志消费。日志服务控制台提供消费预览功能，帮助您通过控制台直接预览Logstore中的部分日志数据。

1.
2. 在Project列表区域，单击目标Project。
3. 在**日志存储 > 日志库**页签中，单击目标Logstore。
4. 选择目标Logstore右侧的 > **消费预览**。
5. 在**消费预览**页面，选择指定预览的Shard与时间段，单击**预览**。

   消费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。



2.2. 消费组消费

2.2.1. 通过消费组消费日志数据

通过消费组（ConsumerGroup）消费日志数据有显著优点，您无需关注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。

前提条件

已安装SDK代码开发环境。更多信息，请参见SDK参考。

基本概念

概念	说明
消费组	日志服务支持通过消费组消费数据。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中的日志数据，消费者之间不会重复消费数据。每个Logstore最多创建30个消费组。
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。
Logstore	日志库是日志服务中日志数据的采集、存储和查询单元。更多信息，请参见日志库（Logstore）。
Shard	分区用于控制Logstore的读写能力，数据必定保存在某一个Shard中。每个Shard均有范围，为MD5左闭右开区间。每个区间范围不会相互覆盖，并且所有的区间的范围是MD5整个取值范围[00000000000000000000000000000000,ffffffffffffffffffffffffffffffff）。更多信息，请参见分区（Shard）。
Checkpoint	消费位点，是程序当前消费到的最新位置，程序重启之后可以从Checkpoint恢复消费进度。

在日志服务中，一个Logstore下面会有多个Shard，通过消费组消费日志数据就是将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。

通过消费组消费可以保存Checkpoint，在程序故障恢复时能从断点继续消费，从而保证数据不会被重复消费。

操作步骤

可以通过Java、C++、Python及Go语言实现消费组消费日志，以下操作步骤以Java为例。

1. 添加Maven依赖。

   在Java项目的根目录下，打开*pom.xml*文件，添加以下代码：

```
<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>0.6.33</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，请您根据实际情况填写。更多信息，请参见服务入口。
    private static String sEndpoint = "cn-hangzhou.log.aliyuncs.com";
    // 日志服务项目名称，请您根据实际情况填写。
    private static String sProject = "ali-cn-hangzhou-sls-admin";
    // 日志库名称，请您根据实际情况填写。
    private static String sLogstore = "sls_operation_log";
    // 消费组名称，请您根据实际情况填写。
    private static String sConsumerGroup = "consumerGroupX";
    // 消费数据的用户AccessKey ID和AccessKey Secret信息，请您根据实际情况填写。更多信息，请参见访问密钥。
    private static String sAccessKeyId = "";
    private static String sAccessKey = "";
    public static void main(String[] args) throws LogHubClientWorkerException, InterruptedException {
        // consumer_1是消费者名称，同一个消费组下面的消费者名称必须不同，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，此时消费者名称可以使用机器IP地址来区分。
        // maxFetchLogGroupSize用于设置每次从服务端获取的LogGroup最大数目，使用默认值即可。您可以使用config.setMaxFetchLogGroupSize(100);调整，请注意取值范围为(0,1000]。
        LogHubConfig config = new LogHubConfig(sConsumerGroup, "consumer_1", sEndpoint, sProject, sLogstore, sAccessKeyId, sAccessKey, LogHubConfig.ConsumePosition.BEGIN_CURSOR,1000);
        ClientWorker worker = new ClientWorker(new SampleLogHubProcessorFactory(), config);
        Thread thread = new Thread(worker);
        //Thread运行之后，ClientWorker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任务，Shutdown完成后请等待还在执行的任务安全退出，建议sleep配置为30秒。
        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，消费时的所有异常都需要处理，不能直接抛出。
    public String process(List<LogGroupData> logGroups,
                          ILogHubCheckPointTracker checkPointTracker) {
        // 打印已获取的数据。
        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.get
Value()));
            }
            for (int lIdx = 0; lIdx < flg.getLogsCount(); ++lIdx) {
                FastLog log = flg.getLogs(lIdx);
                System.out.println("--------\nLog: " + lIdx + ", time: " + log.getTime() +
", GetContentCount: " + log.getContentsCount());
                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秒内发生Worker异常终止，新启动的Worker会从上一
个Checkpoint获取消费数据，可能存在少量的重复数据。
        if (curTime - mLastCheckTime > 30 * 1000) {
            try {
                //参数为true表示立即将Checkpoint更新到服务端；false表示将Checkpoint缓存在本地，默
认间隔60秒会将Checkpoint更新到服务端。
                checkPointTracker.saveCheckPoint(true);
            } catch (LogHubCheckPointException e) {
                e.printStackTrace();
            }
            mLastCheckTime = curTime;
        }
        return null;
    }
    // 当Worker退出时，会调用该函数，您可以在此处执行清理工作。
    public void shutdown(ILogHubCheckPointTracker checkPointTracker) {
        //将Checkpoint立即保存到服务端。
        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();
    }
}
```

更多信息，请参见Java、C++、Python及Go。

4. 运行Main.java。
以模拟消费Nginx日志为例，打印日志如下：

```
:    GET
request_uri    :    /request/path-3/file-7
status    :    200
body_bytes_sent    :    3820
host    :    www.example.com
request_time    :    43
request_length    :    1987
http_user_agent    :    Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/41.0.2228.0 Safari/537.36
http_referer    :    www.example.com
http_x_forwarded_for    :    192.168.10.196
upstream_response_time    :    0.02
--------
Log: 158, time: 1635629778, GetContentCount: 14
......
    category    :    null
    source    :    127.0.0.1
    topic    :    nginx_access_log
    machineUUID    :    null
Tags
    __receive_time__    :    1635629815
--------
Log: 0, time: 1635629788, GetContentCount: 14
......
    category    :    null
    source    :    127.0.0.1
    topic    :    nginx_access_log
    machineUUID    :    null
Tags
    __receive_time__    :    1635629877
--------
......
```

通过控制台查看消费组状态

1.
2. 在Project列表区域，单击目标Project。
3. 单击目标Logstore左侧的>图标，选择**数据消费**。
4. 单击目标消费组，即可查看每个Shard消费数据的进度。

通过代码查看消费组状态

您也可以通过SDK查看消费进度。以Java SDK为例，运行ConsumerGroupTest.java，代码如下：

```
import java.util.List;
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Logstore下的所有消费组，若消费组不存在，则长度为0。
        List<ConsumerGroup> consumerGroups = client.ListConsumerGroup(project, logstore).GetCon
sumerGroups();
        for(ConsumerGroup c: consumerGroups){
            //打印消费组的属性，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.getC
onsumerGroupName()).GetCheckPoints()){
                System.out.println("shard: " + cp.getShard());
                //时间返回精确到微秒，类型为长整型。
                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，输出时请注意格式化。
                    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(), CursorMod
e.END).GetCursor();
                int endPrg = 0;
                try{
                    endPrg = client.GetPrevCursorTime(project, logstore, cp.getShard(), endCurs
or).GetCursorTime();
                }
                catch(LogException e){
                    //do nothing
                }
                //Unix时间戳，单位是秒，输出时请注意格式化。
                System.out.println("最后一条数据到达时刻： " + endPrg);
```

```
                System.out.println("最后一条数据到达时刻: " + endTime);
            }
        }
    }
}
```

打印返回以下结果：

```
名称： etl-6cac01c571d5a4b933649c04a7ba215b
心跳超时时间： 60
按序消费： false
shard: 0
最后一次消费数据的时间： 1639555453575211
消费者名称： etl-356464787983a3d17086a9797e3d5f0e6959b066-256521
消费进度： 1639555453
最后一条数据到达时刻： 1639555453
shard: 1
最后一次消费数据的时间： 1639555392071328
消费者名称： etl-356464787983a3d17086a9797e3d5f0e6959b066-256521
消费进度： 1639555391
最后一条数据到达时刻： 1639555391
名称： etl-2bd3fdfdd63595d56b1ac24393bf5991
心跳超时时间： 60
按序消费： false
shard: 0
最后一次消费数据的时间： 1639555453256773
消费者名称： etl-532719dd2f198b3878ee3c6dfc80aeffb39ee48b-061390
消费进度： 1639555453
最后一条数据到达时刻： 1639555453
shard: 1
最后一次消费数据的时间： 1639555392066234
消费者名称： etl-532719dd2f198b3878ee3c6dfc80aeffb39ee48b-061390
消费进度： 1639555391
最后一条数据到达时刻： 1639555391
名称： consumerGroupX
心跳超时时间： 60
按序消费： false
shard: 0
最后一次消费数据的时间： 1639555434142879
消费者名称： consumer_1
消费进度： 1635615029
最后一条数据到达时刻： 1639555453
shard: 1
最后一次消费数据的时间： 1639555437976929
消费者名称： consumer_1
消费进度： 1635616802
最后一条数据到达时刻： 1639555391
```

相关操作

- 异常诊断

  建议您为消费者程序配置Log4j，将消费组内部遇到的异常信息打印出来，便于定位。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
```

配置Log4j后，执行消费者程序可以看到类似如下异常信息：

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

- 通过消费组消费从某个时间开始的日志数据

```
// consumerStartTimeInSeconds表示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，LogHub
Config.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);
```

> **说明**
>
> - 按照消费需求，请您使用不同的构造方法。
> - 当服务端已保存Checkpoint，则开始消费位置以服务端保存的Checkpoint为准。
> - 日志服务消费数据时，默认优先使用Checkpoint作为消费点。当您指定从固定时间点开始消费数据时，必须保证consumerStartTimeInSeconds时间点落到TTL周期内，否则会造成消费不生效。

- 重置Checkpoint

```
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);
        }
    }
```

RAM用户授权

使用RAM用户访问需要设置消费组相关的权限策略。更多信息，请参见步骤二：为RAM用户授权。

授权的Action如下：

Action	Description	Resource
log:GetCursorOrData（GetCursor，PullLogs）	根据时间获取游标（cursor）。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*
log:CreateConsumerGroup	在指定的Logstore上创建一个消费组。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*/consumergroup/*${consumerGroupName}*
log:ListConsumerGroup	查询指定Logstore的所有消费组。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*/consumergroup/*
log:ConsumerGroupUpdateCheckPoint	更新指定消费组的某个Shard的Checkpoint。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*/consumergroup/*${consumerGroupName}*
log:ConsumerGroupHeartBeat	为指定消费者发送心跳到服务端。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*/consumergroup/*${consumerGroupName}*
log:UpdateConsumerGroup	修改指定消费组属性。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*/consumergroup/*${consumerGroupName}*
log:GetConsumerGroupCheckPoint	获取指定消费组消费的某个或者所有Shard的Checkpoint。	acs:log:*${regionName}*:*${projectOwnerAliUid}*:project/*${projectName}*/logstore/*${logstoreName}*/consumergroup/*${consumerGroupName}*

例如，Project所在地域为cn-hangzhou，Project名称为project-test，消费的Logstore名称为logstore-test，Project的所属阿里云账号ID为174649****602745，消费组名称为consumergroup-test，则需要为RAM用户设置以下权限。

```
{
  "Version": "1",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "log:GetCursorOrData"
      ],
      "Resource": "acs:log:cn-hangzhou:174649****602745:project/project-test/logstore/logstore-
test"
    },
    {
      "Effect": "Allow",
      "Action": [
        "log:CreateConsumerGroup",
        "log:ListConsumerGroup"
      ],
      "Resource": "acs:log:cn-hangzhou:174649****602745:project/project-test/logstore/logstore-
test/consumergroup/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "log:ConsumerGroupUpdateCheckPoint",
        "log:ConsumerGroupHeartBeat",
        "log:UpdateConsumerGroup",
        "log:GetConsumerGroupCheckPoint"
      ],
      "Resource": "acs:log:cn-hangzhou:174649****602745:project/project-test/logstore/logstore-
test/consumergroup/consumergroup-test"
    }
  ]
}
```

2.2.2. 消费组监控与告警

您可以通过日志服务控制台查看日志消费的状态并设置告警。

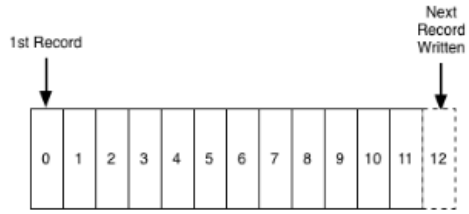
前提条件

已开通服务日志中的**重要日志**。具体操作，请参见开通服务日志。

背景信息

一个消费组包含多个消费者，每个消费者消费Logstore中的一部分Shard。同一个消费组下面的消费者共同消费一个Logstore中的日志数据，消费者之间不会重复消费数据。

Shard数据模型可以简单理解为一个队列，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。



消费组消费延迟告警中的基本概念：

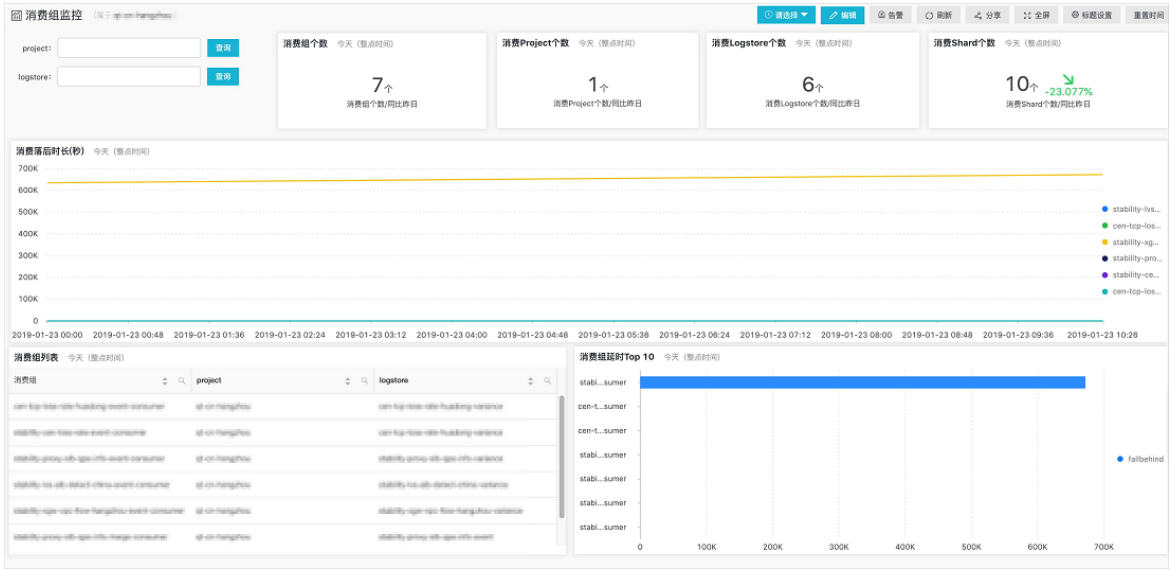
- 消费过程：消费者从队头开始顺序读取数据的过程。
- 消费进度：消费者当前读取的数据对应的写入时间。
- 消费落后时长：当前消费进度和队列中最新的数据写入时间的差值，单位为秒。

消费组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，当超过您预设的阈值时，则认定消费落后太多，触发报警。

操作步骤

1. 登录日志服务控制台。
2. 在Project列表，单击目标Project。
3. 在左侧导航栏，单击**仪表盘**。
4. 在**仪表盘**列表，单击**消费组监控**。

   在仪表盘中查看消费状态，包括消费组延时Top10、消费落后时长、消费组列表等。



5. （可选）设置告警规则。
   i. 在**消费组监控**页面右上角，选择**告警 > 创建新版告警**。
   ii. 在**告警规则**面板，设置告警规则，然后单击**确定**。

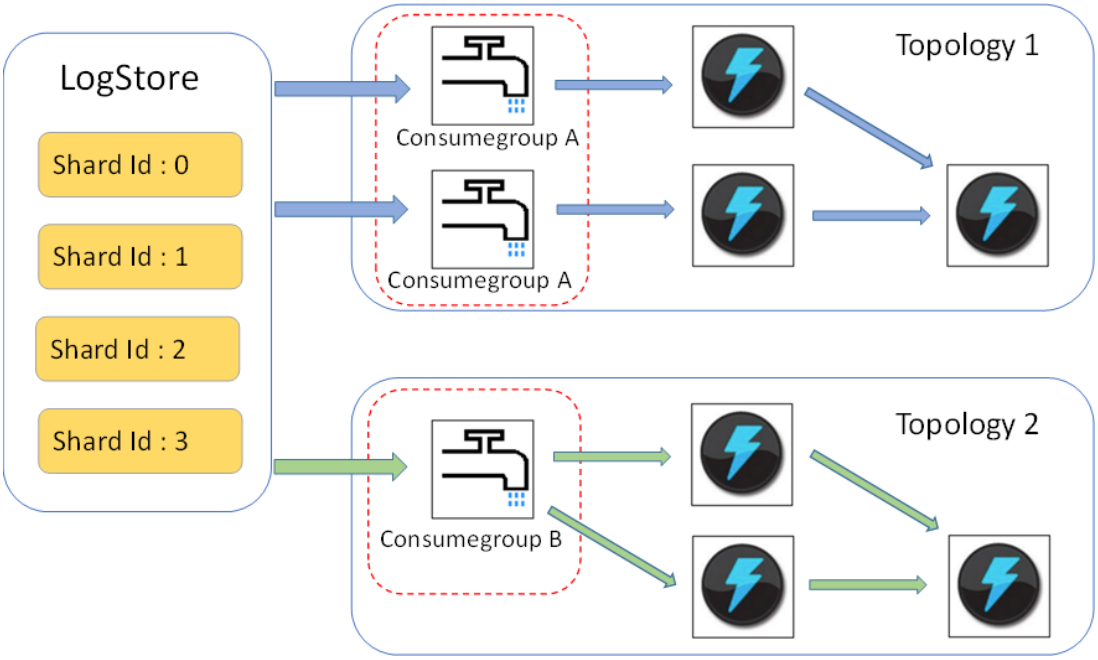
   告警规则设置，请参见设置告警。

3.流式计算
3.1. Storm消费

日志服务LogHub提供了高效、可靠的日志通道功能，您可以通过Logtail、SDK等多种方式来实时采集日志数据。采集日志数据之后，可以通过Storm实时消费写入到日志服务中的数据。为了降低Storm消费的代价，日志服务提供了LogHub Storm Spout来实时读取日志数据。

基本结构和流程

基本结构和流程



- 上图中红色虚线框中是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Storm Topology需要选择唯一的消费组名称来相互标识，同一Topology内的Spout通过消费组消费日志数据来完成负载均衡和自动Failover，详情请参见通过消费组消费日志数据。
- Spout从LogHub中实时读取数据之后，发送至Storm Topology中的Bolt节点，定期将消费完成位置作为Checkpoint保存到LogHub服务端。

> **说明**
>
> - 为了防止滥用，每个Logstore最多支持10个消费组，对于不再使用的消费组，可以调用DeleteConsumerGroup接口删除。
> - 建议Spout的个数和Shard个数相同，否则可能会导致单个Spout处理数据量过多而影响性能。
> - 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。

示例

- Spout使用示例，用于构建Storm Topology。

