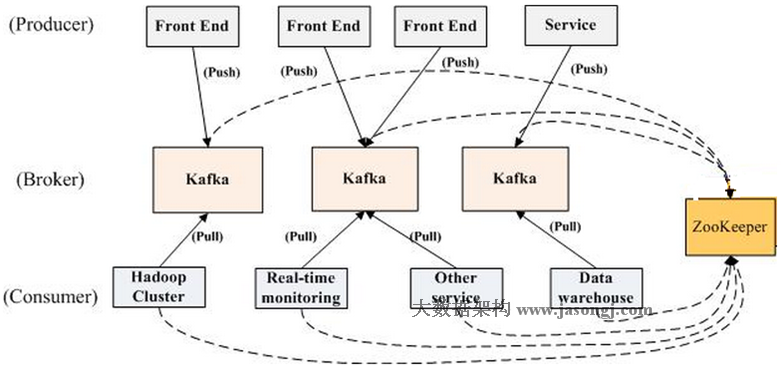
# kafka简单介绍：

kafka is a distributed,partitioned,replicated commit logservice.kafka提供了类似于JMS的特性,但它并不是JMS规范的实现,在kafka中和JMS实现不同的是,即使消息被消费,消息仍然不会被立即删除.,日志文件将会根据broker中的配置,保留一段时间后被删除,无论消息是否被消费,kafka通过这种简单的手段,来释放磁盘空间,以及减少消息消费之后对文件内容改动的磁盘IO开支.

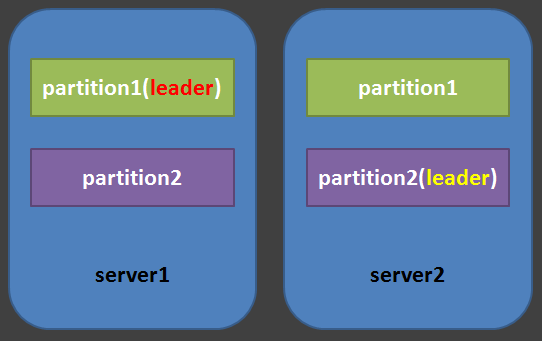
## 名词

* **Broker**  
  Kafka集群包含一个或多个服务器，这种服务器被称为broker
* **Topic**  
  每条发布到Kafka集群的消息都有一个类别，这个类别被称为topic。（物理上不同topic的消息分开存储，逻辑上一个topic的消息虽然保存于一个或多个broker上但用户只需指定消息的topic即可生产或消费数据而不必关心数据存于何处）
* **Partition**  
  parition是物理上的概念，每个topic包含一个或多个partition，创建topic时可指定parition数量。每个partition对应于一个文件夹，该文件夹下存储该partition的数据和索引文件
* **Producer**  
  负责发布消息到Kafka broker
* **Consumer**  
  消费消息。每个consumer属于一个特定的consumer group（可为每个consumer指定group name，若不指定group name则属于默认的group）。使用consumer high level API时，同一topic的一条消息只能被同一个consumer group内的一个consumer消费，但多个consumer group可同时消费这一消息。

## Kafka架构

****

总体架构图



partition和replica

如上图所示，一个典型的kafka集群中包含若干producer，若干broker（Kafka支持水平扩展，一般broker数量越多，集群吞吐率越高），若干consumer group，以及一个[Zookeeper](http://zookeeper.apache.org/)集群。Kafka通过Zookeeper管理集群配置，选举leader。producer使用push模式将消息发布到broker，consumer使用pull模式从broker订阅并消费消息。

向一个分区中读写消息都是通过leader来操作的,follower只是不断的和leader同步,并在leader宕机时,顶替leader工作

## Push vs. Pull

作为一个messaging system，Kafka遵循了传统的方式，选择由producer向broker push消息并由consumer从broker pull消息。

push模式很难适应消费速率不同的消费者，因为消息发送速率是由broker决定的。push模式的目标是尽可能以最快速度传递消息，但是这样很容易造成consumer来不及处理消息，典型的表现就是拒绝服务以及网络拥塞。而pull模式则可以根据consumer的消费能力以适当的速率消费消息。

