# 爬虫中的请求管理

## 课程概要

* 请求管理简介
* 队列与Python
* 消息队列与Python
* 断点续爬与增量式设计
* 总结

## 1. 请求管理简介

### 1.1 请求管理业务

* 请求去重管理：防止重复请求
* 请求缓冲管理：临时存储请求
* 请求调度管理：派遣并控制请求调度顺序、请求优先级管理

### 1.2 请求去重管理

* 请求去重判断依据
  + 请求方法 post, get, str, 大小写转换
  + 请求地址(URL) str, 大小写转换
  + URL查询参数 str, [], {}, 把字典转换为字符串, 但字典是无序的, 多次转换可能会得到不同的结果, 需要在转换前进行排序的操作.
  + 请求体 post, put, str, [], {}, 排序并转换为字符串
* 去重方案：
  + 基于信息摘要算法求指纹的去重
  + 基于布隆过滤器的去重 原始数据必须是字符串, 需要把去重判断依据中的4种依据作为data传递到去重容器中,
* 请求数据处理：
  + 统一大小写（method、url）
  + URL查询参数排序（query）
  + 请求体排序（data）

#### 代码实现

在项目根目录中新建request\_manager的python包, 新建request\_manager/request\_filter的python包. 之前的去重可以看成是请求管理的工具模块, 可以新建一个工具模块 utils, 把filter\_class放到utils模块中. 目录结构如下.

└───reuquest\_manager

│ request.py

│ \_\_init\_\_.py

│

├───request\_filter

│ \_\_init\_\_.py

│

└───utils

│ \_\_init\_\_.py

│

└───filter\_class

│ bloomfilter.py

│ \_\_init\_\_.py

│

└───information\_summary\_filter

memory\_filter.py

mysql\_filter.py

redis\_filter.py

\_\_init\_\_.py

请求去重管理的逻辑处理的是请求去重的对象, 请求去重的对象有请求方法, 请求地址, url查询参数, 请求体这4个属性, 请求数据处理就是对请求对象的4个属性进行处理, 再交给去重方案进行去重处理. 所以首先要创建出来请求对象.

新建request\_manager/request.py

*# 构建一个请求对象*

**class** **Request**(object):

*# 请求对象有4种属性, 请求url, 请求方法, 查询字符串, 请求体*

**def** \_\_init\_\_(self, url, method="GET", query={}, body={}, name="request"):

self.url = url

self.method = method

*# 查询字符串必须为字典格式*

**if** **not** isinstance(query, dict):

**raise** **Exception**("query must be a dict")

self.query = query

*# 请求体也必须为字典格式*

**if** **not** isinstance(body, dict):

**raise** **Exception**("body must be a dict")

self.body = body

self.name = name

修改request\_manager/request\_filter/\_\_init\_\_.py, 实现请求去重的逻辑.

*# 实现请求去重的逻辑*

*# urllib.parse为python3的内置模块*

**import** **urllib.parse**

*# w3lib.url为第3方的模块, 需要pip安装, 优点是py2, py3通用.*

*# import w3lib.url*

**class** **RequestFilter**(object):

**def** \_\_init\_\_(self, filter\_obj):

*# 如何来判断请求是否已经处理过, 使用过滤器对请求进行判断, 所以首先要实例化一个过滤器对象*

self.filter\_obj = filter\_obj

**def** is\_exists(self, request\_obj):

*'''*

*判断请求是否已经处理过*

*return: True or False*

*'''*

*# request\_obj是一个类对象, 不能对对象进行去重, 而是对对象的所有属性进行去重, 把所有的属性整合成一个字符串类型的数据, 对这个数据进行去重判断.*

data = self.\_get\_request\_filter\_data(request\_obj)

*# 调用过滤器 filter\_obj 的 is\_exists 方法来对请求进行判断*

**return** self.filter\_obj.is\_exists(data)

**def** mark\_request(self, request\_obj):

*'''*

*标记已经处理过的请求对象*

*:param request\_obj:*

*:return: 标记*

*'''*

data = self.\_get\_request\_filter\_data(request\_obj)