```
     public static void main( String[] args )
    {
        String mode = "Local";  //使用本地测试模式。
           String consumer_group_name = "";   //每个Topology需要设定唯一的消费组名称。
        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(consumer_group_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LogGroupData的序列化方法LogGroupData
SerializSerializer。
        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);
            } catch (InterruptedException e) {
                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.createTopol
ogy());
            } catch (AlreadyAliveException e) {
                e.printStackTrace();
            } catch (InvalidTopologyException e) {
                e.printStackTrace();
            }
        } else {
            logger.error("invalid mode: " + mode);
        }
    }
}
```

- 消费数据的bolt代码示例，打印每条日志的内容。

```
public class SampleBolt extends BaseRichBolt {
    private static final long serialVersionUID = 4752656887774402264L;
    private static final Logger logger = Logger.getLogger(BaseBasicBolt.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(Lo
gHubSpout.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
    }
}
```

添加Maven依赖

- Storm 1.0版本及之前版本（如0.9.6版本），请使用：

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

- storm 1.0版本及之后版本，请使用：

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

3.2. Flink消费

Flink Log Connector是日志服务提供的用于对接Flink的工具，支持对接开源Flink和实时计算Flink版。本文介绍如何对接Flink消费日志数据。

前提条件

- 已获取Access Key。更多信息，请参见访问密钥。
- 已创建Project和Logstore。更多信息，请参见创建Project和创建Logstore。
- 已为RAM用户授权。更多信息，请参见配置指定Logstore的消费权限。

背景信息

Flink Log Connector包括两部分，消费者（Flink Log Consumer）和生产者（Flink Log Producer），两者用途区别如下：

- 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Flink Log Connector时，需要在项目中添加Maven依赖，示例如下：

```
<dependency>
    <groupId>com.aliyun.openservices</groupId>
    <artifactId>flink-log-connector</artifactId>
    <version>0.1.31</version>
</dependency>
<dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>2.5.0</version>
</dependency>
```

除此之外的代码编写，请您参考GitHub上源码进行编写。更多信息，请参见aliyun-log-flink-connector。

Flink Log Consumer

在Flink Log Connector中，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，在使用时您无需关心Logstore中Shard数量的变化，Flink Log Consumer会自动感知。

Flink中每一个子任务负责消费Logstore中的部分Shard，如果Logstore中Shard发生分裂或合并，子任务消费的Shard也会随之改变。

Flink Log Consumer用到的日志服务API接口如下：

- GetCursorOrData

  用于从Shard中获取数据，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限额，可以通过ConfigConstants.LOG_FETCH_DATA_INTERVAL_MILLIS和ConfigConstants.LOG_MAX_NUMBER_PER_FETCH控制接口调用的时间间隔和每次调用获取的日志数量。Shard的限额请参见分区（Shard）。

  示例如下：

```
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列表及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。示例如下：

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

- CreateConsumerGroup

  当设置消费进度监控时调用该接口创建ConsumerGroup，用于同步Checkpoint。

- UpdateCheckPoint

  该接口将Flink的snapshot同步到日志服务的ConsumerGroup中。

1. 配置启动参数。

   以下是一个简单的消费示例，使用java.util.Properties作为配置工具，所有Flink Log Consumer的配置均在ConfigConstants中。

```
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");
// 设置用户AccessKey ID及AccessKey Secret。
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Shard，如果少于子任务数量，那么部分子任务就会空闲，直到新的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，就可以定制从Shard的头、尾或者某个特定时间开始消费。另外，Flink Log Connector也支持从某个具体的消费组中恢复消费。属性的具体取值如下：

   - 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：表示从某个特定的消费组中保存的Checkpoint开始消费，通过ConfigConstants.LOG_CONSUMERGROUP指定具体的消费组。
   - 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。

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Flink Log 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，获取每一个Shard的实时消费位置，使用时间戳表示。更多信息，请参见通过控制台查看消费组状态和消费组监控与告警。

   示例如下：

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

   > **说明** 该项为可选配置项，设置后Flink Log Consumer会首先创建消费组，如果消费组已经存在，则不执行任何操作，Flink Log Consumer中的snapshot会自动同步到日志服务的消费组中，您可以通过日志服务的控制台查看Flink Log 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，当任务失败时，Flink会恢复消费任务，并从保存的最新的Checkpoint开始消费。

   Checkpoint的周期定义了当任务失败时，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的信息请参见Checkpoints。

Flink Log Producer

Flink Log Producer用于将数据写入日志服务。

> **说明** Flink Log Producer只支持Flink at least once语义，在任务失败时，写入日志服务中的数据有可能会重复，但不会丢失。

Flink Log Producer用到的日志服务API接口如下：

- PutLogs
- ListShards

1. 初始化Flink Log Producer。

i. 初始化配置参数Properties。

Flink Log Producer初始化步骤与Flink Log Consumer类似。Flink Log Producer初始化配置包含以下参数，一般情况下使用默认值即可，如有需要可以自定义配置。示例如下：

```
// 用于发送数据的I/O线程的数量，默认是8。
ConfigConstants.LOG_SENDER_IO_THREAD_COUNT
// 该值定义日志数据被缓存发送的时间，默认是3000。
ConfigConstants.LOG_PACKAGE_TIMEOUT_MILLIS
// 缓存发送的包中日志的数量，默认是4096。
ConfigConstants.LOG_LOGS_COUNT_PER_PACKAGE
// 缓存发送的包的大小，默认是3 MB。
ConfigConstants.LOG_LOGS_BYTES_PER_PACKAGE
// 作业可以使用的内存总的大小，默认是100 MB。
ConfigConstants.LOG_MEM_POOL_BYTES
```

ii. 重载LogSerializationSchema，定义将数据序列化成RawLogGroup的方法。

RawLogGroup是日志的集合，各字段含义请参见日志（Log）。

如果您需要指定数据写到某一个Shard中，可以使用LogPartitioner产生数据的HashKey，LogPartitioner为可选项，如果您没有配置，数据会随机写入某一个Shard。

示例如下：

```
FlinkLogProducer<String> logProducer = new FlinkLogProducer<String>(new SimpleLogSerial
izer(), configProps);
logProducer.setCustomPartitioner(new LogPartitioner<String>() {
      // 生成32位Hash值。
      public String getHashKey(String element) {
          try {
              MessageDigest md = MessageDigest.getInstance("MD5");
              md.update(element.getBytes());
              String hash = new BigInteger(1, md.digest()).toString(16);
              while(hash.length() < 32) hash = "0" + hash;
              return hash;
          } catch (NoSuchAlgorithmException e) {
          }
          return  "00000000000000000000000000000000000000000000000000000000000000000";
      }
  });
```

2. 将模拟产生的字符串写入日志服务，示例如下：

```
// 将数据序列化成日志服务的数据格式。
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 = StreamExecutionEnvironment.getExecutionEnvir
onment();
        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);
        // 设置用户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 SimpleLogSe
rializer(), 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;
        }
    }
}
```

3.3. Spark Streaming消费

日志服务采集到日志数据后，您可以通过运行Spark Streaming任务消费日志数据。

日志服务提供的Spark SDK实现了Receiver模式和Direct模式两种消费模式。Maven依赖如下：

```
<dependency>
  <groupId>com.aliyun.emr</groupId>
  <artifactId>emr-logservice_2.11</artifactId>
  <version>1.7.2</version>
</dependency>
```

Receiver模式

Receiver模式通过消费组消费日志数据并暂存在Spark Executor，Spark Streaming任务启动之后从Executor读取并处理数据。每条数据以JSON字符串的形式返回。消费组自动定时保存Checkpoint到服务端，无需手动更新Checkpoint。更多信息，请参见通过消费组消费日志数据。

- 参数说明

参数	类型	说明
project	String	日志服务Project名称。
logstore	String	日志服务Logstore名称。
consumerGroup	String	消费组名称。
endpoint	String	日志服务Project所在地域的服务入口。更多信息，服务入口。
accessKeyId	String	访问日志服务的AccessKey ID。
accessKeySecret	String	访问日志服务的AccessKey Secret。

- 示例

> **说明** 默认配置下，Receiver模式在异常情况下可能导致数据丢失。为了避免此类情况发生，建议开启Write-Ahead Logs开关（Spark 1.2以上版本支持）。更多关于Write-Ahead Logs的细节请参见Spark。

```
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.aliyun.logservice.LoghubUtils
import org.apache.spark.streaming.{Milliseconds, StreamingContext}
import org.apache.spark.SparkConf
object TestLoghub {
  def main(args: Array[String]): Unit = {
    if (args.length < 7) {
      System.err.println(
        """Usage: TestLoghub <project> <logstore> <loghub group name> <endpoint>
          |         <access key id> <access key secret> <batch interval seconds>
        """.stripMargin)
      System.exit(1)
    }
    val project = args(0)
    val logstore = args(1)
    val consumerGroup = args(2)
    val endpoint = args(3)
    val accessKeyId = args(4)
    val accessKeySecret = args(5)
    val batchInterval = Milliseconds(args(6).toInt * 1000)
    def functionToCreateContext(): StreamingContext = {
      val conf = new SparkConf().setAppName("Test Loghub")
      val ssc = new StreamingContext(conf, batchInterval)
      val loghubStream = LoghubUtils.createStream(
        ssc,
        project,
        logstore,
        consumerGroup,
        endpoint,
        accessKeyId,
        accessKeySecret,
        StorageLevel.MEMORY_AND_DISK)
      loghubStream.checkpoint(batchInterval * 2).foreachRDD(rdd =>
        rdd.map(bytes => new String(bytes)).top(10).foreach(println)
      )
      ssc.checkpoint("hdfs:///tmp/spark/streaming") // set checkpoint directory
      ssc
    }
    val ssc = StreamingContext.getOrCreate("hdfs:///tmp/spark/streaming", functionToCreateCon
text _)
    ssc.start()
    ssc.awaitTermination()
  }
}
```

Direct模式

Direct模式不需要消费组，使用API在任务运行时直接从服务端请求数据。Direct模式具有如下优势：

- 简化并行：Spark partition个数与Logstore Shard总数一致。只需分裂Shard即可提高任务的并行度。
- 高效：不需要开启Write-Ahead Logs来保证数据不丢失。
- Exactly-once语义：直接从服务端按需获取数据，任务成功之后再提交Checkpoint。

  由于Spark异常退出或者其他原因导致任务未正常结束，可能会导致部分数据被重复消费。

Direct模式需要依赖ZooKeeper环境，用于临时保存中间状态。同时，必须设置Checkpoint目录。中间状态数据保存在ZooKeeper内对应的Checkpoint目录内。如果任务重启之后希望重新消费，需要在ZooKeeper内删除该目录，并更改消费组名称。

- 参数说明

参数	类型	说明
project	String	日志服务Project名称。
logstore	String	日志服务Logstore名称。
consumerGroup	String	消费组名称，仅用于保存消费位置。
endpoint	String	日志服务Project所在地域的服务入口。更多信息，请参见服务入口。
accessKeyId	String	访问日志服务的Access Key ID。
accessKeySecret	String	访问日志服务的Access Key Secret。
zkAddress	String	ZooKeeper的连接地址。

- 限流配置

  Spark Streaming流式消费是将数据分成微小的批次进行处理，因此日志服务开始消费时，需预知每个批次的边界，即每个批次需要获取的数据条数。

  日志服务底层的存储模型以LogGroup为单位，正常情况下每个LogGroup对应一次写入请求，例如一次写入请求可能包含数千条日志，这些日志作为一个LogGroup进行存储和消费。而通过WebTracking方式写入日志时，每次写入请求仅包含一条日志，即一个LogGroup只有一条日志。为了能够应对不同写入场景的消费需求，SDK提供如下两个参数进行限流配置。

参数	说明	默认值
spark.loghub.batchGet.step	限制单次消费请求获取的最大LogGroup个数。	100
spark.streaming.loghub.maxRatePerShard	限制单批次内每个Shard消费的日志条数。	10000

  通过spark.streaming.loghub.maxRatePerShard可指定每个批次每个Shard期望消费的最大日志条数。Spark SDK的实现原理是每次从服务端获取spark.loghub.batchGet.step中的LogGroup个数并累计其中的日志条数，直到达到或超过spark.streaming.loghub.maxRatePerShard。因此spark.streaming.loghub.maxRatePerShard并非一个精确控制单批次消费日志条数的参数，实际上每个批次消费的日志条数与spark.loghub.batchGet.step以及每个LogGroup中包含的日志条数相关。

- 示例

