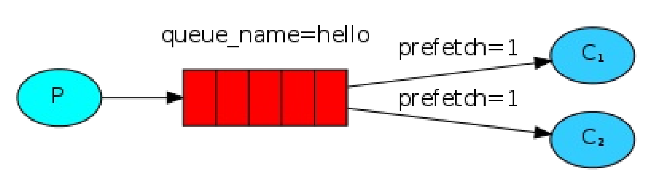
# Правила доставки

## Емкость потребителя

В предыдущем видео мы видели: когда receiver запустился, он сразу принял несколько сообщений из очереди. Сообщения выполнялись 5 секунд, но, несмотря на это, receiver все равно продолжал набирать задачи.

В этом примере это нестрашно. Но если бы задач были сотни тысяч, а receiver начал жадно их потреблять, он просто забился бы. При этом запуск новых receiver не поможет, так как первый уже нахватал задач и другим просто нечего будет делать.

Так происходит потому, что RabbitMQ просто отправляет сообщение, когда оно попадает в очередь. Он не учитывает количество неподтвержденных сообщений для потребителя. Он просто слепо отправляет каждое n-е сообщение n-му потребителю.



Чтобы избежать этого, мы можем использовать метод set_qos с параметром prefetch_count = 1. Это говорит RabbitMQ не отправлять более одного сообщения потребителю за раз. Или, другими словами, не отправлять новое сообщение потребителю, пока он не обработает и не подтвердит предыдущее. Вместо этого сообщение уйдет следующему работнику, который еще не занят.

**Пример: connectors/rabbit/qos/**

connectors/rabbit/qos/receiver.py

```
import asyncio
import os

from aio_pika import connect, IncomingMessage



async def create_connection():
    return await connect(url=os.getenv("RABBITMQ_URL"))



async def on_message(message: IncomingMessage):
    print("Before sleep!")
    async with message.process():
        await asyncio.sleep(5)
    print("After sleep!")



async def receiver():
    connection = await create_connection()
    channel = await connection.channel()
    await channel.set_qos(prefetch_count=1)
    queue = await channel.declare_queue("ack")
    await queue.consume(on_message)



if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.create_task(receiver())

    print(" [*] Waiting for messages.")
    loop.run_forever()
```

Запустить пример нужно в разных терминалах в mercury:

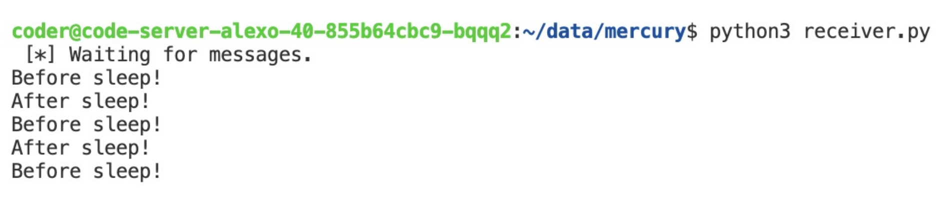
```
python3 connectors/rabbit/qos/receiver.py
python3 connectors/rabbit/qos/sender.py
```

С помощью...

```
await channel.set_qos(prefetch_count=1)
```

...мы установили, что одновременно receiver может обрабатывать не больше одной задачи. Таким образом, можно управлять количеством параллельных корутин, которые будут запускаться в receiver.

Запустите несколько раз connectors/rabbit/qos/sender.py, а затем запустите receiver.py. Вы должны увидеть последовательность сообщений:



```
coder@code-server-alexo-40-855b64cbc9-bqqq2:~/data/mercury$ python3 receiver.py
 [*] Waiting for messages.
Before sleep!
After sleep!
Before sleep!
After sleep!
Before sleep!
```

Попробуйте увеличить prefetch_count — количество параллельно запущенных корутин будет равно параметру prefetch_count.

## Exchanges

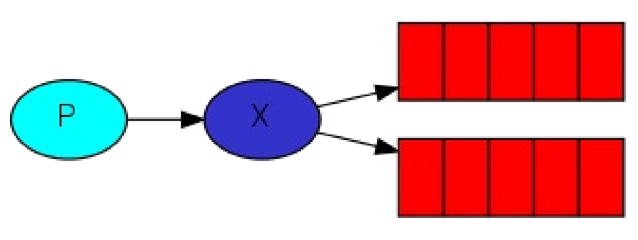
В предыдущих примерах мы рассматривали очереди обработки данных. Каждая задача доставляется ровно одному потребителю. В примерах ниже мы изменим режим доставки и отправим сообщения нескольким потребителям. Этот паттерн проектирования называется pub/sub. Он позволяет очень гибко добавлять новые компоненты в систему.

Представьте картину: президент страны объявляет по телевизору о нерабочей неделе. Президент — sender, телевизор — шина событий, слушатели — потребители, которые что-то делают с этой информацией. Кто-то из слушателей обрадуется, кто-то огорчится и куда-нибудь уедет, а кто-то начнет планировать финансы из-за внезапного «отпуска». Так и в системах, которые построены на основе паттерна pub/sub: кто-то объявляет новость, эта новость рассылается через брокер сообщений, потребители принимают это сообщение и выполняют какие-то действия на его основе.