## partition的设计目的

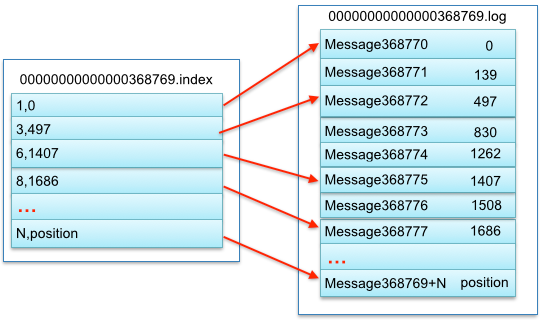
设计partition有多个原因,但最根本的原因是kafka基于文件存储,通过分区,可以将日志内容分散到多个server上,以此来避免文件尺寸达到单机磁盘的上限,一个topic被分成多个partition后可以提高消息保存和消费的效率,越多的partition代表可以提供更高的并发量

## 日志存储

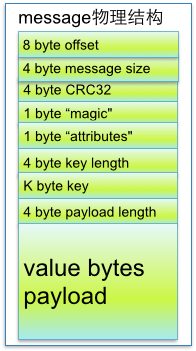
如果一个topic的名称为my\_topic ,它有2个partitions,那么日志将会保存在my\_topic\_0和my\_topic\_1两个目录中,日志文件中保存了一系列messages(日志条目),每个partition在物理存储层面,由多个segment(片段) file组成.segment file的命名为"最小offset-1 "例如"000000000000.log最小offset为1";其中最小offset表示此segmentfile中起始消息的offset.

segment file由两部分组成,分别为index file 和 data file,这两个文件一一对应,成对出现,后缀为".index"和".log",分别表示segment索引文件和数据文件.

index file的存储结构为:



log file中每一条message的存储结构:



## 日志文件删除策略

对于传统的message queue而言，一般会删除已经被消费的消息，而Kafka集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此Kafka提供两种策略去删除旧数据。一是基于时间，二是基于partition文件大小。可以通过配置$KAFKA\_HOME/config/server.properties实现

### 基于时间

log.retention.hours=168(日志文件保留一周)

log.retention.minutes=60(日志文件保留60分钟)

log.retention.ms=60(日志文件保留60秒)

### 基于文件大小

log.retention.bytes=1024576(当总日志文件大于1M时删除)

## kafka性能测试结果

测试环境:

kafka : 0.9.0.1

java : 1.7.0\_45

os: window 7

cpu: 4核

内存: 8G

硬盘: 7200转

生产者测试结果(横坐标:单条消息大小KB,纵坐标(左):每秒条数,纵坐标(右):每秒MB)

消费者测试结果(横坐标:单条消息大小KB,纵坐标(左):每秒条数,纵坐标(右):每秒MB)

# kafka常用配置

## broker配置

* broker.id: broker集群中用于标识一个broker
* log.dirs: 用于指定存放消息文件的路径
* port: broker用于接收producer消息的端口,默认为9092
* zookeeper.connect: zookeeper地址

## producer配置

* bootstrap.servers: 用于指定需要连接的broker地址列表,不需要将全部broker地址配置上也可以运行,但建议将可用的broker全部配上

## consumer配置

* group.id:指定consumer group的唯一标识
* consumer.id:唯一标识consumer.默认值为null,不指定会自动生成,consumer通过consumer.id来与某个partition(A)绑定,在一个分组内某个partition上的消息只会被分配到被绑定的consumer(A)中,如果该consumer(A)宕机,与其绑定的partition的消息会被分配到其他consumer(B),但当宕机的consumer(A)重启后该partition会重新自动绑定到consumer(A)上,但如果使用自动生成的consumer.id,宕机的consumer不会重新与之前的partition绑定,因为自动生成的consumer.id不一样
* zookeeper.connect:指定zookeeper地址,格式为<hostname:port>,<hostname.port>...
* auto.commit.enable:该属性默认为true,当为true时已经被consumer获取到的消息offset会被阶段性提交到zookeeper中,在consumer失效时新consumer将以提交的offset作为起始位置
* auto.commit.interval.ms:指定被消费的offset提交到zookeeper的频率(毫秒).默认值为60\*1000
* auto.offset.reset: auto.offset.reset定义了consumer在zookeeper中发现没有初始的offset时或者发现offset非法时定义consumer的行为,常见配置有:

smallest,自动把offset设为最小的offset;

largest:自动offset设为最大的offset.

默认值为largest,通过设置该属性来指定新启动的consumer是从头开始消费,还是 只消费启动之后被发送到broker的消息.