**return** self.filter\_obj.save(data)

**def** \_get\_request\_filter\_data(self, request\_obj):

*'''*

*获取一个请求对象用来进行去重的数据*

*根据一个请求对象，处理它的4种属性, 转换为字符串, 然后再进行去重处理*

*:param request\_obj: 请求对象*

*:return: 转换后的字符串*

*'''*

*# 1.URL地址的处理*

*# HTTPS://WWW.BAIDU.com/S?wd=PYTHON&a=100&b=200*

*# HTTPS://WWW.BAIDU.com/S?wd=PYTHON&b=200&a=100*

*# 排序后的结果*

*# HTTPS://WWW.BAIDU.com/S?a=100&b=200&wd=PYTHON*

*# 处理的思路*

*# 1. 把协议和域名部分进行大小写统一，其他的保留原始大小写格式, 对于查询参数中的S, 大小写的功能是否相同, 需要对不同的网站进行不同的测试. 关于S的处理要放在具体爬虫的代码中.*

*# 2. 查询参数相同, 但是排列顺序不同的url是相同的url, 所以要对查询参数进行排序，然后和后面query中的查询参数进行合并. 查询参数区分大小写*

*# url中的查询参数, 转换为类似于[("wd","PYTHON"),("b",200),("a",100)]的格式, 再进行排序*

url = request\_obj.url

*# HTTPS://WWW.BAIDU.com/S?wd=PYTHON&a=100&b=200*

*# 使用urllib.parse.urlparse把url进行拆分*

\_parsed\_url = urllib.parse.urlparse(url)

*# ParseResult(scheme='https', netloc='WWW.BAIDU.com', path='/S', params='', query='wd=PYTHON&a=100&b=200', fragment='')*

*# 不带有查询参数的url地址*

url\_without\_query = \_parsed\_url.scheme + "://" + \_parsed\_url.hostname + \_parsed\_url.path

url\_without\_query = "{}://{}{}".format(\_parsed\_url.scheme, \_parsed\_url.hostname, \_parsed\_url.path)

*"""*

*# urllib.parse.parse\_qsl和urllib.parse.parse\_qs的用法*

*# qs表示查询参数, l表示列表*

*\_query = 'wd=PYTHON&a=100&b=200&b=300'*

*# 如果查询字符串中有相同的参数, parse\_qs会进行合并*

*urllib.parse.parse\_qs(\_query)*

*# {'wd': ['PYTHON'], 'a': ['100'], 'b': ['200', '300']}*

*# 如果查询字符串中有相同的参数, parse\_qsl不会进行合并*

*urllib.parse.parse\_qsl(\_query)*

*# [('wd', 'PYTHON'), ('a', '100'), ('b', '200'), ('b', '300')]*

*"""*

*# 把查询参数从字符串的形式分解为列表嵌套元组的形式, 以方便对其进行排序*

*# url\_query = urllib.parse.parse\_qs(\_parsed\_url.query)*

*# {'wd': ['PYTHON'], 'a': ['100'], 'b': ['200']}*

url\_query = urllib.parse.parse\_qsl(\_parsed\_url.query)

*# [('wd', 'PYTHON'), ('a', '100'), ('b', '200')]*

*# 2.method: "Get".upper()*

method = request\_obj.method.upper()

*# 3.query: str(sorted({}.items()))*

*# 查询参数可以在url地址中, 也可以作为单独的参数存在, 这里的query表示以字典或嵌套元组的列表格式单独存在的查询参数, {}, [(),()], 先进行排序, str([])*

*# 为了格式的统一, 这里的query限定使用字典的形式, 可以在request\_manager/request.py中对query进行限定*

*# 需要把url中的请求查询参数和query中的查询参数进行合并去重. 处理后的查询参数要么都放在url中, 要么都放在query独立的查询参数中*

*# 合并时重复查询参数的处理, url中的查询参数与单独的 query 中的查询参数可能没有冲突, 如query中的查询参数为{"c":100}, 也可能存在有冲突, 这就需要进行去重, 可以使用set来完成去重操作.*

*# 单独的查询参数, query是字典的数据类型, 使用items()转换为元组格式的 [(key, value),(key,value)] 数据类型, 然后根据key进行排序*

query = request\_obj.query.items()

*# 查询参数合并的处理方法一: 把url中的查询参数合并到query中.*

all\_query = sorted(query + url\_query)

