

Kafka

- [Архитектура](#)
- [Kafka: введение](#)
- [Установка](#)
- [Topics](#)
- [Python + Kafka](#)

??????????

Apache Kafka — opensource платформа хранения и потоковой передачи событий в режиме реального времени.

Событийная архитектура (Event driven architecture, EDA): шаблон, в котором производство, обнаружение, потребление и реакция на события играют центральную роль. Преимущества:

- Асинхронное взаимодействие, позволяющее функционировать системе при неустойчивой работе части компонентов
- Увеличение гибкости
- Снижение риска каскадных ошибок

Недостатки:

- Согласованность данных.
- Задержки между изменением данных и получением информации об изменении.
- Повышенные операционные издержки

Примеры задач EDA:

Объединяет создание значительных объемов данных, которые асинхронно получают из различных источников и анализируются позже. Примеры:

- анализ активности пользователей
- сбор логов в задачах администрирования
- детекция мошенничества
- предсказание аварии железа

Ключевые компоненты

Кластерная структура стандартная:



Производители создают и отправляют данные, брокеры распределяют данные по потребителям, потребители получают и обрабатывают данные.

Процессы внутри одного брокера:

Topic	Partition	Key	Group	Consumer
-------	-----------	-----	-------	----------

Точка	Критерий выбора	Дополнительно
Topic	Topic name	Задается Producer при отправке сообщения. Обязательный параметр
Partition		Может задаваться Producer при отправке сообщения для определения Partition. Необязательный параметр.
Key		Выбор раздела по hash функции от ключа. Необязательный параметр. Но не гарантируется и зависит от количества разделов и количества ключей. Если не указан ни Key, ни Partition, то сообщения отправляются всем Consumer по кругу
Group		
Consumer		

Offset

Смещение точки неп прочитанного сообщения. В рамках Partition. Commit = фиксация.

Группы

- Сообщение из одной partition не может поступать нескольким consumer в пределах группы
- Один consumer может получать сообщения из нескольких partition
- При нескольких группах consumer, каждая группа будет получать все сообщения
- Одиночный consumer без группы равносителен группе из одного пользователя

Kafka: ????????

Сообщение

- Ключ - метаданные для управления записью сообщения в разделы
- Схема - способ разобрать сообщение. Может быть JSON, XML, Apache Avro
- Смещение - точка считывания, добавляется Kafka

Topic - центр объединения данных.

Иерархия распределения данных: Topic - Partition - message. Каждая partition связана с одним единственным потребителем (consumer). Упорядочиваются в пределах Partition.

- Сообщение одной partition не может поступать нескольким consumer в пределах группы

Производители

Потребители

Могут объединяться в группы. Чтение каждого раздела только одним членом группы. Групп может быть много. Если в одной группе consumer больше чем partition то оставшиеся простаивают. Принадлежность - какой раздел какому потребителю.

Брокер

Отдельный сервер Kafka - брокер. Объединяются в кластер, один из брокеров - контроллер. Если раздел несколькими брокерами, то происходит репликация. Основной брокер - ведущий. Механизмы репликации только в пределах одного кластера. MirrorMaker - репликация между кластерами.

broker.id	Целочисленный идентификатор, с 0, уникальный.
listeners	протокол://имя_хоста:порт Перечисляются через запятую. Имя хоста: конкретный ip - соответствующий интерфейс, 0.0.0.0 - все интерфейсы, не указан - интерфейс по умолчанию. Протокол: PLAINTEXT, SSL Если порт менее 1024 то Kafka от имени root,
log.dirs	Директории размещения логов.

--	--

Python: kafka-python

?????????