# kafka部署

## 单机版部署：

### 启动zookeeper

1.进入Zookeeper设置目录，比如

C:\zookeeper-3.4.7\conf

2. 将“zoo\_sample.cfg”重命名为“zoo.cfg”。

3. 在任意文本编辑器中打开zoo.cfg

4. 找到并编辑

dataDir=/tmp/zookeeper to D:\kafka\zookeeper-3.4.7\data

5.在zoo.cfg文件中修改默认的Zookeeper端口（默认端口2181）

6.启动zookeeper

进入到bin目录下运行zkServer.cmd，zookeeper就成功启动了。

### 启动单机版kafka

1.先下载kafka解压

2.进入config目录下打开“server.properties”文件，

 3.找到并编辑“log.dirs=/tmp/kafka-logs”为

“log.dir= D:\kafka\kafka\_logs”

4. 如果Zookeeper在某些其他的机器或集群上运行，

可以将“zookeeper.connect=localhost:2181”修改为自定义IP与端口。

使用同一个机器则没必要做修改。文件中的Kafka端口和broker.id也是可以配置的。其他设置不变。

5. Kafka会按照默认，在9092端口上运行，并连接zookeeper的默认端口：2181。

6.启动kafka服务器

重要：请确保在启动Kafka服务器前，Zookeeper实例已经准备好并开始运行。

(1).进入Kafka安装目录D:\ kafka\_2.10-0.10.0.0\

(2).按下Shift+右键，选择“打开命令窗口”选项，打开命令行**。**

(3).输入

**.\bin\windows\kafka-server-start.bat  .\config\server.properties**

并回车。

## 搭建多broker

### 准备配置文件

刚才只是启动了单个broker，现在启动由3个broker组成的集群，这些broker节点也都是在本机上的：

首先为每个节点编写配置文件：

 复制config目录下的server.properties文件两份，分别命名为

**server-1.properties**和**server-2.properties**

在拷贝出的新文件中添加以下参数：

config/server-1.properties:

**broker.id=1**

**port=9093**

**log.dirs=D:\\kafka\\kafka\_2.10-0.10.0.0\\kafka-logs-1**

config/server-2.properties:

**broker.id=2**

**port=9094**

**log.dirs=D:\\kafka\\kafka\_2.10-0.10.0.0\\kafka-logs-2**

broker.id在集群中唯一的标注一个节点，因为在同一个机器上，所以必须制定不同的端口和日志文件。

### 启动服务

刚才已经启动了一个zookeeper和一个broker节点，现在启动另外两个节点：

在kafka根目录下打开命令窗口并执行一下命令：

**.\bin\windows\kafka-server-start.bat  .\config\server-1.properties**

**.\bin\windows\kafka-server-start.bat  .\config\server-2.properties**

到这里,多节点kafka就已经搭建完毕

## 搭建多节点多broker集群

### 修改配置文件

准备三台zookeeper，配置文zoo.cfg中端口分别是2181,2182,2183；每个加入如下信息：

server.1=127.0.0.1:2881:3881

server.2=127.0.0.1:2882:3882

server.3=127.0.0.1:2883:3883

并在每个datedir目录下创建myid文件，分别写入1,2,3.

把3.2中kafka的server.properties, server-1.properties, server-2.propertie文件中的zookeeper链接信息都改为：

zookeeper.connect=localhost:2181,localhost:2182,localhost:2183

### 启动服务

先启动三台zookeeper，再分别启动三台kafaka

多zookeeper,多broker搭建完毕

# kafka常用命令

## topic

### 创建topic

|  |
| --- |
| **kafka-topics.bat** **--create** **--zookeeper** localhost:2181 **--replication-factor** 3 **--partitions** 1 **--topic** mytopic |

### 查看topic信息

|  |
| --- |
| **kafka-topics.bat --describe --zookeeper** 127.0.0.1:2181 **--topic** mytopic |

### 查看topic列表

|  |
| --- |
| **kafka-topics.bat** **--list --zookeeper** 127.0.0.1:2181 |

## producer

### 发送消息

|  |
| --- |
| **kafka-console-producer.bat** **--broker-list** 127.0.0.1:9093 **--topic** my\_topic1 |

## consumer

### 接收消息

|  |
| --- |
| **kafka-console-consumer.bat** **--zookeeper** 127.0.0.1:2181 **--from-beginning** **--topic** my\_topic1 |