*# 使用set来去除掉url\_query和query中key-value都重复的查询参数*

all\_query = sorted(set(list(query) + url\_query))

str\_query = str(all\_query)

*# 查询参数合并的处理方法二: 把query中的查询参数合并到url中*

*# url中的查询参数和单独的查询参数进行合并.*

all\_query = sorted(set(list(query) + url\_query))

*# print(all\_query)*

*# 带有完整查询参数的url地址*

url\_with\_query = url\_without\_query + "?" + urllib.parse.urlencode(all\_query)

*# 4.body: str(sorted({}.items()))*

str\_body = str(sorted(request\_obj.body.items()))

*# 方法一*

data = url\_without\_query + method + str\_query + str\_body

*# url method query body*

*# 方法二*

data = url\_with\_query + method + str\_body

*# url method body*

**return** data

在项目根目录中新建 request\_filter\_demo.py, 对请求去重进行测试. 测试时如果提示找不到文件, 就手动把代码上传到远程linux-server中,

**from** **request\_manager.request\_filter** **import** RequestFilter

**from** **request\_manager.utils.filter\_class** **import** get\_filter\_class

**from** **request\_manager.request** **import** Request

*# 创建出多个请求对象*

r1 = Request("https://www.baidu.com/s?wd=python")

*# 为了区分r1和r2, 分别给r1, r2添加name的参数*

r1.name = "r1"

*# r1和r2是相同的请求*

r2 = Request("https://www.baidu.com/s", query={"wd":"python"})

r2.name = "r2"

r3 = Request("HTTPS://www.Baidu.com/s?wd=python")

r3.name = "r3"

r4 = Request("https://www.baidu.com/S?wd=python")

r4.name = "r4"

r5 = Request("https://www.baidu.com/s?wd=python", query={"wd":"python"})

r5.name = "r5"

*# 大部分网站都不会设置查询参数中使用相同的key, 但有一部分网站使用这种设定, 所以还要对这种情况进行处理. 如果使用query={"wd":["python","go"]} 这种形式来传递相同key的查询参数, 还需要另外进行处理.*

r6 = Request("https://www.baidu.com/s?wd=python&wd=go", query={"wd":"python"})

r6.name = "r6"

r7 = Request("https://www.baidu.com/s?wd=go&wd=python", query={"wd":"python"})

r7.name = "r7"

*# 构建多个请求*

rs = [r1, r2, r3, r4, r5, r6, r7]

*# 创建过滤器对象. get\_filter\_class获取到的是一个类, 还要进行实例化*

filter\_obj = get\_filter\_class("memory")()

*# 创建请求过滤器对象*

request\_filter = RequestFilter(filter\_obj)

**for** r **in** rs:

**if** request\_filter.is\_exists(r):

**print**("请求重复", r.name)

**else**:

**print**("处理请求")

*# 请求处理完成后还要把它加入到处理过的队列中*

fp = request\_filter.mark\_request(r)

**print**("标记请求对象", r.name, fp)

运行 request\_filter\_demo.py, 对结果进行测试.

### 1.3 请求缓冲与调度管理

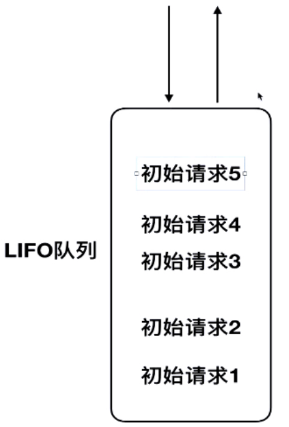
请求缓冲, 在请求被处理之前, 使用一个容器对请求进行一段时间的临时性的保存, 请求缓冲本质上相当于生成一个容器, 其中可以保存多个请求.

而请求调度则是控制如何从请求缓冲的容器中取出和保存请求, 控制添加和取出请求的时间和顺序. 所以可以把请求缓冲和调度管理放在一起来实现.

* 请求缓冲：队列（常用）

列表, 字典, 集合, 元组, 队列都是python中的容器, 元组不能修改. 其它的数据结构的操作比较复杂, 队列提供了保存和取出的操作, 与请求的操作类似, 队列是用来保存请求的最实用和最简单的python数据结构.