Kafka с web интерфейсом:

```
services:
  kafka:
    image: apache/kafka:4.2.0
    container_name: kafka
    hostname: kafka
    restart: unless-stopped

    ports:
      - "9092:9092"    # INTERNAL (docker network)
      - "29092:29092" # EXTERNAL (LAN access)

    environment:
      # =====
      # KRaft
      # =====
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093

      # =====
      # Listeners
      # =====
      KAFKA_LISTENERS: INTERNAL://:9092,EXTERNAL://:29092,CONTROLLER://:9093
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka:9092,EXTERNAL://192.168.1.195:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER

      # =====
      # Single-node safety
      # =====
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

```
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
```

```
# Optional tuning
```

```
KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
```

```
volumes:
```

```
- ./kafka_data:/var/lib/kafka/data
```

```
kafka-ui:
```

```
image: provectuslabs/kafka-ui:latest
```

```
container_name: kafka-ui
```

```
restart: unless-stopped
```

```
ports:
```

```
- "8080:8080"
```

```
environment:
```

```
KAFKA_CLUSTERS_0_NAME: local
```

```
KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092
```

```
depends_on:
```

```
- kafka
```

На <https://kafka.apache.org/quickstart/> есть информация по запуску без docker

Проверка внутренними средствами

Создание темы

```
docker exec -it kafka /opt/kafka/bin/kafka-topics.sh \
  --create \
  --topic demo \
  --bootstrap-server kafka:9092 \
  --partitions 3 \
  --replication-factor 1
```

Запуск продюсера (сервер на 1.120)

```
docker exec -it kafka /opt/kafka/bin/kafka-console-producer.sh \
  --topic demo \
```

```
--bootstrap-server 192.168.1.120:29092
```

Запуск консьюмера (сервер на 1.120)