```
import com.aliyun.openservices.loghub.client.config.LogHubCursorPosition
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Milliseconds, StreamingContext}
import org.apache.spark.streaming.aliyun.logservice.{CanCommitOffsets, LoghubUtils}
object TestDirectLoghub {
  def main(args: Array[String]): Unit = {
    if (args.length < 7) {
      System.err.println(
        """Usage: TestDirectLoghub <project> <logstore> <loghub group name> <endpoint>
          |         <access key id> <access key secret> <batch interval seconds> <zookeeper h
ost:port=localhost:2181>
        """.stripMargin)
      System.exit(1)
    }
    val project = args(0)
    val logstore = args(1)
    val consumerGroup = args(2)
    val endpoint = args(3)
    val accessKeyId = args(4)
    val accessKeySecret = args(5)
    val batchInterval = Milliseconds(args(6).toInt * 1000)
    val zkAddress = if (args.length >= 8) args(7) else "localhost:2181"
    def functionToCreateContext(): StreamingContext = {
      val conf = new SparkConf().setAppName("Test Direct Loghub")
      val ssc = new StreamingContext(conf, batchInterval)
      val zkParas = Map("zookeeper.connect" -> zkAddress,
        "enable.auto.commit" -> "false")
      val loghubStream = LoghubUtils.createDirectStream(
        ssc,
        project,
        logstore,
        consumerGroup,
        accessKeyId,
        accessKeySecret,
        endpoint,
        zkParas,
        LogHubCursorPosition.END_CURSOR)
      loghubStream.checkpoint(batchInterval).foreachRDD(rdd => {
        println(s"count by key: ${rdd.map(s => {
          s.sorted
          (s.length, s)
        }).countByKey().size}")
        loghubStream.asInstanceOf[CanCommitOffsets].commitAsync()
      })
      ssc.checkpoint("hdfs:///tmp/spark/streaming") // set checkpoint directory
      ssc
    }
    val ssc = StreamingContext.getOrCreate("hdfs:///tmp/spark/streaming", functionToCreateCon
text _)
    ssc.start()
    ssc.awaitTermination()
  }
}
```

更多信息，请参见GitHub。

4.云产品

4.1. 函数计算消费

4.1.1. 通过函数计算消费日志数据

依托阿里云函数计算服务，日志服务提供流式数据加工服务。您可以通过配置一个函数计算触发器作业，定时获取更新的数据并触发函数的执行，进而增量消费日志服务Logstore的数据，并完成自定义加工任务。日志服务提供的函数模板或者用户自定义函数均可作为数据加工函数。

前提条件

- 已为日志服务触发函数执行授权。更多信息，请参见云资源访问授权。
- 已创建Project和Logstore。更多信息，请参见创建Project和Logstore。

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、查询及分析。

- 数据投递场景

  为数据的目的端提供支撑，构建云上大数据产品之间的数据管道。

数据加工函数

- 函数类型
  - 模板函数

    更多信息，请参见aliyun-log-fc-functions。
  - 用户自定义函数

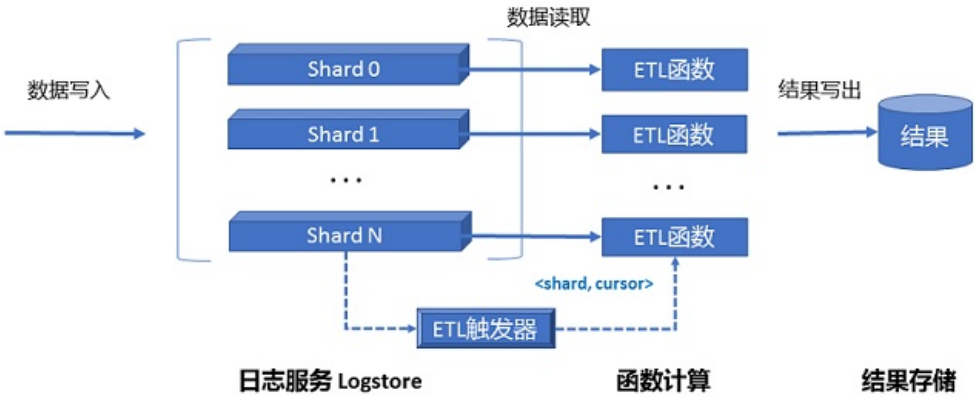
    函数配置的格式与函数的具体实现有关。更多信息，请参见ETL函数开发指南。

- 函数计算触发机制

  函数计算触发器作业对应于函数计算的一个触发器，当创建函数计算触发器作业后，日志服务会根据该触发器作业的配置启动定时器，定时器轮询Logstore中的Shard信息，当发现有新的数据写入时，即生成 `<shard_id,begin_cursor,end_cursor >` 三元组信息作为函数Event，并触发函数执行。

  > **说明** 当存储系统升级时，即使没有新数据写入，也可能发生Cursor变化，在这种情况下，每个Shard会额外空触发一次。针对这种情况，您可以在函数内通过Cursor尝试获取Shard的数据，如果获取不到数据说明是一次空触发，可以在函数内做忽略处理。更多信息，请参见自定义函数开发指南。

  函数计算触发器作业触发机制是时间触发。例如：您设置的函数计算触发器作业触发间隔为60秒，Logstore的Shard0一直有数据写入，那么Shard0每60秒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Shard0数据进行下一步处理。



操作步骤

1. 创建函数计算服务。

   具体操作，请参见创建服务。

   > **说明** 日志服务Project所在地域和函数计算服务所在地域必须一致。

2. 创建函数。

   具体操作，请参见创建函数。

3. 创建触发器。

   具体操作，请参见配置SLS触发器。

相关操作

- 查询触发器日志

  您可以为触发器日志Logstore创建索引，查看任务执行统计结果。更多信息，请参见配置索引。

- 查看函数运行日志

  您可以通过命令行工具，查看函数执行过程的详细信息。更多信息，请参见查看调用日志。

常见问题

当您创建触发器后但未触发函数执行，可以从以下两个方面排查。

- 确认函数计算触发器作业配置的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。

- 查看触发器日志、函数运行日志查看是否有异常。

4.1.2. 自定义函数开发指南

当您通过函数计算消费日志数据时，在不同场景您可以选择使用日志服务提供的函数模板或自定义函数。本文介绍如何构建一个自定义函数。

函数Event

函数Event即运行函数计算的入口参数，格式为一个JSON Object序列化后字符串。

- 参数说明

参数	说明
jobName	日志服务ETL Job名称。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	对于一个ETL Job，taskId是某一次确定性的函数调用标识。
cursorTime	本次函数调用包括的数据中，最后一条日志到达日志服务的服务器端的unix_timestamp。
source	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中的重要组成部分，定义了本次函数调用的消费范围。 ○ endpoint：Project所属地域的服务入口，详情请参见服务入口。 ○ 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。 ○ 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	JSON Object类型，在您创建触发器**函数配置**时设置。自定义日志服务ETL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。详情请参见配置SLS触发器。

- 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，您可以调用GetCursor接口获取cursor并按上述示例构建一个函数Event用于测试。

函数开发

您可以通过Java、Python、Node.js等多种语言实现函数开发，日志服务提供了相应Runtime的SDK，以便您在函数中进行集成，详情请参见SDK参考。

以下内容以Java 8 Runtime为例，介绍如何开发日志服务ETL函数。关于Java 8函数编程细节，详情请参见函数计算服务Java编程指南。

- Java函数模板

  目前，日志服务提供了基于Java8 Runtime的自定义数模板，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Log Service Java SDK获取数据，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，通过 `UserDefinedFunctionParameter.java` 实现。
  - 函数内针对数据的自定义业务逻辑，通过 `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接口实现

  您可以在processData内完成对一批数据的消费、加工、投递。例如在LogstoreReplication中，实现了将数据从一个Logstore中读出后写到另一个Logstore。

  > **说明**
  > - processData处理数据成功则返回true，处理数据遇到异常无法重试成功则返回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，会忽略其中处理异常的数据。
  > - 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，会通过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。
  > - 如果Shard流量较大，请为函数配置足够的内存规格，以避免函数OOM导致异常终止。
  > - 如果在函数内执行耗时操作或者Shard流量较大，请您设置较短的函数触发间隔和较长的函数运行超时时间。
  > - 请您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

  调度日志记录ETL任务开始时间、结束时间、任务是否成功以及成功返回的信息。如果ETL任务出错会生成ETL出错日志，并向系统管理员发送报警邮件或短信。请您在创建触发器时设置触发器日志Logstore，并为该Logstore开启并配置索引，详情请参见配置索引。

  对于函数执行结果的统计，可以通过函数返回，例如在Java8 Runtime函数的 `outputStream` 中体现。日志服务提供的函数模板会写出一个JSON Object序列化后的字符串，该字符串将记录在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信息，包括某一步骤的开始和结束时间、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()` 记录过程日志并存放在日志服务指定Project的Logstore中，建议您为该Logstore开启索引查询功能。

4.2. 实时计算（Blink）消费

阿里云实时计算通过创建日志服务源表的方式，可以直接消费日志服务中的数据。本文介绍了如何为实时计算创建日志服务源表以及创建过程涉及到的属性字段提取和类型映射。

创建日志服务源表

日志服务是实时数据存储，实时计算能将其作为流式数据输入。假设有如下日志内容：

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

实时计算日志服务源表DDL示例如下：

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

WITH参数说明：

参数	是否必须	说明
endPoint	是	日志服务Endpoint，详情请参见服务入口。
accessId	是	访问日志服务的AccessKey ID。
accessKey	是	访问日志服务的密钥AccessKey Secret。
project	是	日志服务Project名称。
logStore	是	日志服务Logstore名称。

参数	是否必须	说明
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Blink的版本是1.4.2版本及以上版本，则默认为100条，最大1000条。 **说明** 如果单条日志的大小和batchGetSize都很大，可能会导致Java系统频繁的对内存数据进行垃圾回收。
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。

属性字段提取

除日志字段外，支持提取如下三个属性字段，也支持提取其它自定义字段。

属性字段	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，示例如下：

```
create table sls_stream(
  __timestamp__ bigint HEADER,
  __receive_time__ bigint HEADER
  b int,
  c varchar
) with (
  type ='sls',
  endPoint ='<your endpoint>',
  accessId ='<your AccessKey ID>',
  accessKey ='<your AccessKey Secret>',
  startTime = '2017-07-05 00:00:00',
  project ='ali-cloud-streamtest',
  logStore ='stream-test',
  consumerGroup ='consumerGroupTest1'
);
```

类型映射

日志服务字段类型和实时计算字段类型对应关系如下，建议您使用该对应关系进行DDL声明。

日志服务字段类型	实时计算字段类型
STRING	VARCHAR

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

注意事项

- Blink 2.2.0版本及之前的版本不支持Shard的扩容和缩容，如果分裂或者缩容了某个正在实时计算读取的Logstore，会导致任务持续出错且不可恢复，这种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。
- Blink 1.6.0版本及之前版本，在Shard数目很多的情况下指定消费组可能会影响读取性能。
- 日志服务暂不支持Map类型的数据。
- 不存在的字段默认为null。
- 字段顺序支持无序，建议您在设置时字段顺序和表中参数顺序一致。
- 如果一个Shard没有新数据写入，会导致任务的整体延迟增加。在这种情况下，只需要把并发数调整为读写的Shard数量即可。
- 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，建议您用本地DEBUG方法，上线运行后打印到日志中查看。

4.3. 云监控消费

云监控可以直接消费Logstore中日志数据，提供日志监控功能。

通过将日志服务与云监控结合，形成了轻量级、全面、易用的日志监控解决方案。

- 简单、易用，相比传统ELK方案，零编码即可享有完整监控解决方案。
- 提供日志数据实时分析、监控图表展示、报警服务的全套解决方案。
- 基于阿里云Apsara Monitor服务，提供稳定可靠的日志监控体验。
- 全SaaS服务，几乎无时间成本、人力成本和运维成本，让您快速拥有企业级业务日志实时监控能力。

创建日志监控操作步骤请参见管理日志监控项。

5.第三方软件

5.1. 简介

日志服务支持将日志投递到SIEM，以确保阿里云上的所有法规、审计与其他相关日志能够导入到您的安全运维中心（SOC）中。

名词解释

- SIEM：安全信息与事件管理系统（Security Information and Event Management），如Splunk，IBM QRadar等。
- Splunk HEC：Splunk的HTTP事件接收器（Splunk HTTP Event Collector），一个 HTTP或HTTPS接口，用于接收日志。

部署建议

- 硬件参数：
  - Linux（如Ubuntu x64）操作系统。
  - 2.0+ GHZ X 8核。
  - 16 GB内存，推荐32GB。
  - 1 Gbps网卡。
  - 至少2 GB可用磁盘空间，建议10 GB以上。
- 网络参数：

  从组织内的环境到阿里云的带宽应该大于数据在阿里云端产生的速度，否则日志无法实时消费。假设数据产生速度均匀，峰值为平时的2倍左右，每天产生1 TB原始日志。数据传输为5倍压缩的场景下，推荐带宽应该在4 MB/s（32 Mbps）左右。
- Python语言：我们用Python语言进行消费。Java 语言用法，请参考通过消费组消费日志数据。

Python SDK

- 推荐使用标准CPython解释器。
- 日志服务的Python SDK可以使用**python3 -m pip install aliyun-log-python-sdk -U**命令进行安装。
- 更多日志服务Python SDK的使用，请参见User Guide。

协同消费库

协同消费库（Consumer Library）是日志服务中对日志进行消费的高级模式，提供了消费组的概念对消费端进行抽象和管理，和直接使用SDK进行数据读取的区别在于，用户无需关心日志服务的实现细节，只需要专注于业务逻辑，另外，消费者之间的负载均衡、failover 等用户也都无需关心。

在日志服务中，一个日志库（Logstore）下面会有多个分区（Shard），协同消费库将 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，这个消费组的 shard 从属关系会调整，以达到消费负载均衡的目的，但是上面的分配原则不会变，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。

Spark Streaming、Storm 以及 Flink Connector 都以 Consumer Library 作为基础实现。

投递方式

推荐使用日志服务消费组构建程序实现实时消费，然后通过HTTPS或者Syslog来发送日志给SIEM。

- HTTPS投递方式请参考通过HTTPS投递日志到SIEM。
- Syslog投递方式请参考通过Syslog投递日志到SIEM。

5.2. 通过HTTPS投递日志到SIEM

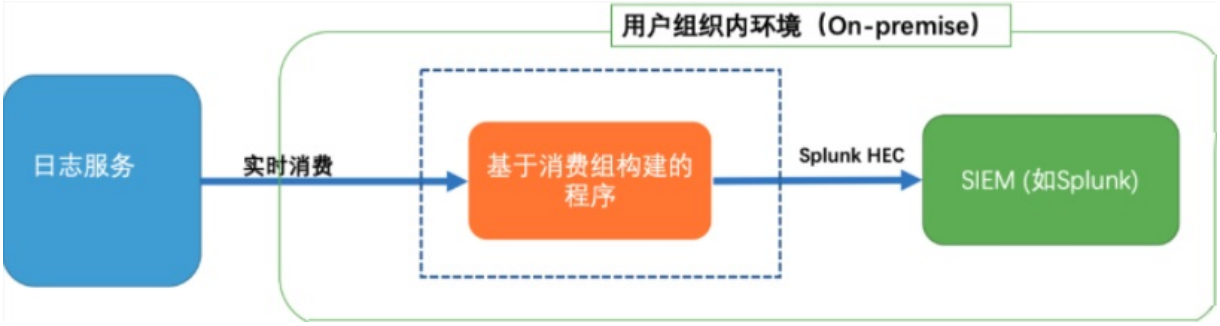
本文通过以Splunk HEC为例来展示如何通过HEC将阿里云相关日志投递到SIEM。

例如当前SIEM（如Splunk）位于组织内部环境（on-premise），而不是云端。为了安全考虑，没有开放任何端口让外界环境来访问此SIEM。

> **说明** 本章节中的配置代码仅为示例，最新的代码示例请参见Github或Github（多源日志库时）。

投递流程

推荐使用日志服务消费组构建程序进行实时消费，然后通过Splunk API（HEC）发送日志给Splunk。



主程序示例

如下代码展示主程序控制逻辑。