* 请求调度优先级类型：
  + 广度优先（FIFO队列）先进先出队列实现广度优先. 先处理初始请求1, 初始请求1处理的过程中会产生新的请求, 添加到队列的底部, 把初始请求1处理结束后才会去处理初始请求2, 初始请求2处理结束后再去处理初始请求3, 如此循环, 把原来所有的初始请求全部处理结束才会去处理新产生的请求. 相当于一层层的去处理不同层级的请求, 这就实现了广度优先.
  + 深度优先（LIFO队列）先进后出. 把最后添加的初始请求5及其产生的所有新的请求处理完成后才去处理之前添加的初始请求4. 先把一个请求的所有层级全部处理完, 再去处理其它请求的所有层级, 相当于深度优先. 但是从并发的角度来看, LIFO队列并不是严格意义上的深度优先. 想要实现严格意义上的深度优先, 需要对每一个初始请求设计一个队列, 只要某一个初始请求没有完全处理结束 , 就不会再去取新的初始请求.
  + 权重优先（优先级队列） 在添加请求时, 除了请求对象本身外, 还要添加一个权重值, 添加之后队列会自动把权重高的请求放在最前面优先进行处理.

## 2. 队列与Python

### 2.1 临时队列 (内存型队列)

队列更多的是使用在多线程或多进程中, 用于不同的线程或进程之间的通信. 所以更多的是结束一些异步模型来使用.

queue.Queue与gevent.queue.Queue二者归为一类, 是阻塞型的.

asyncio.Queue与tornado.queues.Queue二者归为一类, 是非阻塞型的.

#### python内置队列模块

queue, 用于多线程之间的通信. 队列对象（queue.Queue，queue.LifoQueue 或 queue.PriorityQueue）

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/queue/>

import queue

**pdir(queue)**

class:

LifoQueue: Variant of Queue that retrieves most recently added entries first.

PriorityQueue: Variant of Queue that retrieves open entries in priority order (lowest first).

Queue: Create a queue object with a given maximum size.

deque: deque([iterable[, maxlen]]) --> deque object

exception:

Empty: Exception raised by Queue.get(block=0)/get\_nowait().

Full: Exception raised by Queue.put(block=0)/put\_nowait().

function:

heappop: Pop the smallest item off the heap, maintaining the heap invariant.

heappush: heappush(heap, item) -> None. Push item onto heap, maintaining the heap invariant.

time: monotonic() -> float

**pdir(queue.Queue)**

function:

\_get:

\_init:

\_put:

\_qsize:

empty: Return True if the queue is empty, False otherwise (not reliable!).

full: Return True if the queue is full, False otherwise (not reliable!).

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue without blocking.

join: Blocks until all items in the Queue have been gotten and processed.

put: Put an item into the queue.

put\_nowait: Put an item into the queue without blocking.

qsize: Return the approximate size of the queue (not reliable!).

task\_done: Indicate that a formerly enqueued task is complete.

**help(queue.Queue.join)**

Help on function join in module queue:

join(self)

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the

queue. The count goes down whenever a consumer thread calls task\_done()

to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

**help(queue.Queue.empty)**

Help on function empty in module queue:

empty(self)

Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0

as a direct substitute, but be aware that either approach risks a race

condition where a queue can grow before the result of empty() or

qsize() can be used.

To create code that needs to wait for all queued tasks to be

completed, the preferred technique is to use the join() method.

**help(queue.Queue.full)**

Help on function full in module queue:

full(self)

Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n

as a direct substitute, but be aware that either approach risks a race

condition where a queue can shrink before the result of full() or

qsize() can be used.

**help(queue.Queue.get)**

Help on function get in module queue:

get(self, block=True, timeout=None)

Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default),

block if necessary until an item is available. If 'timeout' is

a non-negative number, it blocks at most 'timeout' seconds and raises

the Empty exception if no item was available within that time.

Otherwise ('block' is false), return an item if one is immediately

available, else raise the Empty exception ('timeout' is ignored

in that case).

**help(queue.Queue.get\_nowait)**

Help on function get\_nowait in module queue:

get\_nowait(self)

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise

raise the Empty exception.