```
docker exec -it kafka /opt/kafka/bin/kafka-console-consumer.sh \  
  --topic demo \  
  --from-beginning \  
  --bootstrap-server 192.168.1.120:29092
```

Topics

Тема

Сообщение:

- значение (value) - данные, обязательное
- (partition number)
- ключ (key) -

Создание темы (topic)

Topics / Create

Topic Name *

firsttopic

Number of Partitions *

Number of Partitions

Cleanup policy

Delete

Min In Sync Replicas

Min In Sync Replicas

Replication Factor

Replication Factor

Time to retain data (in ms)

Time to retain data (in ms)

12 hours

1 day

2 days

7 days

4 weeks

Max size on disk in GB

Not Set

Maximum message size in bytes

Maximum message size

Custom parameters

+ Add Custom Parameter

Cancel

Create topic

Поле	Тип данных	Описание
Topic Name	Строка	Название темы.

Поле	Тип данных	Описание
Number of partitions	Целое число	Количество разделов — управление производительностью и масштабируемостью кластера. Единицы параллелизма в Kafka. Разделы определяют скорость и масштабирование. • Для производителей (Producers): несколько разделов позволяет отправлять данные параллельно. Больше разделов — выше общая пропускная способность на запись. • Для потребителей (Consumers): ключевой момент. В одной группе потребителей (consumer group) каждый раздел может обрабатываться только одним потребителем. Количество разделов определяет максимальное число потребителей в группе, которые могут работать параллельно. Пример: для 10 разделов можно запустить до 10 потребителей, и каждый будет обрабатывать свой раздел. Если потребителей больше, чем разделов, то лишние потребители будут простаивать и тратить ресурсы впустую . Дополнительные факторы • Гарантируется порядок сообщений только внутри одного раздела. Если критичен порядок событий (например, все действия одного пользователя должны обрабатываться последовательно), используется ключ сообщения (key). Все сообщения с одним ключом всегда попадают в один и тот же раздел, сохраняя порядок. • Каждый раздел это доп. нагрузка (больше файлов, больше метаданных, больше времени на выборы лидера). Слишком

Поле	Тип данных	Описание
Cleanup policy	Список. • Delete • Compact • Compact, Delete	Политика управления сроком жизни данных. По умолчанию delete. Политика delete (Удаление) Удаление старых данных по истечении заданного времени или при превышении лимита размера. Как это работает: удаление когда старше retention.ms (по умолчанию 7 дней), или когда общий размер раздела превышает retention.bytes. Происходит в фоновом режиме с заданным интервалом. Когда применять: потоки событий и логов, где важна вся история. Например, логи приложений, метрики, трекинговые события. Здесь каждое сообщение самоценно, и старые данные со временем теряют актуальность . Политика compact (Сжатие) Эта политика работает по принципу "сохранить только последнее известное состояние" для каждого ключа сообщения . Как это работает: В фоновом режиме для каждого уникального ключа (key) в разделе остается только последнее по времени сообщение, а все промежуточные удаляет. Сообщения обязаны иметь ключ (key), без ключа будут удалены при сжатии. Когда применять: для хранения состояние или конфигурации. Например: • Таблица с актуальными данными пользователей (адрес, почта) • Информация о текущем статусе заказа. • Список доступных товаров на складе. • Внутренний топик Kafka __consumer_offsets, где хранятся текущие смещения потребителей, по умолчанию использует именно эту политику . Комбинированная политика delete,compact Как это работает: сначала удаляются старые данные по времени/размеру, а затем сжимается оставшийся лог. Это полезно, когда нужно ограничить общий объем данных и гарантировать, что для каждого

Поле	Тип данных	Описание
Min in sync replicas	Целое число	Минимальное количество брокеров-подписчиков, которые должны подтвердить запись, чтобы она считалась успешной. Иначе — это порог, защищающий от потери данных при сбое брокеров. Как это работает (в связке с ack) Работает в паре с настройкой продюсера acks (подтверждения). У топика replication.factor = 3 (3 копии данных на разных брокерах), а min.insync.replicas = 2. • Продюсер отправляет сообщение с настройкой acks=all (или acks=-1), что означает: "Жди подтверждения от всех реплик, которые сейчас в синхронизации". • Брокер-лидер принимает сообщение и ждет, пока его сохранят у себя как минимум 2 брокера (лидер + 1 реплика-подписчик). • Только когда эти 2 подтверждения получены, лидер отправляет продюсеру ответ об успешной записи. Ключевой момент: если в кластере внезапно упадут 2 брокера из 3, и в живых останется только 1 (лидер), то условие min.insync.replicas=2 перестанет выполняться. В этом случае брокер откажется принимать новые сообщения и вернет продюсеру ошибку NotEnoughReplicasException. Компромисс: Надежность vs Доступность Это выбор между Consistency (согласованностью) и Availability (доступностью) из теоремы CAP. Высокое значение (например, 2 или 3): □ Плюс: Максимальная надежность. Вы гарантированно не потеряете данные, даже если несколько брокеров выйдут из строя. □ Минус: Риск недоступности. Если выйдет из строя больше брокеров, чем min.insync.replicas, топик станет read-only, пока не починят брокеры или не уменьшите параметр

Поле	Тип данных	Описание
Replication factor	Целое число	Фундамент отказоустойчивости. Определяет, сколько всего копий каждого раздела будет храниться в кластере Kafka. Что это означает на практике Есть топик с replication.factor = 3. Это значит, что каждый раздел этого топика будет физически скопирован на 3 разных брокера. Для каждого раздела Kafka выбирает: • 1 Лидера (Leader) — брокер, который обрабатывает все чтения и записи для этого раздела. • N-1 Подписчиков (Followers) — брокеры, которые просто копируют данные у лидера и ждут своего часа. Репликация происходит на уровне разделов, а не топиков целиком. Если в топике 10 разделов и replication.factor=3, то в кластере будет храниться 30 копий этих разделов (10 разделов × 3 реплики). Как это работает: • Продюсер пишет данные → они попадают к лидеру раздела. • Лидер сохраняет данные у себя и реплицирует их на всех подписчиков. • Подписчики копируют данные и отправляют лидеру подтверждение. • Лидер ждет нужное количество подтверждений (то самое min.insync.replicas) и отвечает продюсеру. Что происходит при сбое: • Если лидер падает, Kafka выбирает нового лидера из числа синхронизированных подписчиков (In-Sync Replicas, ISR). • Если падает подписчик, он просто выпадает из ISR, но данные не теряются, пока жив лидер и хотя бы min.insync.replicas

Поле	Тип данных	Описание									
Time to retain data	Целое число, ms	определяет время хранения сообщения в топике перед удалением. По умолчанию 7 дней (168 часов). Есть два уровня настройки: на уровне брокера и на уровне конкретного топика. Удаление происходит по сегментам, а не по сообщениям. Kafka не удаляет отдельные сообщения. Лог раздела разбит на сегменты (segments). Параметр retention.ms определяет, когда закрытый (неактивный) сегмент становится кандидатом на удаление. Что это значит на практике: Активный сегмент (тот, в который сейчас пишутся данные) никогда не удаляется, даже если его сообщения «просрочены». Проверка на удаление запускается периодически (по умолчанию каждые 5 минут), а не мгновенно. Дополнительная задержка в 1 минуту (file.delete.delay.ms) перед физическим удалением файла. Пример из практики (низкая нагрузка) Вы хотите хранить данные 1 день, но у вас мало сообщений. Активный сегмент не «закрывается» (не достигает лимита по размеру segment.bytes или времени segment.ms), поэтому он живет долго. Результат: сообщения могут храниться до 2 дней (1 активный сегмент + 1 неактивный), хотя вы задали retention.ms=1 день. Сравнение с retention.bytes Kafka позволяет комбинировать два независимых ограничения: <table><tr><td>Параметр</td><td>Ограничивает</td><td>Когда срабатывает</td></tr><tr><td>retention.ms</td><td>время хранения</td><td>Сегмент старше указанного времени</td></tr><tr><td>retention.bytes</td><td>размер раздела</td><td>Суммарный размер сегментов превышает лимит</td></tr></table> Практические рекомендации Логи приложений 3–7 дней (достаточно для отладки)	Параметр	Ограничивает	Когда срабатывает	retention.ms	время хранения	Сегмент старше указанного времени	retention.bytes	размер раздела	Суммарный размер сегментов превышает лимит
Параметр	Ограничивает	Когда срабатывает									
retention.ms	время хранения	Сегмент старше указанного времени									
retention.bytes	размер раздела	Суммарный размер сегментов превышает лимит									

Поле	Тип данных	Описание
Maximum message size in bytes	Целое число, bytes	Потолок одного пакета. Но пакеты могут объединяться, так что потолок может быть превышен.
Custom parameters	Разные	Дополнительные параметры для тонкой настройки.

Python + Kafka

Библиотеки

- `confluent_kafka` от разработчиков
- `aiokafka` асинхронная библиотека

confluent_kafka

AdminClient

Модуль автоматизации административных задач.

Настройки:

```
from confluent_kafka.admin import AdminClient, NewTopic

config = {
    'bootstrap.servers': '192.168.1.195:29092'
}

admin_client=AdminClient(config)
```

Создание темы:

```
def create_partition():
    topic = 'test_topic'
    new_topic = NewTopic(topic, num_partitions=2, replication_factor=1)
    futures = admin_client.create_topics([new_topic])
    for topic_name, future in futures.items():
        try:
            future.result()
            print(f"Created {topic_name} successfully.")
        except Exception as e :
            print(f'Error in topic {topic_name} creation: ', e)
```

Топик в списке появляется не сразу, а где-то через 30 секунд.

Список тем:

```
def list_topics():
    metadata = admin_client.list_topics(timeout=10)
    print("Существующие топики:", list(metadata.topics.keys()))
```

consumer.py

```
from confluent_kafka import Consumer, KafkaException

config = {
    'bootstrap.servers': '192.168.1.195:29092',
    'group.id' : 'my_group'
}

consumer = Consumer(config)

if __name__ == '__main__':
    topic = 'test_topic'
    timeout_s = 1
    consumer.subscribe([topic])

    try:
        while True:
            msg = consumer.poll(timeout=timeout_s)
            if msg is None:
                continue
            if msg.error():
                raise KafkaException(msg.error())
            else:
                print(f'Input: {msg.value().decode()}')
    finally:
        consumer.close()
```

Если consumer в одной группе и количество partitions не меньше количества consumers, то сообщения распределяются между ними. Если partitions будет меньше consumers (которые в одной группе), то оставшаяся часть будет простаивать.

Функция consumer.subscribe получает список топиков, то есть одним consumer можно подписаться на несколько.

Если при двух и более partitions для одного из partitions отваливается consumer то Kafka автоматически проводит ребалансировку в пределах группы. После подключения опять произойдет ребалансировка.

producer.py