```
def main():
    option, settings = get_monitor_option()
    logger.info("*** start to consume data...")
    worker = ConsumerWorker(SyncData, option, args=(settings,) )
    worker.start(join=True)
if __name__ == '__main__':
    main()
```

程序配置示例

- 配置内容
  - 程序日志文件：以便后续测试或者诊断问题。
  - 基本配置项：包括日志服务连接配置和消费组配置。
  - 消费组的高级选项：性能调参，不推荐修改。
  - SIEM（Splunk）相关参数与选项。
- 代码示例

  请仔细阅读代码中相关注释并根据业务需求调整选项。

```
#encoding: utf8
```

```
import os
import logging
from logging.handlers import RotatingFileHandler
root = logging.getLogger()
handler = RotatingFileHandler("{0}_{1}.log".format(os.path.basename(__file__), current_proces
s().pid), maxBytes=100*1024*1024, backupCount=5)
handler.setFormatter(logging.Formatter(fmt='[%(asctime)s] - [%(threadName)s] - {%(module)s:%(
funcName)s:%(lineno)d} %(levelname)s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S'))
root.setLevel(logging.INFO)
root.addHandler(handler)
root.addHandler(logging.StreamHandler())
logger = logging.getLogger(__name__)
def get_option():
    ##########################
    # 基本选项
    ##########################
    # 从环境变量中加载日志服务参数与选项。
    endpoint = os.environ.get('SLS_ENDPOINT', '')
    accessKeyId = os.environ.get('SLS_AK_ID', '')
    accessKey = os.environ.get('SLS_AK_KEY', '')
    project = os.environ.get('SLS_PROJECT', '')
    logstore = os.environ.get('SLS_LOGSTORE', '')
    consumer_group = os.environ.get('SLS_CG', '')
    # 消费的起点。这个参数在首次运行程序的时候有效，后续再次运行时将从上一次消费的保存点继续消费。
    # 可以使用“begin”、“end”，或者特定的ISO时间格式。
    cursor_start_time = "2018-12-26 0:0:0"
    ##########################
    # 一些高级选项
    ##########################
    # 一般不建议修改消费者名称，尤其是需要进行并发消费时。
    consumer_name = "{0}-{1}".format(consumer_group, current_process().pid)
    # 心跳时长，当服务器在2倍时间内没有收到特定Shard的心跳报告时，服务器会认为对应消费者离线并重新调配任
务。
    # 所以当网络环境不佳时，不建议将时长设置的比较小。
    heartbeat_interval = 20
    # 消费数据的最大间隔，如果数据生成的速度很快，不需要调整这个参数。
    data_fetch_interval = 1
    # 构建一个消费组和消费者
    option = LogHubConfig(endpoint, accessKeyId, accessKey, project, logstore, consumer_group
, consumer_name,
                          cursor_position=CursorPosition.SPECIAL_TIMER_CURSOR,
                          cursor_start_time=cursor_start_time,
                          heartbeat_interval=heartbeat_interval,
                          data_fetch_interval=data_fetch_interval)
    # Splunk选项
    settings = {
                "host": "10.1.2.3",
                "port": 80,
                "token": "a023nsdu123123123",
                'https': False,             # 可选, bool
                'timeout': 120,             # 可选, int
                'ssl_verify': True,         # 可选, bool
                "sourcetype": "",           # 可选, sourcetype
                "index": "",                # 可选, index
                "source": "",               # 可选, source
            }
    return option, settings
```

消费与投递示例

如下代码展示如何从日志服务获取数据并投递到Splunk，请仔细阅读代码中相关注释并根据需求调整格式。

```
from aliyun.log.consumer import *
from aliyun.log.pulllog_response import PullLogResponse
from multiprocessing import current_process
import time
import json
import socket
import requests
class SyncData(ConsumerProcessorBase):
    """
    这个消费者从日志服务消费数据并发送给Splunk。
    """
    def __init__(self, splunk_setting):
      """初始化并验证Splunk连通性"""
        super(SyncData, self).__init__()
        assert splunk_setting, ValueError("You need to configure settings of remote target")
        assert isinstance(splunk_setting, dict), ValueError("The settings should be dict to inc
lude necessary address and confidentials.")
        self.option = splunk_setting
        self.timeout = self.option.get("timeout", 120)
        # 测试Splunk连通性
        s = socket.socket()
        s.settimeout(self.timeout)
        s.connect((self.option["host"], self.option['port']))
        self.r = requests.session()
        self.r.max_redirects = 1
        self.r.verify = self.option.get("ssl_verify", True)
        self.r.headers['Authorization'] = "Splunk {}".format(self.option['token'])
        self.url = "{0}://{1}:{2}/services/collector/event".format("http" if not self.option.ge
t('https') else "https", self.option['host'], self.option['port'])
        self.default_fields = {}
        if self.option.get("sourcetype"):
            self.default_fields['sourcetype'] = self.option.get("sourcetype")
        if self.option.get("source"):
            self.default_fields['source'] = self.option.get("source")
        if self.option.get("index"):
            self.default_fields['index'] = self.option.get("index")
    def process(self, log_groups, check_point_tracker):
        logs = PullLogResponse.loggroups_to_flattern_list(log_groups, time_as_str=True, decode_
bytes=True)
        logger.info("Get data from shard {0}, log count: {1}".format(self.shard_id, len(logs)))
        for log in logs:
            # 发送数据到Splunk
            event = {}
            event.update(self.default_fields)
            event['time'] = log[u'__time__']
            del log['__time__']
            json_topic = {"actiontrail_audit_event": ["event"] }
            topic = log.get("__topic__", "")
            if topic in json_topic:
                try:
                    for field in json_topic[topic]:
```

```
                        log[field] = json.loads(log[field])
                except Exception as ex:
                    pass
            event['event'] = json.dumps(log)
            data = json.dumps(event, sort_keys=True)
                try:
                    req = self.r.post(self.url, data=data, timeout=self.timeout)
                    req.raise_for_status()
                except Exception as err:
                    logger.debug("Failed to connect to remote Splunk server ({0}). Exception: {
1}", self.url, err)
                    # 根据需要，添加一些重试或者报告。
        logger.info("Complete send data to remote")
        self.save_checkpoint(check_point_tracker)
```

启动程序示例

例如程序命名为sync_data.py，启动程序示例如下所示。

```
export SLS_ENDPOINT=<Endpoint of your region>
export SLS_AK_ID=<YOUR AK ID>
export SLS_AK_KEY=<YOUR AK KEY>
export SLS_PROJECT=<SLS Project Name>
export SLS_LOGSTORE=<SLS Logstore Name>
export SLS_CG=<消费组名，可以简单命名为"syc_data">
python3 sync_data.py
```

多源日志库示例

针对多源日志库，需要共用一个executor以避免进程过多。更多信息，请参见多源日志库时发送日志到Splunk。核心的变化是主函数，示例如下所示。

```
exeuctor, options, settings = get_option()
    logger.info("*** start to consume data...")
    workers = []
    for option in options:
        worker = ConsumerWorker(SyncData, option, args=(settings,) )
        workers.append(worker)
        worker.start()
    try:
        for i, worker in enumerate(workers):
            while worker.is_alive():
                worker.join(timeout=60)
            logger.info("worker project: {0} logstore: {1} exit unexpected, try to shutdown it"
.format(
                options[i].project, options[i].logstore))
            worker.shutdown()
    except KeyboardInterrupt:
        logger.info("*** try to exit **** ")
        for worker in workers:
            worker.shutdown()
        # wait for all workers to shutdown before shutting down executor
        for worker in workers:
            while worker.is_alive():
                worker.join(timeout=60)
    exeuctor.shutdown()
```

限制与约束

每一个日志库（logstore）最多可以配置30个消费组，如果遇到 `ConsumerGroupQuotaExceed` 则表示超出限制，建议在控制台删除一些不再使用的消费组。

消费状态与监控

- 在控制台查看消费组状态，详情请参见通过控制台查看消费组状态。
- 通过云监控查看消费组延迟情况并配置告警，详情请参见消费组延迟。

并发消费

基于消费组的程序，可以直接启动多次程序以实现并发效果。

```
nohup python3 sync_data.py &
nohup python3 sync_data.py &
nohup python3 sync_data.py &
...
```

> **说明** 所有消费者的名称均不相同（消费者名以进程ID为后缀），且属于同一个消费组。因为一个分区（Shard）只能被一个消费者消费，例如一个日志库有10个分区，那么最多有10个消费组同时消费。

吞吐量

基于测试，在没有带宽、接收端速率限制（如Splunk端）的情况下，用python3运行上述样例，单个消费者大约占用20%的单核CPU资源，此时消费可以达到10 MB/s原始日志的速率。因此10个消费者理论上可以达到100 MB/s原始日志，即每个CPU核每天可以消费0.9 TB原始日志。

高可用

消费组将检测点（check-point）保存在服务器端，当一个消费者停止，另外一个消费者将自动接管并从断点继续消费。可以在不同机器上启动消费者，这样在一台机器停止或者损坏的情况下，其他机器上的消费者可以自动接管并从断点进行消费。为了备用，也可以通过不同机器启动大于Shard数量的消费者。

HTTPS

如果服务入口（Endpoint）配置为 `https://` 前缀，例如 `https://cn-beijing.log.aliyuncs.com`，则程序自动使用HTTPS加密与日志服务连接。

服务器证书*.aliyuncs.com是由GlobalSign签发，默认大多数Linux和Windows机器自动信任此证书。如果有机器不信任此证书，请参见Certificate installation下载并安装此证书。

5.3. 通过Syslog投递日志到SIEM

Syslog是一个常见的日志通道，几乎所有的SIEM（例如IBM Qradar, HP Arcsight）都支持通过Syslog渠道接收日志。本文主要介绍如何通过Syslog将日志服务中的日志投递到SIEM。

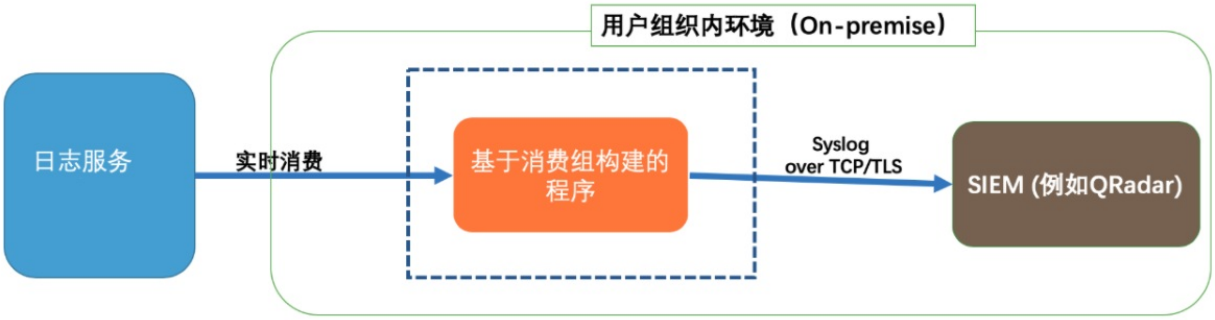
背景信息

- Syslog主要是基于RFC5424和RFC3164定义相关格式规范，RFC3164协议是2001年发布的，RFC5424协议是2009年发布的升级版本。因为新版兼容旧版，且新版本解决了很多问题，因此推荐使用RFC5424协议。更多信息，请参见RFC5424和RFC3164。
- Syslog over TCP/TLS：Syslog只规定日志格式，理论上TCP和UDP都支持Syslog，可以较好的保证数据传输稳定性。RFC5425协议也定义了TLS的安全传输层，如果您的SIEM支持TCP通道或者TLS通道，则建议优先使用。更多信息，请参见RFC5425。
- Syslog facility：早期Unix定义的程序组件，此处选择 `user` 作为默认组件。更多信息，请参见程序组件。
- Syslog severity：定义日志级别，您可以根据需求设置指定内容的日志为较高的级别。默认一般用 `info`。更多信息，请参见日志级别。

> **说明** 本文中的配置代码仅为示例，最新最全的代码示例请参见Github。

投递流程

推荐使用日志服务消费组构建程序来进行实时消费，然后通过Syslog over TCP/TLS来发送日志给SIEM。



主程序示例

如下代码展示主程序控制逻辑。

```
def main():
    option, settings = get_monitor_option()
    logger.info("*** start to consume data...")
    worker = ConsumerWorker(SyncData, option, args=(settings,) )
    worker.start(join=True)
if __name__ == '__main__':
    main()
```

程序配置示例

- 配置内容：
  - 程序日志文件：以便后续测试或者诊断问题。
  - 基本配置项：包括日志服务连接配置和消费组配置。
  - 消费组的高级选项：性能调参，不推荐修改。
  - SIEM的Syslog server相关参数与选项。

> **说明** 如果SIEM支持基于TCP或者TLS的Syslog通道，则需要配置proto为TLS及配置正确的SSL的证书。

- 代码示例

  请仔细阅读代码中相关注释并根据业务需求调整选项。

```
#encoding: utf8
import os
import logging
from logging.handlers import RotatingFileHandler
root = logging.getLogger()
handler = RotatingFileHandler("{0}_{1}.log".format(os.path.basename(__file__), current_proces
s().pid), maxBytes=100*1024*1024, backupCount=5)
handler.setFormatter(logging.Formatter(fmt='[%(asctime)s] - [%(threadName)s] - {%(module)s:%(
funcName)s:%(lineno)d} %(levelname)s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S'))
root.setLevel(logging.INFO)
root.addHandler(handler)
root.addHandler(logging.StreamHandler())
logger = logging.getLogger(__name__)
def get_option():
    ##########################
    # 基本选项
    ##########################
    #从环境变量中加载日志服务参数与选项。
    endpoint = os.environ.get('SLS_ENDPOINT', '')
    accessKeyId = os.environ.get('SLS_AK_ID', '')
    accessKey = os.environ.get('SLS_AK_KEY', '')
    project = os.environ.get('SLS_PROJECT', '')
    logstore = os.environ.get('SLS_LOGSTORE', '')
    consumer_group = os.environ.get('SLS_CG', '')
    # 消费的起点。这个参数在首次运行程序的时候有效，后续再次运行时将从上一次消费的保存点继续消费。
    # 可以使用“begin”、“end”，或者特定的ISO时间格式。
    cursor_start_time = "2018-12-26 0:0:0"
    ##########################
    # 高级选项
    ##########################
    # 一般不要修改消费者名称，尤其是需要并发消费时。
```

```
    consumer_name = "{0}-{1}".format(consumer_group, current_process().pid)
    # 心跳时长，当服务器在2倍时间内没有收到特定Shard的心跳报告时，服务器会认为对应消费者离线并重新调配任务。
    # 所以当网络环境不佳的时候，不建议将时长设置的比较小。
    heartbeat_interval = 20
    # 消费数据的最大间隔，如果数据生成的速度很快，不需要调整这个参数。
    data_fetch_interval = 1
    # 构建一个消费组和消费者
    option = LogHubConfig(endpoint, accessKeyId, accessKey, project, logstore, consumer_group, consumer_name,
                          cursor_position=CursorPosition.SPECIAL_TIMER_CURSOR,
                          cursor_start_time=cursor_start_time,
                          heartbeat_interval=heartbeat_interval,
                          data_fetch_interval=data_fetch_interval)
    # syslog options
    settings = {
                "host": "1.2.3.4", # 必选
                "port": 514,       # 必选，端口
                "protocol": "tcp", # 必选，TCP、UDP或TLS（仅Python3）。
                "sep": "||",       # 必选，key=value键值对的分隔符，这里用双竖线（||）分隔。
                "cert_path": None, # 可选，TLS的证书位置。
                "timeout": 120,    # 可选，超时时间，默认120秒。
                "facility": syslogclient.FAC_USER,  #可选，可以参考其他syslogclient.FAC_*的值。
                "severity": syslogclient.SEV_INFO,  #可选，可以参考其他syslogclient.SEV_*的值。
                "hostname": None,  # 可选，机器名，默认选择本机机器名。
                "tag": None        # 可选，标签，默认是短划线（-）。
            }
    return option, settings
```

消费与投递示例

如下代码展示如何从日志服务获取数据投递到SIEM Syslog服务器。请仔细阅读代码中相关注释并根据需求调整格式。