**help(queue.Queue.join)**

Help on function join in module queue:

join(self)

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the

queue. The count goes down whenever a consumer thread calls task\_done()

to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

**help(queue.Queue.put)**

Help on function put in module queue:

put(self, item, block=True, timeout=None)

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default),

block if necessary until a free slot is available. If 'timeout' is

a non-negative number, it blocks at most 'timeout' seconds and raises

the Full exception if no free slot was available within that time.

Otherwise ('block' is false), put an item on the queue if a free slot

is immediately available, else raise the Full exception ('timeout'

is ignored in that case).

**help(queue.Queue.put\_nowait)**

Help on function put\_nowait in module queue:

put\_nowait(self, item)

Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available.

Otherwise raise the Full exception.

**help(queue.Queue.qsize)**

Help on function qsize in module queue:

qsize(self)

Return the approximate size of the queue (not reliable!).

**help(queue.Queue.task\_done)**

Help on function task\_done in module queue:

task\_done(self)

Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task,

a subsequent call to task\_done() tells the queue that the processing

on the task is complete.

If a join() is currently blocking, it will resume when all items

have been processed (meaning that a task\_done() call was received

for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items

placed in the queue.

**pdir(queue.LifoQueue)**

function:

\_get:

\_init:

\_put:

\_qsize:

empty: Return True if the queue is empty, False otherwise (not reliable!).

full: Return True if the queue is full, False otherwise (not reliable!).

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue without blocking.

join: Blocks until all items in the Queue have been gotten and processed.

put: Put an item into the queue.

put\_nowait: Put an item into the queue without blocking.

qsize: Return the approximate size of the queue (not reliable!).

task\_done: Indicate that a formerly enqueued task is complete.

**pdir(queue.PriorityQueue)**

function:

\_get:

\_init:

\_put:

\_qsize:

empty: Return True if the queue is empty, False otherwise (not reliable!).

full: Return True if the queue is full, False otherwise (not reliable!).

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue without blocking.

join: Blocks until all items in the Queue have been gotten and processed.

put: Put an item into the queue.

put\_nowait: Put an item into the queue without blocking.

qsize: Return the approximate size of the queue (not reliable!).

task\_done: Indicate that a formerly enqueued task is complete.

#### [gevent](http://www.gevent.org/api/gevent.queue.html)中的队列模块

<http://www.gevent.org/api/gevent.queue.html>

import gevent.queue

**pdir(gevent.queue)**

descriptor:

get\_hub: class cython\_function\_or\_method with getter

class:

Channel: Channel(maxsize=1)

ItemWaiter: ItemWaiter(item, queue)

JoinableQueue: JoinableQueue(maxsize=None, items=(), unfinished\_tasks=None)

LifoQueue: A subclass of :class:`Queue` that retrieves most recently added entries first.

PriorityQueue: A subclass of :class:`Queue` that retrieves entries in priority order (lowest first).

Queue: Queue(maxsize=None, items=(), \_warn\_depth=2)

Timeout: Timeout(seconds=None, exception=None, ref=True, priority=-1)

UnboundQueue: UnboundQueue(maxsize=None, items=())

Waiter: Waiter(hub=None)

exception:

Empty: Exception raised by Queue.get(block=0)/get\_nowait().

Full: Exception raised by Queue.put(block=0)/put\_nowait().

InvalidSwitchError: Raised when the event loop returns control to a greenlet in an

function:

getcurrent:

**pdir(gevent.queue.Queue)**

descriptor:

\_create\_queue: class cython\_function\_or\_method with getter, Queue.\_create\_queue(self, items=())

\_format: class cython\_function\_or\_method with getter, Queue.\_format(self)

\_get: class cython\_function\_or\_method with getter, Queue.\_get(self)

\_peek: class cython\_function\_or\_method with getter, Queue.\_peek(self)

\_put: class cython\_function\_or\_method with getter, Queue.\_put(self, item)