```
import time
from confluent_kafka import Producer

def delivery_report(err, msg):
    if err is not None:
        print(f"Message failed: {err}")
        raise err
    else:
        print(f'Message {msg.value()} in topic {msg.topic()} delivered')

config = {
    'bootstrap.servers': '192.168.1.195:29092',
    'message.timeout.ms' : 5000
}

producer = Producer(config)

def send_message(topic_: str, message_: str):
    producer.produce(topic_, value=message_, callback=delivery_report)
    producer.flush()

if __name__ == '__main__':
    topic = 'test_topic'
    message = 'Producer 1'

    i = 0
    while True:
        send_message(topic, f'{message} # {i}')
        i += 1
        time.sleep(1)
```

Распределение сообщений по ключам

Для распределения по консюмерам используется хэш функция от ключа и количества партиций. При коротких ключах может быть коллизия. Довольно неприятная и

сложнообходимая ситуация. Проще указать номер partition (int).

```
import time
from confluent_kafka import Producer

def delivery_report(err, msg):
    if err is not None:
        print(f"Message failed: {err}")
        raise err
    else:
        print(f'Message {msg.value()} in topic {msg.topic()} delivered')

config = {
    'bootstrap.servers': '192.168.1.195:29092',
    'message.timeout.ms' : 5000
}

producer = Producer(config)

def send_message(topic_: str, message_: str, part_: int | None = None):
    producer.produce(topic_, partition=part_, value=message_, callback=delivery_report)
    producer.flush()

if __name__ == '__main__':
    topic = 'test_topic'
    message = 'Producer 1'

    i = 0
    while True:
        part1 = 0
        part2 = 1
        send_message(topic, message_=f'key={key1} {message} # {i}', part_=part1)
        send_message(topic, message_=f'key={key2} {message} # {i}', part_=part2)
        i += 1
        time.sleep(1)

#===== consumer
=====

from confluent_kafka import Consumer, KafkaException
```

```

config = {
    'bootstrap.servers': '192.168.1.195:29092',
    'group.id' : 'my_group'
}

consumer = Consumer(config)

if __name__ == '__main__':
    topic = 'test_topic'
    timeout_s = 1
    consumer.subscribe([topic])

    try:
        while True:
            msg = consumer.poll(timeout=timeout_s)
            if msg is None:
                continue
            if msg.error():
                raise KafkaException(msg.error())
            else:
                print(f'Input: {msg.value().decode()}')
    finally:
        consumer.close()

```

Транзакции

В config для consumer добавляется элемент 'enable.auto.commit': False

И затем consumer.commit()

Сериализация данных

Через json

```

while True:
    obj = {
        'id': i,
        'value': randomword(8),
        'timestamp': datetime.now().isoformat(timespec='seconds'),
    }

```

```

send_message(topic, json.dumps(obj))
i += 1
time.sleep(1)

=====

    str_msg = msg.value().decode('utf-8')
    obj = json.loads(str_msg)
    print(f'id={obj.get("id")}, timestamp={obj.get("timestamp")}')

```

aiokafka

consumer.py

```

from aiokafka import AIOKafkaConsumer
import asyncio

async def consume():
    consumer = AIOKafkaConsumer(
        'test_first_one',
        bootstrap_servers='192.168.1.195:29092'
    )
    await consumer.start()
    try:
        async for msg in consumer:
            print(f"Consumed message {msg.value.decode()}")
    finally:
        await consumer.stop()

if __name__ == "__main__":
    asyncio.run(consume())

```

producer.py

```

from aiokafka import AIOKafkaProducer
import asyncio

```

```
async def send():
    producer = AIOKafkaProducer(
        bootstrap_servers="192.168.1.195:29092"
    )
    topik = 'test_first_one'
    message = 'Hello from python!'
    await producer.start()
    i = 0
    try:
        while True:
            bytes_msg = f'{message} {i}'.encode('utf-8') # convert string into byte
            await producer.send_and_wait(topik, bytes_msg)
            i += 1
            print(f'Message {i} sent.')
            await asyncio.sleep(1)
    finally:
        await producer.stop()

if __name__ == '__main__':
    asyncio.run(send())
```