```
from syslogclient import SyslogClientRFC5424 as SyslogClient
class SyncData(ConsumerProcessorBase):
    """
    消费者从日志服务消费数据并发送给Syslog server。
    """
    def __init__(self, splunk_setting):
      """初始化并验证Syslog server连通性。"""
        super(SyncData, self).__init__()   # remember to call base's init
        assert target_setting, ValueError("You need to configure settings of remote target")
        assert isinstance(target_setting, dict), ValueError("The settings should be dict to include necessary address and confidentials.")
        self.option = target_setting
        self.protocol = self.option['protocol']
        self.timeout = int(self.option.get('timeout', 120))
        self.sep = self.option.get('sep', "||")
        self.host = self.option["host"]
        self.port = int(self.option.get('port', 514))
        self.cert_path=self.option.get('cert_path', None)
        # try connection
        with SyslogClient(self.host, self.port, proto=self.protocol, timeout=self.timeout, cert_path=self.cert_path) as client:
            pass
    def process(self, log_groups, check_point_tracker):
```

```
    def process(self, log_groups, check_point_tracker):
        logs = PullLogResponse.loggroups_to_flattern_list(log_groups, time_as_str=True, decode_
bytes=True)
        logger.info("Get data from shard {0}, log count: {1}".format(self.shard_id, len(logs)))
        try:
            with SyslogClient(self.host, self.port, proto=self.protocol, timeout=self.timeout,
cert_path=self.cert_path) as client:
                for log in logs:
                    # Put your sync code here to send to remote.
                    # the format of log is just a dict with example as below (Note, all strings
are unicode):
                    #    Python2: {"__time__": "12312312", "__topic__": "topic", u"field1": u"v
alue1", u"field2": u"value2"}
                    #    Python3: {"__time__": "12312312", "__topic__": "topic", "field1": "val
ue1", "field2": "value2"}
                    # suppose we only care about audit log
                    timestamp = datetime.fromtimestamp(int(log[u'__time__']))
                    del log['__time__']
                    io = six.StringIO()
                    first = True
          # 可以根据需要修改格式化内容，这里使用Key=Value传输，并使用默认的双竖线（||）进行分割。
                    for k, v in six.iteritems(log):
                        io.write("{0}{1}={2}".format(self.sep, k, v))
                    data = io.getvalue()
          # 可以根据需要修改facility或者severity。
                    client.log(data, facility=self.option.get("facility", None), severity=self.
option.get("severity", None), timestamp=timestamp, program=self.option.get("tag", None), hostna
me=self.option.get("hostname", None))
        except Exception as err:
            logger.debug("Failed to connect to remote syslog server ({0}). Exception: {1}".form
at(self.option, err))
            # 需要添加一些错误处理的代码，例如重试或者通知等。
            raise err
        logger.info("Complete send data to remote")
        self.save_checkpoint(check_point_tracker)
```

启动程序示例

例如程序命名为sync_data.py，启动程序示例如下所示。

```
export SLS_ENDPOINT=<Endpoint of your region>
export SLS_AK_ID=<YOUR AK ID>
export SLS_AK_KEY=<YOUR AK KEY>
export SLS_PROJECT=<SLS Project Name>
export SLS_LOGSTORE=<SLS Logstore Name>
export SLS_CG=<消费组名，可以简单命名为"syc_data">
python3 sync_data.py
```

限制与约束

每一个日志库（logstore）最多可以配置30个消费组，如果遇到 `ConsumerGroupQuotaExceed` 则表示超出限制，建议在控制台删除一些不再使用的消费组。

消费状态与监控

- 在控制台查看消费组状态，详情请参见通过控制台查看消费组状态。

- 通过云监控查看消费组延迟情况并配置告警，详情请参见消费组延迟。

并发消费

基于消费组的程序，可以直接启动多次程序以实现并发效果。

```
nohup python3 sync_data.py &
nohup python3 sync_data.py &
nohup python3 sync_data.py &
...
```

> **说明** 所有消费者的名称均不相同（消费者名以进程ID为后缀），且属于同一个消费组。因为一个分区（Shard）只能被一个消费者消费，例如一个日志库有10个分区，那么最多有10个消费组同时消费。

吞吐量

基于测试，在没有带宽、接收端速率限制（如Splunk端）的情况下，用python3运行上述样例，单个消费者大约占用20%的单核CPU资源，此时消费可以达到10 MB/s原始日志的速率。因此10个消费者理论上可以达到100 MB/s原始日志，即每个CPU核每天可以消费0.9 TB原始日志。

高可用

消费组将检测点（check-point）保存在服务器端，当一个消费者停止，另外一个消费者将自动接管并从断点继续消费。可以在不同机器上启动消费者，这样在一台机器停止或者损坏的情况下，其他机器上的消费者可以自动接管并从断点进行消费。为了备用，也可以通过不同机器启动大于Shard数量的消费者。

5.4. 阿里云日志服务Splunk Add-on

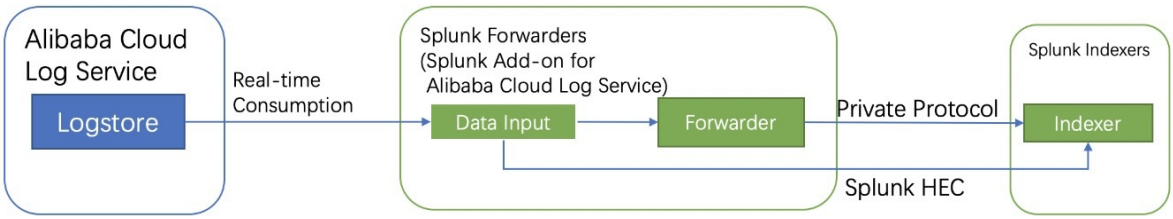
本文介绍如何通过阿里云日志服务Splunk Add-on将日志服务上的日志投递到Splunk。

架构

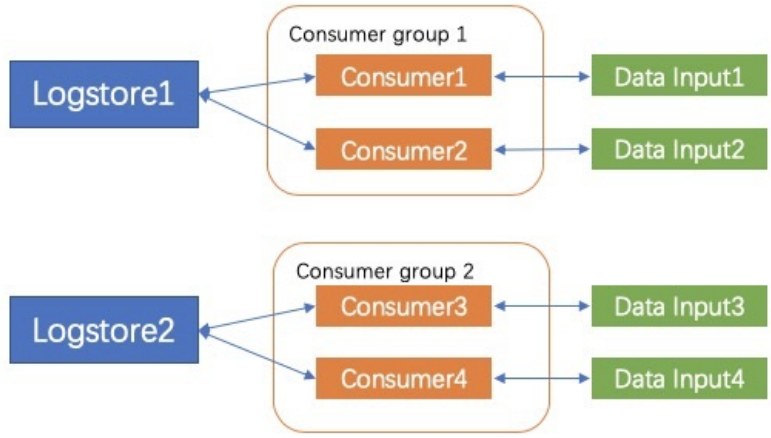
Splunk Add-on架构：

- 通过Splunk Data Input创建日志服务消费组，并从日志服务进行实时日志消费。
- 将采集到的日志通过Splunk私有协议（Private Protocol）或者HTTP Event Collector（HEC）投递到Splunk indexer。

> **说明** 此Add-on仅用于采集数据，只需要在Splunk Heavy Forwarder上安装，不需要在Indexer和Search Head上安装。



机制



- 一个Data Input相当于一个消费者，对日志进行消费。
- 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。
- 一个Logstore下面会有多个Shard。
  - 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。
- 所创建的消费者名字是由消费组名、host名、进程名、event协议类型组合而成，保证同一消费组内的消费者不重名。

更多关于消费组的信息，请参见通过消费组消费日志数据。

准备工作

- 获取日志服务的AccessKey。

  您可以通过阿里云RAM获取日志服务Project的AccessKey。更多信息，请参见访问密钥和通过访问密钥访问数据。

  您可以通过权限助手配置RAM权限。更多信息，请参见配置权限助手。常用的RAM配置如下：

  > **说明** <Project name>为您的日志服务Project名称，<Logstore name>为您的日志服务Logstore名称，请根据实际情况替换，名字替换支持通配符*。

```
{
  "Version": "1",
  "Statement": [
    {
      "Action": [
        "log:ListShards",
        "log:GetCursorOrData",
        "log:GetConsumerGroupCheckPoint",
        "log:UpdateConsumerGroup",
        "log:ConsumerGroupHeartBeat",
        "log:ConsumerGroupUpdateCheckPoint",
        "log:ListConsumerGroup",
        "log:CreateConsumerGroup"
      ],
      "Resource": [
        "acs:log:*:*:project/<Project name>/logstore/<Logstore name>",
        "acs:log:*:*:project/<Project name>/logstore/<Logstore name>/*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

- 检查Splunk版本及运行环境。
  - 确保使用最新的Add-on版本。
  - 操作系统：Linux、Mac OS、Windows。
  - Splunk版本：Splunk heavy forwarder 8.0及以上版本、Splunk indexer 7.0及以上版本。
- 配置Splunk HTTP Event Collector。更多信息，请参见Configure HTTP Event Collector on Splunk Enterprise。

  如果需要使用HEC来发送event，请确保HEC配置成功。如果选择Splunk私有协议，则可以跳过该步骤。

  > **说明** 目前创建Event Collector token时，不支持开启indexer acknowledgment功能。

安装Add-on

您可以登录Splunk web UI页面，通过以下两种方法安装Splunk Add-on。

> **说明** 此Add-on仅用于采集数据，只需要在Splunk Heavy Forwarder上安装，不需要在Indexer和Search Head上安装。

- 方法一
  i. 单击  图标。
  ii. 在**应用**页面，单击**浏览更多应用**。
  iii. 搜索**Alibaba Cloud Log Service Add-on for Splunk**，单击**安装**。
  iv. 安装完成后，根据页面提示重启 Splunk服务。
- 方法二
  i. 单击  图标。

ii. 在**应用**页面，单击**从文件安装应用**。

iii. 选择目标.tgz文件，单击**上载**。

您可以App Search Results上下载目标.tgz文件。

iv. 单击**安装**。

v. 安装完成后，根据页面提示重启 Splunk服务。

配置Add-on

当Splunk运行在非阿里云ECS上时，您可以使用阿里云访问密钥AccessKey ID和AccessKey Secret来访问日志服务。具体操作如下：

1. 在Splunk Web UI页面中，单击**Alibaba Cloud Log Service Add-on for Splunk**。
2. 配置全局账号。

   选择**配置 > Account**，在**Account**页签中，配置日志服务的AccessKey信息。

   > **说明** 此处配置的用户名、密码分别对应日志服务的AccessKey ID、AccessKey Secret。

3. 配置Add-on的运行日志级别。

   选择**配置 > Logging**，在**Logging**页签中，配置Add-on的运行日志级别。

4. 创建Data Input。

   i. 单击**输入**，进入**输入**页面。

   ii. 单击**Create New Input**，创建新的Data Input。

   Data input参数说明

参数	必选项 & 格式	描述	取值举例
名字	是，String	全局唯一的Data Input名。	无
间隔	是，Integer	Splunk Data Input退出后的重启时间，单位：s。	默认值：10s
索引	是，String	Splunk索引。	无
SLS AccessKey	是，String	阿里云访问密钥由AccessKey ID、AccessKey Secret组成。 **说明** 此处配置的用户名、密码分别对应日志服务的AccessKey ID、AccessKey Secret。	全局账号配置中配置的用户名和密码。

参数	必选项 & 格式	描述	取值举例
SLS endpoint	是，String	阿里云日志服务入口。更多信息，请参见服务入口。	■ cn-huhehaote.log.aliyuncs.com ■ https://cn-huhehaote.log.aliyuncs.com
SLS project	是，String	日志服务Project。更多信息，请参见管理Project。	无
SLS logstore	是，String	日志服务Logstore。更多信息，请参见管理Logstore。	无
SLS consumer group	是，String	日志服务消费组。扩容时，多个Data Input需要配置相同的消费组名称。更多信息，请参见通过消费组消费日志数据。	无
SLS cursor start time	是，String	消费起始时间。该参数只有消费组首次创建时有效。非首次创建日志都是从上次的保存点开始消费。 **说明** 这里的时间是日志到达时间。	取值：begin、end、ISO格式的时间（例如2018-12-26 0:0:0+8:00)。
SLS heartbeat interval	是，Integer	SLS消费者与Sever间的心跳间隔。单位：s。	默认值：60s
SLS data fetch interval	是，Integer	日志拉取间隔，如果日志频率较低，建议不要设的太小。单位：s。	默认值：1s
Topic filter	否，String	Topic过滤字符串，以;间隔区分多个过滤的Topic。如果日志的topic被命中，则忽略该日志，从而不能投递到Splunk。	“TopicA;TopicB”意味着topic为TopicA、TopicB的日志将被忽略。

参数	必选项 & 格式	描述	取值举例
Unfolded fields	否，JSON	Json格式的topic到字段列表的映射关系。{"topicA": ["field_nameA1", "field_nameA2", ...], "topicB": ["field_nameB1", "field_nameB2", ...], ...}	{"actiontrail_audit_event": ["event"] } 表示topic为actiontrail_audit_event的日志，该日志的event字段将把字符串展开为JSON格式。
Event source	否，String	Splunk event数据源。	无
Event source type	否，String	Splunk event数据源类型。	无
Event retry times	否，Integer	0表示无限重传。	默认值：0次
Event protocol	是	Splunk event发送协议。如果选择私有协议，后续参数可以忽略。	■ HTTP for HEC ■ HTTPS for HEC ■ Private protocol
HEC host	是，只有Event protocol选择HEC时有效，String。	HEC host。更多信息，请参见Set up and use HTTP Event Collector in Splunk Web。	无
HEC port	是，只有Event protocol选择HEC时有效，Integer。	HEC端口。	无
HEC token	是，只有Event protocol选择HEC时有效，String。	HEC token。更多信息，请参见HEC token。	无
HEC timeout	是，只有Event protocol选择HEC时有效，Integer。	HEC超时时间，单位：s。	默认：120s

当Splunk运行在阿里云ECS上时，可为ECS实例绑定RAM角色，并以此RAM角色来访问日志服务。具体操作如下：

> **注意** 请确保Splunk运行在被授予RAM角色的阿里云ECS实例上。

1. 授予实例RAM角色。

   此操作包括创建实例RAM角色、为RAM角色授予权限和为实例授予RAM角色。具体操作，请参见授予实例RAM角色。

   其中，RAM角色的权限策略请参见准备工作中AccessKey的授权策略。

2. 在Splunk Web UI页面中，单击**Alibaba Cloud Log Service Add-on for Splunk**。
3. 配置全局账号。

   选择**配置 > Account**，在**Account**页签中，配置ECS实例的RAM角色信息。即添加一个账号，用户名为ECS_RAM_ROLE，密码为步骤1中所创建的RAM角色名称。