# kafka与spring集成

## maven导入jar包

|  |
| --- |
| <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka\_2.10</artifactId>  <version>0.8.2.1</version>  </dependency>  <dependency>  <groupId>org.springframework.integration</groupId>  <artifactId>spring-integration-kafka</artifactId>  <version>1.2.0.RELEASE</version>  </dependency> |

## 配置消息发送者

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:context=*"http://www.springframework.org/schema/context"*  xmlns:aop=*"http://www.springframework.org/schema/aop"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:int-kafka=*"http://www.springframework.org/schema/integration/kafka"*  xmlns:int=*"http://www.springframework.org/schema/integration"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-4.0.xsd*  *http://www.springframework.org/schema/aop*  *http://www.springframework.org/schema/aop/spring-aop-4.0.xsd*  *http://www.springframework.org/schema/integration*  *http://www.springframework.org/schema/integration/spring-integration-4.1.xsd*  *http://www.springframework.org/schema/integration/kafka*  *http://www.springframework.org/schema/integration/kafka/spring-integration-kafka.xsd"*>  <!--kafka config-->  <int:channel id=*"inputToKafka"*/>  <int-kafka:outbound-channel-adapter kafka-producer-context-ref=*"kafkaProducerContext"*  auto-startup=*"true"*  channel=*"inputToKafka"*  order=*"1"*>  </int-kafka:outbound-channel-adapter>  <bean id=*"producerProperties"* class=*"org.springframework.beans.factory.config.PropertiesFactoryBean"*>  <property name=*"properties"*>  <props>  <prop key=*"topic.metadata.refresh.interval.ms"*>3600000</prop>  <prop key=*"message.send.max.retries"*>5</prop>  <prop key=*"send.buffer.bytes"*>5242880</prop>  </props>  </property>  </bean>  <bean id=*"stringSerializer"* class=*"org.apache.kafka.common.serialization.StringSerializer"*/>  <int-kafka:producer-context id=*"kafkaProducerContext"* producer-properties=*"producerProperties"*>  <int-kafka:producer-configurations>  <int-kafka:producer-configuration broker-list=*"localhost:9092"*  key-serializer=*"stringSerializer"*  value-class-type=*"java.lang.String"*  value-serializer=*"stringSerializer"*  topic=*"test"*/>  </int-kafka:producer-configurations>  </int-kafka:producer-context>  </beans> |

## 发送者测试类

|  |
| --- |
| public class KafkaServiceImpl implements KafkaService {  @Autowired  @Qualifier("inputToKafka")  MessageChannel channel;  public void sendUserInfo(String topic, Object obj) {  Message msg = MessageBuilder.withPayload(obj)  .setHeader(KafkaHeaders.TOPIC, "topic").build();  channel.send(msg);  }  } |

## 配置消费者

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*  xmlns:int=*"http://www.springframework.org/schema/integration"*  xmlns:int-kafka=*"http://www.springframework.org/schema/integration/kafka"*  xsi:schemaLocation=*"*  *http://www.springframework.org/schema/integration/kafka http://www.springframework.org/schema/integration/kafka/spring-integration-kafka.xsd*  *http://www.springframework.org/schema/integration http://www.springframework.org/schema/integration/spring-integration.xsd*  *http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>  <int:channel id=*"inputFromKafka"*/>  <int-kafka:zookeeper-connect id=*"zookeeperConnect"* zk-connect=*"localhost:2181"*  zk-connection-timeout=*"6000"*  zk-session-timeout=*"6000"*  zk-sync-time=*"2000"*/>  <int-kafka:inbound-channel-adapter  id=*"kafkaInboundChannelAdapter"*  kafka-consumer-context-ref=*"consumerContext"*  auto-startup=*"true"*  channel=*"inputFromKafka"*>  <int:poller fixed-delay=*"1"* time-unit=*"MILLISECONDS"*/>  </int-kafka:inbound-channel-adapter>  <bean id=*"kafkaDecoder"* class=*"org.springframework.integration.kafka.serializer.common.StringDecoder"*>  </bean>  <bean id=*"consumerProperties"* class=*"org.springframework.beans.factory.config.PropertiesFactoryBean"*>  <property name=*"properties"*>  <props>  <prop key=*"auto.offset.reset"*>smallest</prop>  <prop key=*"socket.receive.buffer.bytes"*>10485760</prop> <!-- 10M -->  <prop key=*"fetch.message.max.bytes"*>5242880</prop>  <prop key=*"auto.commit.interval.ms"*>1000</prop>  </props>  </property>  </bean>  <bean id=*"kafkaService"* class=*"com.taikang.common.service.KafkaService"*>  </bean>  <int:outbound-channel-adapter channel=*"inputFromKafka"*  ref=*"kafkaService"* method=*"processMessage"* />  <int-kafka:consumer-context id=*"consumerContext"*  consumer-timeout=*"1000"*  zookeeper-connect=*"zookeeperConnect"* consumer-properties=*"consumerProperties"*>  <int-kafka:consumer-configurations>  <int-kafka:consumer-configuration group-id=*"default1"*  value-decoder=*"kafkaDecoder"*  key-decoder=*"kafkaDecoder"*  max-messages=*"5000"*>  <int-kafka:topic id=*"test"* streams=*"4"*/>  <int-kafka:topic id=*"my-replicated-topic"* streams=*"4"*/>  </int-kafka:consumer-configuration>  </int-kafka:consumer-configurations>  </int-kafka:consumer-context>  </beans> |