\_unlock: class cython\_function\_or\_method with getter, Queue.\_unlock(self)

copy: class cython\_function\_or\_method with getter, Queue.copy(self)

empty: class cython\_function\_or\_method with getter, Queue.empty(self) -> bool

full: class cython\_function\_or\_method with getter, Queue.full(self) -> bool

get: class cython\_function\_or\_method with getter, Queue.get(self, block=True, timeout=None)

get\_nowait: class cython\_function\_or\_method with getter, Queue.get\_nowait(self)

hub: class getset\_descriptor with getter, setter, deleter

maxsize: class getset\_descriptor with getter, setter, deleter

peek: class cython\_function\_or\_method with getter, Queue.peek(self, block=True, timeout=None)

peek\_nowait: class cython\_function\_or\_method with getter, Queue.peek\_nowait(self)

put: class cython\_function\_or\_method with getter, Queue.put(self, item, block=True, timeout=None)

put\_nowait: class cython\_function\_or\_method with getter, Queue.put\_nowait(self, item)

qsize: class cython\_function\_or\_method with getter, Queue.qsize(self) -> Py\_ssize\_t

queue: class getset\_descriptor with getter, setter, deleter

function:

next:

与queue.Queue和asyncio.Queue的不同是, 此处的get和get\_nowait会把数据从队列中删除并返回, 而peek和peek\_nowait则是只返回不删除.

pdir(gevent.queue.LifoQueue)

descriptor:

\_create\_queue: class cython\_function\_or\_method with getter, LifoQueue.\_create\_queue(self, items=())

\_format: class cython\_function\_or\_method with getter, Queue.\_format(self)

\_get: class cython\_function\_or\_method with getter, LifoQueue.\_get(self)

\_peek: class cython\_function\_or\_method with getter, LifoQueue.\_peek(self)

\_put: class cython\_function\_or\_method with getter, LifoQueue.\_put(self, item)

\_unlock: class cython\_function\_or\_method with getter, Queue.\_unlock(self)

copy: class cython\_function\_or\_method with getter, Queue.copy(self)

empty: class cython\_function\_or\_method with getter, Queue.empty(self) -> bool

full: class cython\_function\_or\_method with getter, Queue.full(self) -> bool

get: class cython\_function\_or\_method with getter, Queue.get(self, block=True, timeout=None)

get\_nowait: class cython\_function\_or\_method with getter, Queue.get\_nowait(self)

hub: class getset\_descriptor with getter, setter, deleter

maxsize: class getset\_descriptor with getter, setter, deleter

peek: class cython\_function\_or\_method with getter, Queue.peek(self, block=True, timeout=None)

peek\_nowait: class cython\_function\_or\_method with getter, Queue.peek\_nowait(self)

put: class cython\_function\_or\_method with getter, Queue.put(self, item, block=True, timeout=None)

put\_nowait: class cython\_function\_or\_method with getter, Queue.put\_nowait(self, item)

qsize: class cython\_function\_or\_method with getter, Queue.qsize(self) -> Py\_ssize\_t

queue: class getset\_descriptor with getter, setter, deleter

function:

next:

pdir(gevent.queue.PriorityQueue)

descriptor:

\_create\_queue: class cython\_function\_or\_method with getter, PriorityQueue.\_create\_queue(self, items=())

\_format: class cython\_function\_or\_method with getter, Queue.\_format(self)

\_get: class cython\_function\_or\_method with getter, PriorityQueue.\_get(self)

\_peek: class cython\_function\_or\_method with getter, Queue.\_peek(self)

\_put: class cython\_function\_or\_method with getter, PriorityQueue.\_put(self, item)

\_unlock: class cython\_function\_or\_method with getter, Queue.\_unlock(self)

copy: class cython\_function\_or\_method with getter, Queue.copy(self)

empty: class cython\_function\_or\_method with getter, Queue.empty(self) -> bool

full: class cython\_function\_or\_method with getter, Queue.full(self) -> bool

get: class cython\_function\_or\_method with getter, Queue.get(self, block=True, timeout=None)

get\_nowait: class cython\_function\_or\_method with getter, Queue.get\_nowait(self)

hub: class getset\_descriptor with getter, setter, deleter

maxsize: class getset\_descriptor with getter, setter, deleter

peek: class cython\_function\_or\_method with getter, Queue.peek(self, block=True, timeout=None)

peek\_nowait: class cython\_function\_or\_method with getter, Queue.peek\_nowait(self)

put: class cython\_function\_or\_method with getter, Queue.put(self, item, block=True, timeout=None)

put\_nowait: class cython\_function\_or\_method with getter, Queue.put\_nowait(self, item)

qsize: class cython\_function\_or\_method with getter, Queue.qsize(self) -> Py\_ssize\_t

queue: class getset\_descriptor with getter, setter, deleter

function:

next:

#### [asyncio](https://www.rddoc.com/doc/Python/3.6.0/zh/library/asyncio-queue/)中的队列模块

<https://www.rddoc.com/doc/Python/3.6.0/zh/library/asyncio-queue/>

import asyncio

pdir(asyncio.Queue)

descriptor:

maxsize: @property with getter, Number of items allowed in the queue.

function:

\_format:

\_get:

\_init:

\_put:

\_wakeup\_next:

empty: Return True if the queue is empty, False otherwise.

full: Return True if there are maxsize items in the queue.

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue.

join: Block until all items in the queue have been gotten and processed.

put: Put an item into the queue.

put\_nowait: Put an item into the queue without blocking.

qsize: Number of items in the queue.

task\_done: Indicate that a formerly enqueued task is complete.

相对于queue.Queue, 因为是异步io实现的, 所以这里的操作都是非阻塞的了, 并且get\_nowait和put\_nowait中也都没有timeout的参数了.

pdir(asyncio.LifoQueue)

descriptor:

maxsize: @property with getter, Number of items allowed in the queue.

function:

\_format:

\_get:

\_init:

\_put:

\_wakeup\_next:

empty: Return True if the queue is empty, False otherwise.

full: Return True if there are maxsize items in the queue.

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue.

join: Block until all items in the queue have been gotten and processed.

put: Put an item into the queue.

put\_nowait: Put an item into the queue without blocking.

qsize: Number of items in the queue.

task\_done: Indicate that a formerly enqueued task is complete.

pdir(asyncio.PriorityQueue)

descriptor:

maxsize: @property with getter, Number of items allowed in the queue.

function:

\_format:

\_get:

\_init:

\_put:

\_wakeup\_next:

empty: Return True if the queue is empty, False otherwise.

full: Return True if there are maxsize items in the queue.

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue.

join: Block until all items in the queue have been gotten and processed.

put: Put an item into the queue.

put\_nowait: Put an item into the queue without blocking.

qsize: Number of items in the queue.

task\_done: Indicate that a formerly enqueued task is complete.

#### tornado[中的队列模块](http://www.tornadoweb.org/en/stable/queues.html)

<http://www.gevent.org/api/gevent.queue.html>

import tornado.queues

**pdir(tornado.queues)**

property:

\_\_all\_\_, \_\_builtins\_\_, absolute\_import, collections, division, gen, heapq, ioloop, print\_function

class:

Event: An event blocks coroutines until its internal flag is set to True.

Future: Placeholder for an asynchronous result.

LifoQueue: A `.Queue` that retrieves the most recently put items first.

PriorityQueue: A `.Queue` that retrieves entries in priority order, lowest first.

Queue: Coordinate producer and consumer coroutines.

\_QueueIterator:

exception:

QueueEmpty: Raised by `.Queue.get\_nowait` when the queue has no items.

QueueFull: Raised by `.Queue.put\_nowait` when a queue is at its maximum size.

function:

\_set\_timeout:

**pdir(tornado.queues.Queue)**

descriptor:

maxsize: @property with getter, Number of items allowed in the queue.

function:

\_Queue\_\_put\_internal:

\_consume\_expired:

\_format:

\_get:

\_init:

\_put:

empty:

full:

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue without blocking.

join: Block until all items in the queue are processed.

put: Put an item into the queue, perhaps waiting until there is room.

put\_nowait: Put an item into the queue without blocking.

qsize: Number of items in the queue.

task\_done: Indicate that a formerly enqueued task is complete.

put和put\_nowait 与 asyncio.Queue中的put和put\_nowait类似, 都是非阻塞的, 如果队列已满, 会自动进行异步队列的切换, 不会阻塞等待.

**pdir(tornado.queues.LifoQueue)**

descriptor:

maxsize: @property with getter, Number of items allowed in the queue.

function:

\_Queue\_\_put\_internal:

\_consume\_expired:

\_format:

\_get:

\_init:

\_put:

empty:

full:

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue without blocking.

join: Block until all items in the queue are processed.

put: Put an item into the queue, perhaps waiting until there is room.

put\_nowait: Put an item into the queue without blocking.

qsize: Number of items in the queue.

task\_done: Indicate that a formerly enqueued task is complete.