Чтобы изменить способ доставки сообщений в RabbitMQ, нужно использовать *exchanges.* Основная идея модели обмена сообщениями в RabbitMQ заключается в том, что sender никогда не отправляет сообщения напрямую в очередь. На самом деле, довольно часто производитель даже не знает, будет ли сообщение вообще доставлено в какую-либо очередь.

Вместо этого производитель может только отправлять сообщения в exchange. Exchange, с одной стороны, получает сообщения от производителей, а с другой, помещает их в очереди. Exchange должен точно знать, что делать с полученным сообщением.

Должен ли он быть добавлен в определенную очередь? Следует ли добавлять его во многие очереди? Или его следует выбросить? Правила для этого определяются *типом обмена.*



Типы обмена бывают:

DIRECT — отправит сообщение в очередь на основе routing_key *(см. следующие карточки)

FANOUT — отправит сообщение всем очередям, которые присоединены к exchange

TOPIC — отправит сообщение в очередь на основе routing_key. Отличие в том, что в routing_key можно указывать шаблон, например, *logs.**(см. следующие карточки)

HEADERS — роутинг происходит на основе заголовков, а не на основе routing_key (этот тип рассматривать не будем)

Чтобы объявить exchange, нужно вызвать:

```
logs_exchange = await channel.declare_exchange("logs", exchange_type)
```

На самом деле в предыдущих примерах мы уже пользовались exchange, но это был особый exchange без имени (имя равно пустой строке) — *channel.default_exchange.*

Выше упоминалось, что с помощью exchange можно определить, в какую очередь положить сообщение. Для этого нужно соединить exchange с сообщением (bind). Более того, очередь может быть ассоциирована с несколькими очередями. При этом, если exchange не будет соединен с какой-либо очередью, RabbitMQ просто удалит помещенное в exchange сообщение.

Чтобы соединить очередь с exchange, нужно вызвать:

```
await queue.bind(logs_exchange)
```

**Пример: connectors/rabbit/logs/**

Нужно построить систему логирования, но так, чтобы способы логирования можно было гибко добавлять.

Для примера рассмотрим 2 способа: вывод лога событий на экран, stdout, и запись события на диск. То есть в системе будет 4 компонента: генератор событий, брокер сообщений, stdout logger, file logger.

Генератор событий оставим такой же, как в предыдущих примерах, а потребителей сделаем 2: stdout logger и file logger.

## file logger

```
import asyncio
import os

from aio_pika import connect, IncomingMessage, ExchangeType



async def create_connection():
    return await connect(url=os.getenv("RABBITMQ_URL"))



async def on_message(message: IncomingMessage):
    async with message.process():
        with open('logs.txt', 'w') as fd:
            fd.write(str(message))
            fd.write('\n')



async def receiver():
    connection = await create_connection()
    channel = await connection.channel()
    logs_exchange = await channel.declare_exchange(
        "logs", ExchangeType.FANOUT
    )
    queue = await channel.declare_queue("logger")
    await queue.bind(logs_exchange)
    await queue.consume(on_message)



if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.create_task(receiver())

    print(" [*] Waiting for messages.")
    loop.run_forever()
```

## stdout logger

```
import asyncio
import os

from aio_pika import connect, IncomingMessage, ExchangeType



async def create_connection():
    return await connect(url=os.getenv("RABBITMQ_URL"))



async def on_message(message: IncomingMessage):
    async with message.process():
        print(message)



async def receiver():
    connection = await create_connection()
    channel = await connection.channel()
    logs_exchange = await channel.declare_exchange(
        "logs", ExchangeType.FANOUT
    )
    queue = await channel.declare_queue("logger")
    await queue.bind(logs_exchange)
    await queue.consume(on_message)



if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.create_task(receiver())

    print(" [*] Waiting for messages.")
    loop.run_forever()
```

Запустить пример нужно в разных терминалах в mercury:

```
python3 connectors/rabbit/logs/to_file.py
python3 connectors/rabbit/logs/to_stdout.py
python3 connectors/rabbit/logs/sender.py
```

Что тут происходит:

- объявили *logs_exchange*, в параметрах указали *имя* и ExchangeType.FANOUT
- объявили очередь await channel.declare_queue("logger")
- присоединили *logs_exchange* к *queue:* await queue.bind(logs_exchange)

Если запустить, то сообщения у нас опять будут приходить по очереди между двумя логгерами.

Это неудивительно, ведь мы никак не меняли поведение очереди. Мы настроили режим exchange — теперь он отправляет всем присоединенным очередям копию сообщения. Поэтому в logger нужно создавать свою очередь под себя.

Фиксировать имя очереди нельзя, так как если мы захотим запустить еще один logger, то сообщения опять начнуть распределяться между ними. Поэтому нужно генерировать название случайно при запуске. Также, чтобы не захламлять RabbitMQ, стоит удалять их после завершения logger. Для этого в Rabbit уже есть готовые механизмы:

- можно объявить очередь без имени, и RabbitMQ сам сгенерирует ей случайное название
- можно пометить очередь как временную — флаг exclusive при создании

**Пример создания временной очереди:**

```
# постоянная очередь без названия
queue = await channel.declare_queue()
# временная очередь без названия, которая существует в рамках соединения
queue = await channel.declare_queue(exclusive=True)
```

Для того, что пример заработал, нужно заменить строчку:

```
queue = await channel.declare_queue("logger")
# на
queue = await channel.declare_queue(exclusive=True)
```