## 消费者实现类

|  |
| --- |
| **package** com.taikang.common.service;  **import** java.util.Map;  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  **public** **class** KafkaService {  **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(KafkaService.**class**);  **public** **void** processMessage(Map<String, Map<Integer, String>> msgs) {  **for** (Map.Entry < String,Map<Integer, String>>entry:  msgs.entrySet()){  ***logger***.debug("Suchit Topic:" + entry.getKey());  **for** (String msg : entry.getValue().values()) {  ***logger***.info("Suchit Consumed Message: " + msg);  }  }  }  } |

# kafka丢数据的一些场景

## producer到broker

调用producer API发送数据只会将数据保存到一个队列中,方法就返回了,有一个专门的线程负责发送队列中的消息,如果队列中的数据为发出去之前producer端crash,那么队列中的数据就丢失了.

kafka中只能保证至少发送一次,不能保证只发送一次

## broker端

broker端也使用了缓存技术,当broker接收到消息后,为了提高性能,不会立即将消息持久化到磁盘,而是缓存起来,然后批量flush到磁盘.如果在flush之前broker宕机就会丢数据,在0.8以后kafka提供了replica技术,可以保证服务端不丢数据.

## consumer端

在使用high-level API时,如果auto.commit.enable=true,当consumer fetch了一些数据,但还没有完全处理掉的时候,刚好到commit interval触发了提交offset操作,接着consumer crash掉了,这时已经fetch的数据还没有处理完成但已经被commit掉,重启后未处理的数据不会重新被fetch到,数据丢失.

# kafka源码分析

## kafka producer主要组件

* Metrics
* MetricsReporter
* Partitioner
* MetaData
* Cluster
* RecordAccumulator
* ChannelBuilder
* KafkaChannel
* Sender
* KafkaThread
* Selector
* NetworkClient

## KafkaClient

### ready()

建立一个SocketChannel,并注册到java.nio.Selector中

使用TransportLayer封装SocketChannel和对应的SelectionKey

使用KafkaChannel封装TransportLayer,并将kafkaChannel对象放入一个map中

### send()

将需要发送的消息存入对应的KafkaChannel对象中

### poll()

检测socketChannel连接情况,并发送消息

## kafka producer metadata更新过程

kafka producer发送消息和同步消息,由sender线程驱动,

### sender循环

sender线程中一直不停的循环,循环过程如下:

|  |
| --- |
| **public** **void** run(**long** now) {  //获取cluster对象  Cluster cluster = metadata.fetch();  //检查有没有需要发送的消息  RecordAccumulator.ReadyCheckResult result = **this**.accumulator.ready(cluster, now);  // 移除发往尚未建立连接节点的消息  Iterator<Node> iter = result.readyNodes.iterator();  **long** notReadyTimeout = Long.***MAX\_VALUE***;  **while** (iter.hasNext()) {  Node node = iter.next();  **if** (!**this**.client.ready(node, now)) {  iter.remove();  notReadyTimeout = Math.*min*(notReadyTimeout, **this**.client.connectionDelay(node, now));  }  }  // 将需要发送的消息封装在RecordBatch对象中  Map<Integer, List<RecordBatch>> batches = **this**.accumulator.drain(cluster,  result.readyNodes,  **this**.maxRequestSize,  now);  //将发送消息封装到请求对象中  List<ClientRequest> requests = createProduceRequests(batches, now);  //发送消息(其实是创建了一个发送对象,然后压入了)  **for** (ClientRequest request : requests)  client.send(request, now);  **this**.client.poll(pollTimeout, now);  } |