4. 创建Data Input。

i. 单击**输入**，进入**输入**页面。

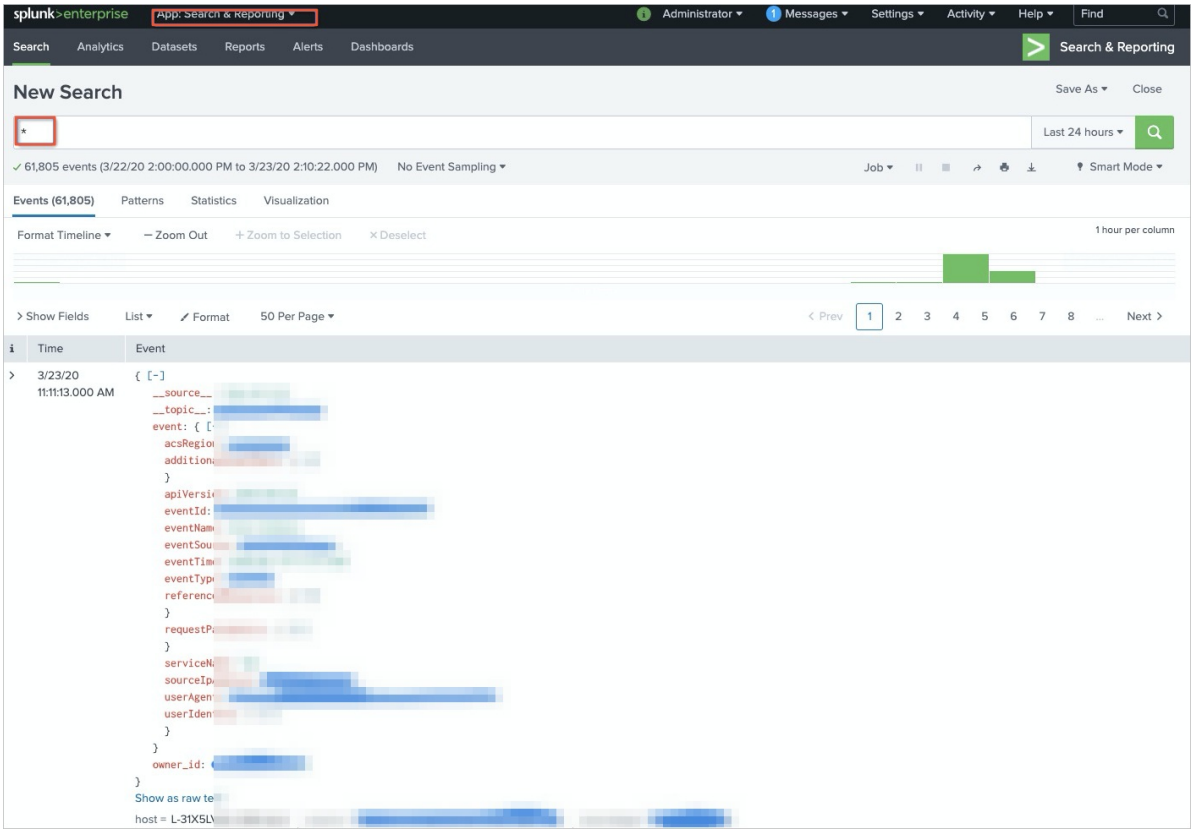
ii. 单击**Create New Input**，创建新的Data Input。

其中SLS AccessKey参数需配置为步骤3中所创建的全局账号，其他参数说明请参见Data input参数说明。

开始工作

- 查询数据

首先确保创建的Data Input是启动状态。然后您可以在Splunk web UI首页单击**Search & Reporting**，进入**App: Search & Reporting**页面，查询采集到的审计日志。



- 内部日志
  - 使用 `index="_internal" | search "SLS info"` 查询日志服务相关的信息。
  - 使用 `index="_internal" | search "error"` 查询运行时的错误信息。

性能与安全

- 性能规格

日志服务Splunk Add-on性能及数据传输的吞吐量受到如下因素的影响：

  - 服务入口：可以使用公网、经典网络、VPC网络或者全球加速服务入口。一般情况下，建议采用经典网络或VPC网络的服务入口。更多信息，请参见服务入口。
  - 带宽：包括日志服务与Splunk heavy forwarder间的带宽、Splunk heavy forwarder与indexer之间的带宽。
  - Splunk indexer处理能力：indexer的数据接收能力。
  - Shard数量：对于单个日志服务Logstore，配置的Shard数越多，数据传输能力越强。您需要根据原始日志的生成速率确认Shard数量。更多信息，请参见管理Shard。

- Splunk Data Input的配置数量：同一个Logstore，配置越多的Data Input（相同的消费组），吞吐量越大。

  > **说明** 消费者的并发受到日志服务Logstore Shard数量的影响。

- Splunk heavy forwarder占用的CPU core数和内存：一般情况下一个Splunk Data Input需要消耗1～2G内存，占用1个CPU core。

当上述条件满足的情况下，一个Splunk Data Input可以提供1M/s～2M/s的日志消费能力。实际使用时，您需要根据原始日志的生成速率确认Shard数量。

例如，一个Logstore有10M/s的日志生产能力，需要至少拆分出10个split，并且在Add-on中配置10个Data Input。如果是单机部署，需要具备10个空闲CPU core及12G的内存。

- 高可用性

  消费组将检测点（check-point）保存在服务器端，当一个消费者停止，另外一个消费者将自动接管并从断点继续消费。可以在不同机器上创建Splunk Data Input，这样在一台机器停止或者损坏的情况下，其他机器上的Splunk Data Input创建的消费者可以自动接管并从断点进行消费。理论上，为了备用，也可以在不同机器上启动大于Shard数量的Splunk Data Input。

- 使用HTTPS
  - 日志服务

    如果服务入口（endpoint）配置为https://前缀，例如https://cn-beijing.log.aliyuncs.com，程序与日志服务的连接将自动使用HTTPS加密。

    服务器证书*.aliyuncs.com是GlobalSign签发，默认大多数Linux、Windows的机器会自动信任此证书。如果某些特殊情况，机器不信任此证书，请参见Install a trusted root CA or self-signed certificate。

  - Splunk

    为了使用HTTPS的HEC能力，需要在Splunk HEC全局配置界面上打开SSL功能。更多信息，请参见Configure HTTP Event Collector on Splunk Enterprise。

- AccessKey存储保护

  日志服务的AccessKey和Splunk的HEC token等关键信息，都保存在Splunk保密存储中，不用担心泄漏风险。

常见问题

- 配置错误
  - 当创建或修改Data Input时，会有基本配置信息检查，配置参数说明请参见Data input参数说明。
  - 日志服务配置错误，例如创建消费组失败。
    - 命令：`index="_internal" | search "error"`
    - 异常日志：

      ```
      aliyun.log.consumer.exceptions.ClientWorkerException:
      error occour when create consumer group,
      errorCode: LogStoreNotExist,
      errorMessage: logstore xxxx does not exist
      ```

    - ConsumerGroupQuotaExceed错误

      在日志服务中，一个Logstore最多配置20个消费组。请在日志服务控制台查看消费组信息，如果空闲数量不足，建议删除不需要的消费组。如果超过20个消费组，系统会报ConsumerGroupQuotaExceed错误。

- 权限错误

- 无权限访问阿里云日志服务
  - 命令：`index="_internal" | search "error"`
  - 异常日志：

```
aliyun.log.consumer.exceptions.ClientWorkerException:
error occour when create consumer group,
errorCode: SignatureNotMatch,
errorMessage: signature J70VwxYH0+W/AciA4BdkuWxK6W8= not match
```

- ECS RAM鉴权失败
  - 命令：`index="_internal" | search "error"`
  - 异常日志：

```
ECS RAM Role detected in user config, but failed to get ECS RAM credentials. Please check
if ECS instance and RAM role 'ECS-Role' are configured appropriately.
```

    其中，*ECS-Role*为您创建的RAM角色，将根据实际情况显示。

  - 可能原因：
    - 确认Data Input中的SLS AccessKey参数是否已配置为具备RAM角色的全局账号。
    - 确认全局账号是否正确配置了RAM角色，账号必须为ECS_RAM_ROLE，密码为RAM角色名称。
    - 确认ECS实例是否已绑定RAM角色。
    - 确认RAM角色的可信实体类型是否为阿里云服务，受信服务是否为云服务器。
    - 确认绑定RAM角色的ECS实例是否为运行Splunk的机器。
- 无权限访问HEC
  - 命令：`index="_internal" | search "error"`
  - 异常日志：

```
ERROR HttpInputDataHandler - Failed processing http input, token name=n/a, channel=n/a, s
ource_IP=127.0.0.1, reply=4, events_processed=0, http_input_body_size=369
WARNING pid=48412 tid=ThreadPoolExecutor-0_1 file=base_modinput.py:log_warning:302 |
SLS info: Failed to write [{"event": "{\"__topic__\": \"topic_test0\", \"__source__\": \"
127.0.0.1\", \"__tag__:__client_ip__\": \"10.10.10.10\", \"__tag__:__receive_time__\": \"
1584945639\", \"content\": \"goroutine id [0, 1584945637]\", \"content2\": \"num[9], time
[2020-03-23 14:40:37|1584945637]\"}", "index": "main", "source": "sls log", "sourcetype":
"http of hec", "time": "1584945637"}] remote Splunk server (http://127.0.0.1:8088/service
s/collector) using hec.
Exception: 403 Client Error: Forbidden for url: http://127.0.0.1:8088/services/collector,
times: 3
```

  - 可能原因
    - HEC没有配置或启动。
    - Data Input中的HEC基本信息配置错误。例如：如果使用HTTPS，需要开启SSL配置。
    - 检查Event Collector token的indexer acknowledgment功能是否为关闭状态。

- 消费时延

- 如果您开启了服务日志功能，可以进行如下操作。
  - 通过消费组监控仪表盘查看消费组状态。更多信息，请参见服务日志仪表盘。

- 对消费者监控仪表盘中的图表设置告警。更多信息，请参见设置告警。

- 如果您未开启服务日志功能，可通过日志服务控制台查看消费组状态。更多信息，请参见通过控制台查看消费组状态。

建议参见性能规格，采用扩充Shard数或者创建具有相同消费组的Data Input的方式解决消费时延问题。

- 网络抖动
  - 命令：`index="_internal" | search "SLS info: Failed to write"`
  - 异常日志：

```
WARNING pid=58837 tid=ThreadPoolExecutor-0_0 file=base_modinput.py:log_warning:302 |
SLS info: Failed to write [{"event": "{\"__topic__\": \"topic_test0\", \"__source__\": \"12
7.0.0.1\", \"__tag__:__client_ip__\": \"10.10.10.10\", \"__tag__:__receive_time__\": \"1584
951417\", \"content2\": \"num[999], time[2020-03-23 16:16:57|1584951417]\", \"content\": \"
goroutine id [0, 1584951315]\"}", "index": "main", "source": "sls log", "sourcetype": "http
of hec", "time": "1584951417"}] remote Splunk server (http://127.0.0.1:8088/services/collec
tor) using hec.
Exception: ('Connection aborted.', ConnectionResetError(54, 'Connection reset by peer')), t
imes: 3
```

正常情况下，event会自动重传。但是如果问题持续存在，请联系网络管理员排查网络状况。

- 修改消费起始时间

> **说明** Data Input的消费起始时间参数，只有消费组首次创建时有效。非首次创建日志都是从上次的保存点开始消费。

i. 在Splunk Web UI的**输入**页面，禁用目标Data Input。

ii. 登录日志服务控制台，在所消费的目标Logstore下的**数据消费**栏中，删除对应的消费组。

iii. 在Splunk Web UI的**输入**页面，单击目标Data Input右侧的**操作 > 编辑**，修改SLS cursor start time参数，并重新启动Data Input。

5.5. Logstash消费

日志服务支持通过Logstash消费日志数据，您可以通过配置日志服务的Input插件对接Logstash获取日志服务中的日志数据并写入到其他系统中，例如Kafka、HDFS等。

功能特性

- 分布式协同消费：可配置多台服务器同时消费某一个Logstore。
- 高性能：基于Java ConsumerGroup实现，单核消费速度可达20 MB/s。
- 高可靠性：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，消费者增加或减少后会自动负载均衡。

操作步骤

1. 安装Logstash。
   i. 下载安装包。
   ii. 解压安装包到指定目录。
2. 安装input插件。
   i. 下载input插件。下载地址为logstash-input-sls。

ii. 安装input插件。

```
logstash-plugin install logstash-input-sls
```

> **说明** 插件安装失败原因及解决方案，请参见插件安装配置。

3. 启动Logstash。

```
logstash -f logstash.conf
```

配置参数如下表所示。

参数	类型	是否必须	说明
endpoint	string	是	日志服务项目所在的Endpoint。更多信息，请参见服务入口。
access_id	string	是	阿里云AccessKey ID，需要具备消费组相关权限。更多信息，请参见指定Logstore的消费权限。
access_key	string	是	阿里云AccessKey Secret，需要具备消费组相关权限。更多信息，请参见指定Logstore的消费权限。
project	string	是	日志服务Project名称。
logstore	string	是	日志服务Logstore名称。
consumer_group	string	是	消费组名称。
consumer_name	string	是	消费者名称，同一个消费组中的消费者名称不能重复。
position	string	是	消费开始位置。 ○ **begin**：从Logstore写入的第一条数据开始消费。 ○ **end**：从当前时间点开始消费。 ○ **yyyy-MM-dd HH:mm:ss**：从指定时间点开始消费。
checkpoint_second	number	否	每隔几秒Checkpoint一次，建议10秒~60秒，不能低于10秒，默认为30秒。
include_meta	boolean	否	日志数据是否包含Meta，Meta包括日志source、time、tag、topic，默认为true。
consumer_name_with_ip	boolean	否	消费者名称是否包含IP地址，默认为true，分布式协同消费下必须设置为true。

示例

配置Logstash消费某一个Logstore并将日志打印到标准输出，示例如下：

```
input {
  logservice{
  endpoint => "your project endpoint"
  access_id => "your access id"
  access_key => "your access key"
  project => "your project name"
  logstore => "your logstore name"
  consumer_group => "consumer group name"
  consumer_name => "consumer name"
  position => "end"
  checkpoint_second => 30
  include_meta => true
  consumer_name_with_ip => true
  }
}
output {
  stdout {}
}
```

5.6. Flume消费

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

背景信息

aliyun-log-flume是一个实现日志服务与Flume对接的插件，与Flume对接后，日志服务可以通过Flume与其它数据系统如HDFS、Kafka等对接。aliyun-log-flume提供Sink和Source实现日志服务与Flume的对接。

- Sink：Flume读取其他数据源的数据然后写入日志服务。
- Source：Flume消费日志服务的日志数据然后写入其他系统。

更多信息，请参见aliyun-log-flume。

操作步骤

1. 下载并安装Flume。

   更多信息，请参见Flume。
2. 下载aliyun-log-flume插件，并将插件存放于*cd/***/flume/lib*目录下。

   更多信息，请参见aliyun-log-flume-1.3.jar。
3. 在*cd/***/flume/conf*目录下，创建配置文件flumejob.conf。
   - Sink配置及示例请参见Sink。
   - Source配置及示例请参见Source。
4. 启动Flume。

Sink

通过Sink将其他数据源的数据通过Flume写入日志服务。目前支持两种解析格式：

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

Sink的配置如下：

参数	是否必须	说明
type	是	默认配置为com.aliyun.Loghub.flume.sink.LoghubSink。
endpoint	是	Project的服务入口，例如 `http://cn-qingdao.log.aliyuncs.com` 。请根据实际情况替换服务入口。更多信息，请参见服务入口。
project	是	Project名称。
logstore	是	Logstore名称。
accessKeyId	是	阿里云AccessKey ID，用于标识用户。为保证账号安全，建议您使用RAM用户的AccessKey。如何获取AccessKey，请参见访问密钥。
accessKey	是	阿里云AccessKey Secret，用于验证用户的密钥。为保证账号安全，建议您使用RAM用户的AccessKey。如何获取AccessKey，请参见访问密钥。
batchSize	否	每批次写入日志服务的数据条数。默认为1000条。
maxBufferSize	否	缓存队列的大小。默认为1000条。
serializer	否	Event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。 • **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模式，设置为该模式时需要填写完整列名称。
columns	否	当serializer为**DELIMITED**时，必须指定该字段列表，用半角逗号（,）分隔，顺序与实际数据中的字段顺序一致。
separatorChar	否	当serializer为**DELIMITED**时，用于指定数据的分隔符，必须为单个字符。默认为英文逗号（,）。
quoteChar	否	当serializer为**DELIMITED**时，用于指定引用符。默认为半角双引号（"）。
escapeChar	否	当serializer为**DELIMITED**时，用于指定转义字符。默认为半角双引号（"）。
useRecordTime	否	用于设置是否使用数据中的timestamp字段作为日志时间。默认为false表示使用当前时间。