### NetWorkClient中的poll方法

|  |
| --- |
| @Override  **public** List<ClientResponse> poll(**long** timeout, **long** now) {  //检查metadata是否需要更新  **long** metadataTimeout = metadataUpdater.maybeUpdate(now);  **try** {  //使用kafka自定义的selector选择已经准备好的通道  **this**.selector.poll(Utils.*min*(timeout, metadataTimeout, requestTimeoutMs));  } **catch** (IOException e) {  ***log***.error("Unexpected error during I/O", e);  }  // process completed actions  **long** updatedNow = **this**.time.milliseconds();  List<ClientResponse> responses = **new** ArrayList<>();  //遍历已经发出的请求,并检查请求是否需要等待响应,如果不需要等待响应就自己创建一个  //response对象作为返回对象,如果需要等待服务端响应,那么将该请求压入inFlightRequest  //队列中,当返回响应时做后续处理  handleCompletedSends(responses, updatedNow);  //遍历已经收到的响应信息,并从inFlightRequest队列中找到对应的Request信息,解析响应,  //然后判断该响应是否为更新metadata的响应,如果是那么根据得到的响应信息更新metadata  //对象,并将metadata的version加1,如果不是那么就根据响应创建一个response对象  handleCompletedReceives(responses, updatedNow);  //当和一个节点断开连接后,将该节点对应的connection信息更新为未连接状态  handleDisconnections(responses, updatedNow);  //当和某个节点的连接成功后,将 该节点对应的connection信息更新为已连接状态  handleConnections();  handleTimedOutRequests(responses, updatedNow);  // invoke callbacks  **for** (ClientResponse response : responses) {  **if** (response.request().hasCallback()) {  **try** {  response.request().callback().onComplete(response);  } **catch** (Exception e) {  ***log***.error("Uncaught error in request completion:", e);  }  }  }  **return** responses;  } |

### kafka自定义selector的poll方法

|  |
| --- |
| **public** **void** poll(**long** timeout) **throws** IOException {  **if** (timeout < 0)  **throw** **new** IllegalArgumentException("timeout should be >= 0");  clear();  **if** (hasStagedReceives())  timeout = 0;  /\* check ready keys \*/  **long** startSelect = time.nanoseconds();  **int** readyKeys = select(timeout);  **if** (readyKeys > 0) {  Set<SelectionKey> keys = **this**.nioSelector.selectedKeys();  Iterator<SelectionKey> iter = keys.iterator();  **while** (iter.hasNext()) {  SelectionKey key = iter.next();  iter.remove();  KafkaChannel channel = channel(key);  **try** {  /\* complete any connections that have finished their handshake \*/  **if** (key.isConnectable()) {  channel.finishConnect();  **this**.connected.add(channel.id());  **this**.sensors.connectionCreated.record();  }  /\* if channel is not ready finish prepare \*/  **if** (channel.isConnected() && !channel.ready())  channel.prepare();  /\* if channel is ready read from any connections that have readable data \*/  **if** (channel.ready() && key.isReadable() && !hasStagedReceive(channel)) {  NetworkReceive networkReceive;  **while** ((networkReceive = channel.read()) != **null**)  addToStagedReceives(channel, networkReceive);  }  /\* if channel is ready write to any sockets that have space in their buffer and for which we have data \*/  **if** (channel.ready() && key.isWritable()) {  Send send = channel.write();  **if** (send != **null**) {  **this**.completedSends.add(send);  **this**.sensors.recordBytesSent(channel.id(), send.size());  }  }  /\* cancel any defunct sockets \*/  **if** (!key.isValid()) {  close(channel);  **this**.disconnected.add(channel.id());  }  } **catch** (Exception e) {  String desc = channel.socketDescription();  **if** (e **instanceof** IOException)  ***log***.debug("Connection with {} disconnected", desc, e);  **else**  ***log***.warn("Unexpected error from {}; closing connection", desc, e);  close(channel);  **this**.disconnected.add(channel.id());  }  }  }  addToCompletedReceives();  **long** endIo = time.nanoseconds();  **this**.sensors.ioTime.record(endIo - endSelect, time.milliseconds());  maybeCloseOldestConnection();  } |