Sink配置示例请参见GitHub。

Source

通过Source将日志服务的日志数据通过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。

Source配置如下：

参数	是否必须	说明
type	是	默认配置为com.aliyun.Loghub.flume.source.LoghubSource。
endpoint	是	Project的服务入口，例如 `http://cn-qingdao.log.aliyuncs.com` 。请根据实际情况替换服务入口。更多信息，请参见服务入口。
project	是	Project名称。
logstore	是	Logstore名称。
accessKeyId	是	阿里云AccessKey ID，用于标识用户。为保证账号安全，建议您使用RAM用户的AccessKey。如何获取AccessKey，请参见访问密钥。
accessKey	是	阿里云AccessKey Secret，用于验证用户的密钥。为保证账号安全，建议您使用RAM用户的AccessKey。如何获取AccessKey，请参见访问密钥。
heartbeatIntervalMs	否	客户端和日志服务的心跳间隔，默认为30000毫秒。
fetchIntervalMs	否	数据拉取间隔，默认为100毫秒。
fetchInOrder	否	是否按顺序消费。默认为false。
batchSize	否	每批次读取的数据条数，默认为100条。
consumerGroup	否	读取的消费组名称。
initialPosition	否	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	否	当initialPosition为**timestamp**时，必须指定时间戳，为Unix时间戳格式。
deserializer	是	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。 • **JSON**：设置解析格式为JSON模式。 • 自定义**deserializer**：设置解析格式为自定义的反序列化模式，设置为该模式时需要填写完整列名称。
columns	否	当deserializer为**DELIMITED**时，必须指定字段列表，用半角逗号（,）分隔，顺序与实际数据中的字段顺序一致。
separatorChar	否	当deserializer为**DELIMITED**时，用于指定数据的分隔符，必须为单个字符。默认为英文逗号（,）。

参数	是否必须	说明
quoteChar	否	当deserializer为DELIMITED时，用于指定引用符。默认为半角双引号（"）。
escapeChar	否	当deserializer为DELIMITED时，用于指定转义字符。默认为半角双引号（"）。
appendTimestamp	否	当deserializer为DELIMITED时，用于设置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	否	当deserializer为JSON时，用于设置是否将日志Source作为一个字段，字段名称为__source__。默认为false。
tagAsField	否	当deserializer为JSON时，用于设置是否将日志Tag作为字段，字段名称为__tag__：{tag名称}。默认为false。
timeAsField	否	当deserializer为JSON时，用于设置是否将日志时间作为一个字段，字段名称为__time__。默认为false。
useRecordTime	否	用于设置是否使用日志的时间，如果为false则使用当前时间。默认为false。

Source配置示例请参见GitHub。

6.最佳实践

6.1. 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。

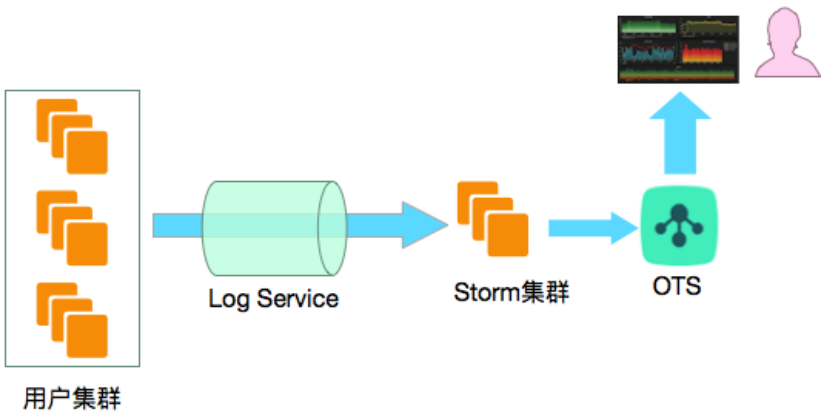
日志服务帮用户完成了日志收集，统一了日志格式，提供API用于下游消费。下游系统可以接入多重系统重复消费，比如导入Spark、Storm做实时计算，也可以导入elastic search做搜索，真正做到了一次收集，多次消费。在众多数据消费场景中，监控是最常见的一种场景。本文介绍阿里云基于日志服务的监控系统。

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。

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。

6.2. 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。本文介绍一种基于日志服务按量计费方案，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

计量日志记录了您所涉及计费的项目，后端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:203.0.113.10 ClientIP:198.
51.100.10 Latency:1968 InFlow:1409 NetFlow:474 OutFlow:0 UserId:44 AliUid:1264425845****** Proj
ectName: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提供账单。
- 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，后端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。

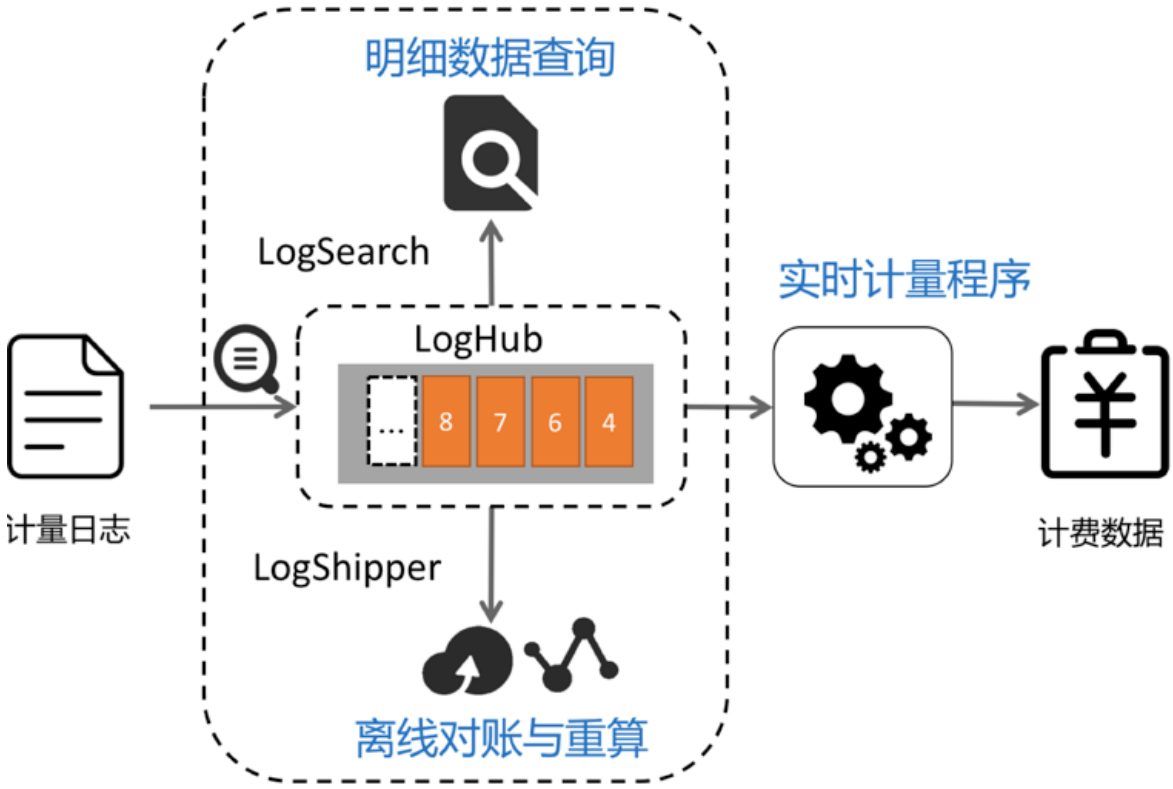
- 容错处理：计费程序可能有Bug，如何确保计量数据与计费程序独立。

本文介绍一种基于日志服务按量计费方案，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，供大家参考。

系统架构

以日志服务LogHub功能为例：

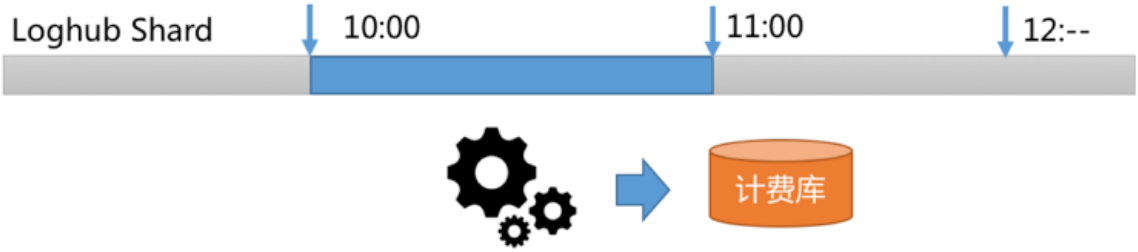
1. 使用LogHub进行计量日志实时采集与计量程序对接：LogHub支持的50+种接入手段。
2. 计量程序每隔固定时间消费LogHub中步长数据，在内存中计算生成计费数据结果。
3. （附加）对明细数据查询需求，可以将计量日志配置索引查询。
4. （附加）将计量日志推送至OSS、MaxCompute进行离线存储，进行T+1等对账与统计。



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：

- 假设每天有10亿条计量日志，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1.6GB。
- 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

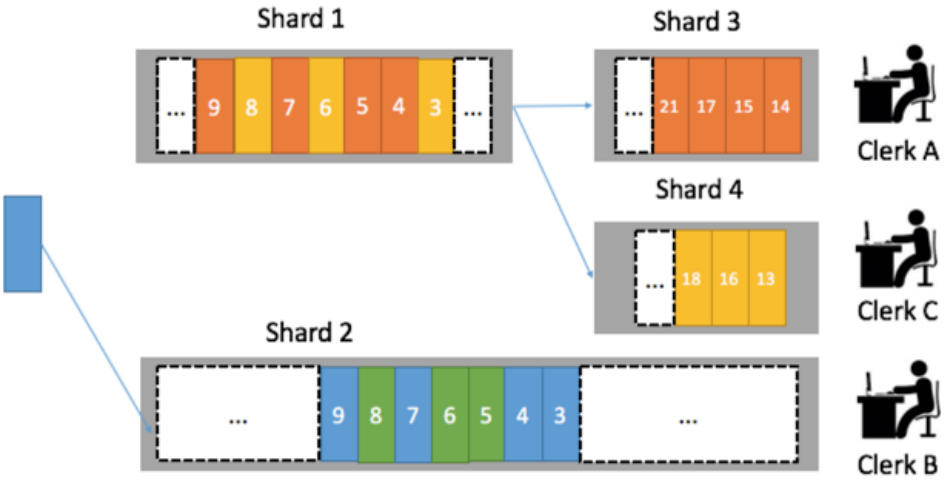
在一些计费场景下（例如运营商、IoT等），计量日志量会很大（例如十万亿，数据量为每天2PB），折算压缩数据后一小时有16TB，以万兆网络读取需要1600秒，已不能满足快速出账单需求。

- 控制产生的计费数据量

  对于产生计量日志程序进行改造（例如Nginx），先在内存中做聚合，每隔1分钟转储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，一个小时数据量为1000*200*60=12GB（压缩后为240MB）。

- 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，在生命周期内可以根据任意时间段进行计算。

- 计量日志散落在很多服务器中怎么办？

  i. 使用Logtail Agent实时采集。

  ii. 使用自定义用户标识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中的数据创建索引，支持实时查询与统计分析。

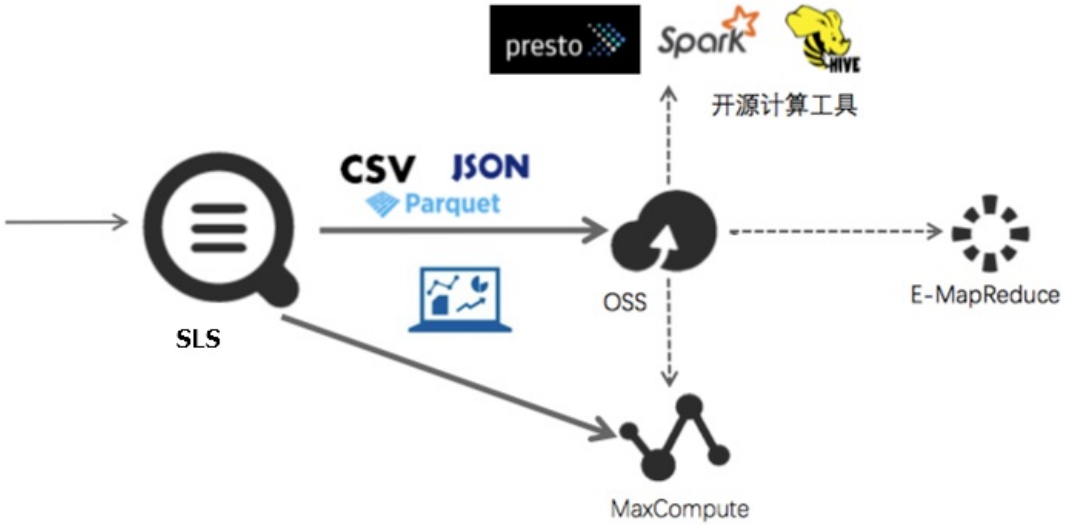
  ```
  Inflow>300000 and Method=Post* and Status in [200 300]
  ```

  您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
```

- 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，其支持自定义分区、自定义存储格式等配置。将日志存储在OSS、MaxCompute上，利用E-MapReduce、MaxCompute、Hadoop、Hive、Presto、Spark等进行计算。



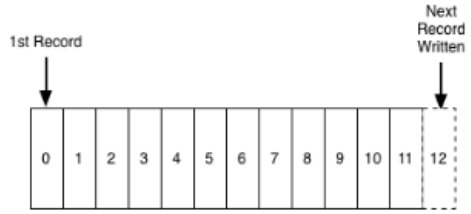
6.3. 消费-通过消费组实现高可靠消费

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Logstore消费组功能，如何与Spark Streaming、Storm Spout等配合，完成日志数据的处理过程。

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`。



日志有两个主要特性：

- 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，例如有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。

在该文档的示例中，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：

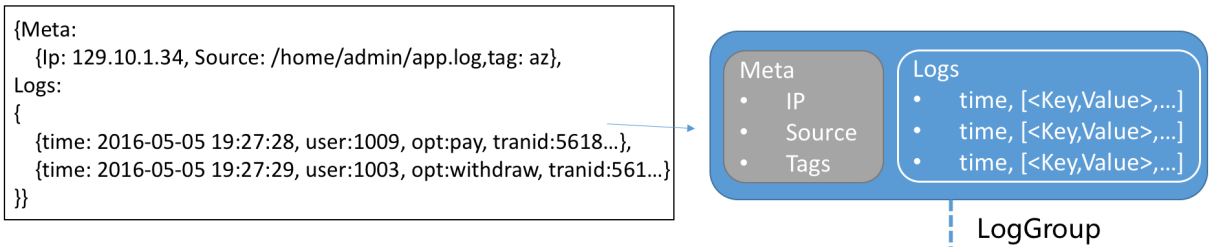
```
2016-06-28 08:00:00 张三 存款 1000元美元
2016-06-27 09:00:00 李四 取款 20000元美元
```

Logstore数据模型

为了能抽象问题，这里以日志服务Logstore作为演示模型。其包含以下内容：

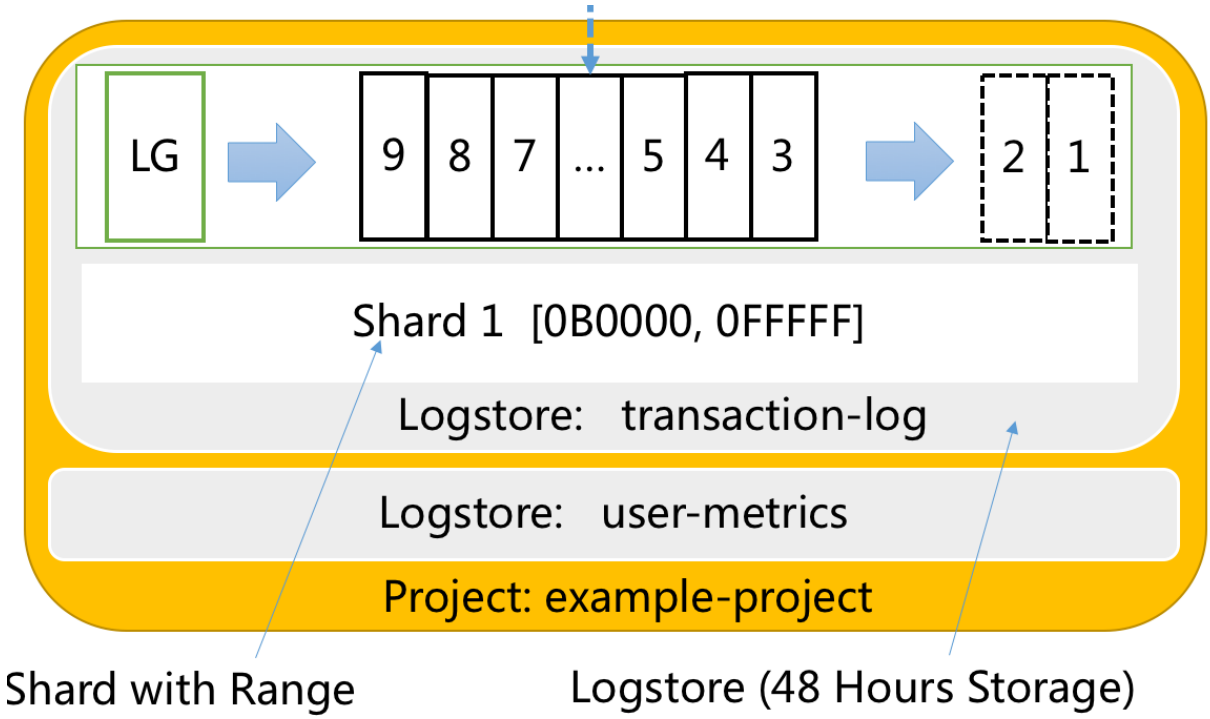
- Log：由时间及一组Key-Value对组成。
- LogGroup：一组日志的集合，包含相同Meta（IP、Source）等。

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读能力或10 MB/S写能力。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）。

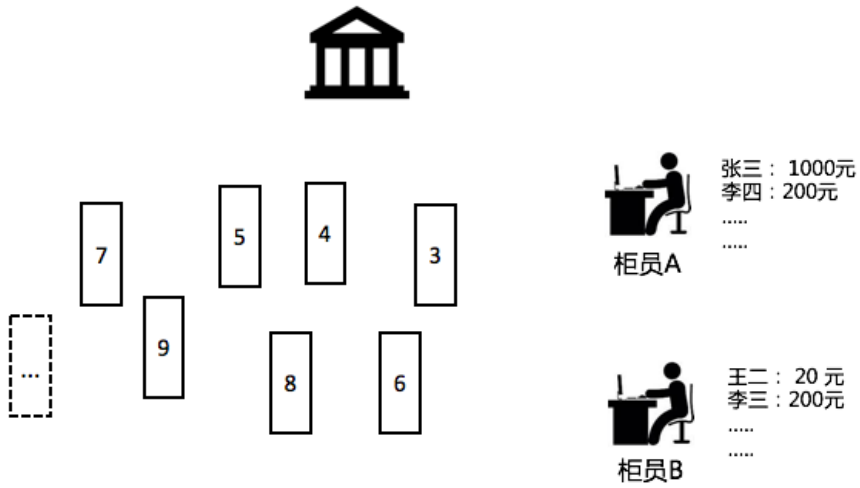
该场景中，各角色及其主要操作包括：

- 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。

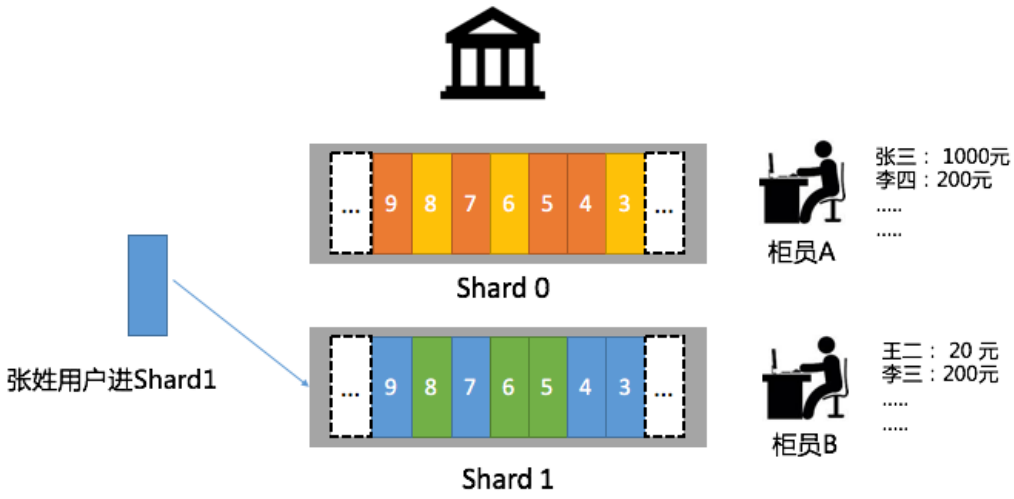
从这个例子可以看到，存取款是一个严格有序的操作，需要同一个柜员（处理器）来处理同一个用户的操作，这样才能保持状态一致性。



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。用户请求先进先出，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？

假设有10个柜员，我们可以创建10个Shard。要保证10个柜员对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落在一个特定Shard中（区间包含Z），处理端面对的一直是柜员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。

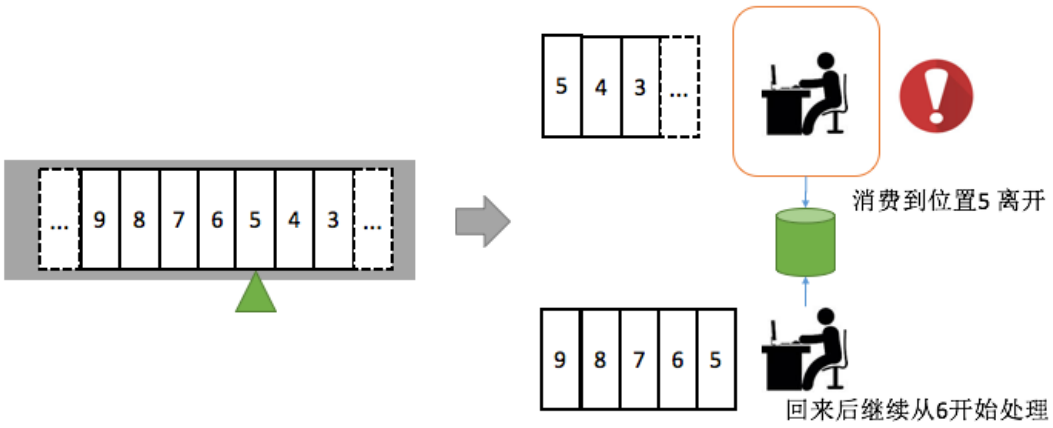


问题2：不丢失（At-Least Once）

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，后果是无法容忍的。

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。例如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A回来后，发现张三请求没有记录下来，A会把张三请求再次处理一遍，这就会造成重复。

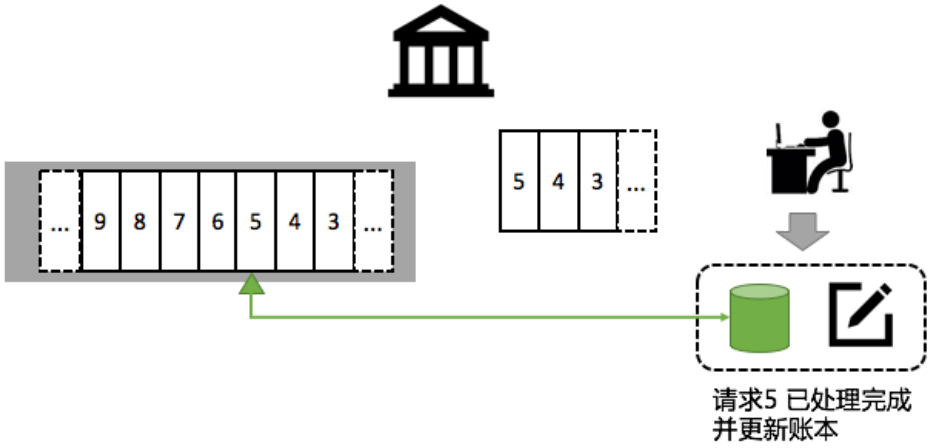
问题3：不重复（Exactly Once）

重复是否一定会带来问题？答案是不一定。

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

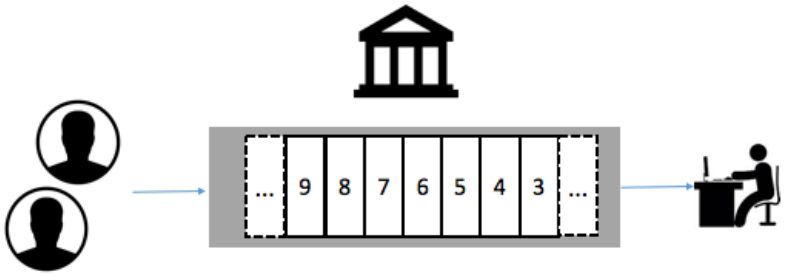
以上三个概念解释完成后，原理并不复杂。但在现实世界中，用户数、处理量规模的变化与不确定性会使得以上三个问题变得更复杂。

- 遇到发工资的日期，用户数会大涨。
- 柜员（Clerk）毕竟不是机器人，也需要休假，需要吃午饭。
- 银行经理为了整体服务体验，需要增加柜员。那以什么作为判断标准增加柜员呢？
- 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。

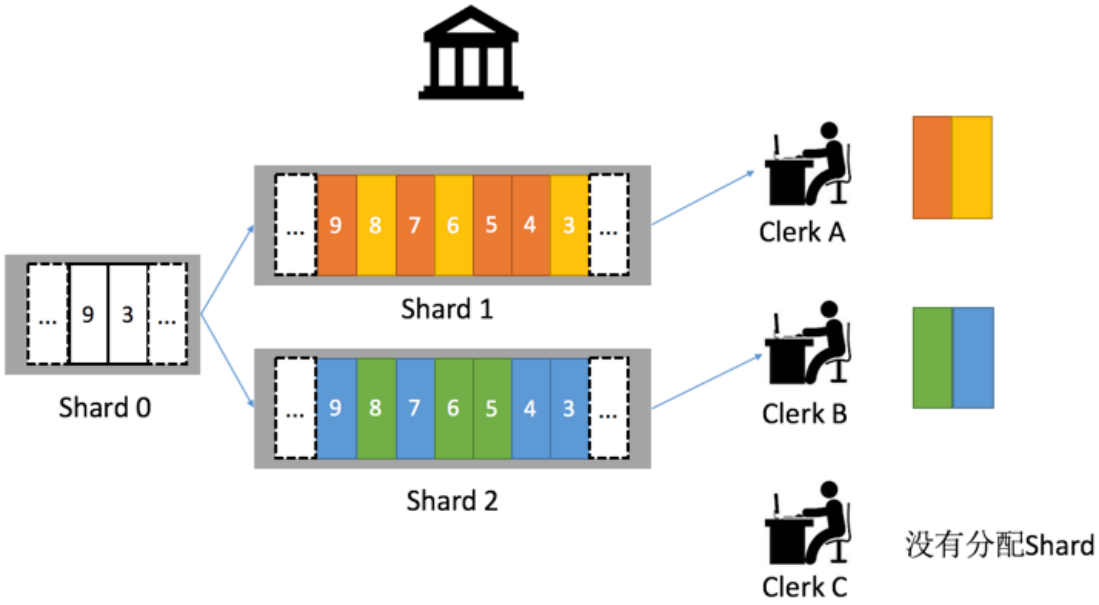
  只有一个Shard0，用户请求全部排在Shard0下，柜员A也正好可以处理。



- 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Shard2中排队等待处理。这两个Shard区间明显是不均匀的，因为用户的姓氏分布就是不均匀的，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。

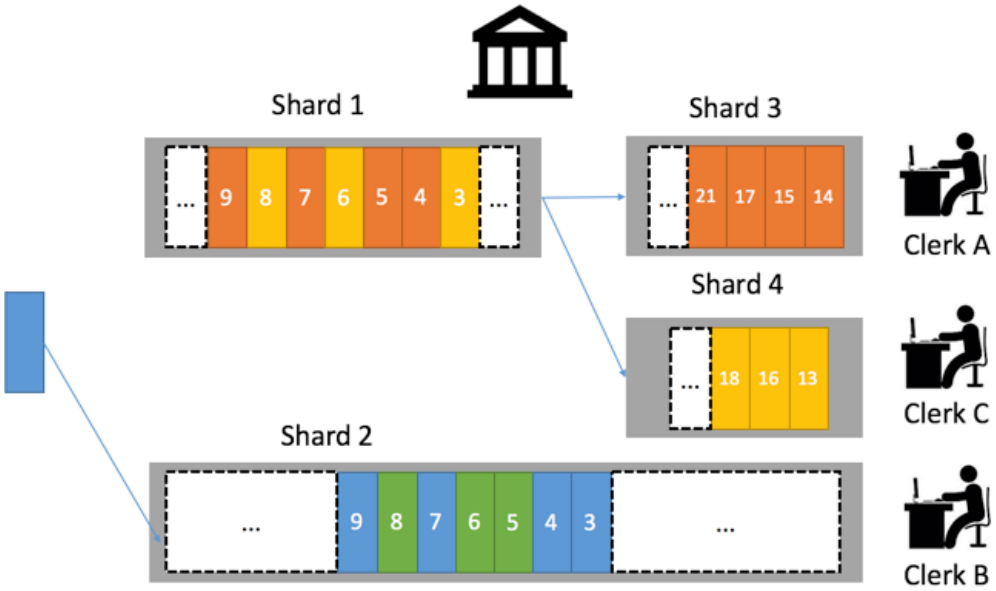


  柜员A处理2个Shard非常吃力，于是经理派出柜员B、C出场。因为只有2个Shard，B开始接管A负责一个Shard，C处于闲置状态。

- 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Shard1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：

- 对Shard进行弹性伸缩。
- 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。
- 过程中不重复（需要消费者配合）。
- 观察到消费进度，以便合理调配计算资源。
- 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）。

您可以通过Logstore消费组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。更多信息，请参见通过消费组消费日志数据。

另外，Storm Spout、Spark Streaming已经通过消费组实现了对应的接口，欢迎使用。更多信息，请参见Storm Sport消费、Spark Streaming消费。

7.常见问题

7.1. 为什么实时消费的速率未达到Shard的读写阈值上限？

实时消费的速率未达到Shard的读写阈值上限的可能原因如下：

- 您所设置的加工规则逻辑较为复杂。
- 通过数据加工进行跨地域传输数据时，依赖公网网络环境，可能因为公网网络问题影响消费速率。