**pdir(tornado.queues.PriorityQueue)**

descriptor:

maxsize: @property with getter, Number of items allowed in the queue.

function:

\_Queue\_\_put\_internal:

\_consume\_expired:

\_format:

\_get:

\_init:

\_put:

empty:

full:

get: Remove and return an item from the queue.

get\_nowait: Remove and return an item from the queue without blocking.

join: Block until all items in the queue are processed.

put: Put an item into the queue, perhaps waiting until there is room.

put\_nowait: Put an item into the queue without blocking.

qsize: Number of items in the queue.

task\_done: Indicate that a formerly enqueued task is complete.

### 2.2 持久化队列

* 现成：
  + queuelib中的diskqueue, 基于磁盘文件的队列

<https://github.com/scrapy/queuelib>

pip install queuelib

import queuelib

**pdir(queuelib)**

property:

\_\_builtins\_\_, pqueue, queue, rrqueue

class:

FifoDiskQueue: Persistent FIFO queue.

LifoDiskQueue: Persistent LIFO queue.

PriorityQueue: A priority queue implemented using multiple internal queues (typically,

RoundRobinQueue: A round robin queue implemented using multiple internal queues (typically,

**pdir(queuelib.queue)**

property:

\_\_builtins\_\_, glob, json, os, sqlite3, struct

class:

FifoDiskQueue: Persistent FIFO queue.

FifoMemoryQueue: In-memory FIFO queue, API compliant with FifoDiskQueue.

FifoSQLiteQueue:

LifoDiskQueue: Persistent LIFO queue.

LifoMemoryQueue: In-memory LIFO queue, API compliant with LifoDiskQueue.

LifoSQLiteQueue:

deque: deque([iterable[, maxlen]]) --> deque object

* + 基于redis实现的queue是一个fifo的队列（如pyspider中的redis\_queue）

<https://github.com/binux/pyspider/blob/master/pyspider/message_queue/redis_queue.py>

from pyspider.message\_queue import redis\_queue

**pdir(redis\_queue)**

property:

BaseQueue, \_\_builtins\_\_, redis, time, umsgpack

class:

Queue: A Queue like message built over redis

RedisQueue: A Queue like message built over redis

**pdir(redis\_queue.RedisQueue)**

property:

max\_timeout

exception:

Empty: Exception raised by Queue.get(block=0)/get\_nowait().

Full: Exception raised by Queue.put(block=0)/put\_nowait().

function:

empty:

full:

get:

get\_nowait:

put:

put\_nowait:

qsize:

因为pyspider中的redis\_queue队列接口名称与之前内存型队列的接口名称相同, 只需要使用一个参数来选择是使用redis型队列还是内存型队列, 就可以使用相同的代码来实现内存型和redis型队列.

pyspider中的redis\_queue是一个先进先出的队列FIFO, 没有实现后进先出的队列LIFO和优先级队列PriorityQueue.

还可以基于其它数据库产品来实现其它类型的队列.

* 利用Python实现基于redis的FIFO、LIFO、Priority队列
  + RedisFifoQueue
  + RedisLifoQueue
  + RedisPriorityQueue

redis的FIFO, LIFO, PQ队列的实现原理

Redis中主要有5种数据结构, Hashes, Lists, Sets, Sorted Sets, Strings, Lists列表型就支持FIFO队列.

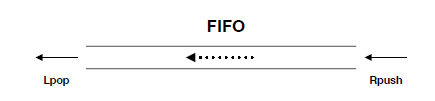


LPUSH key value [value ...]

从队列的左边入队一个或多个元素

RPOP key

从队列的右边出队一个元

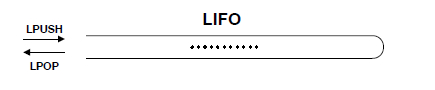


RPUSH key value [value ...]

从队列的右边入队一个元素

LPOP key

从队列的左边出队一个元素



LPUSH key value [value ...]

从队列的左边入队一个或多个元素

LPOP key

从队列的左边出队一个元素

查看pyspider中redis\_queue的源码, 向队列中添加元素是使用的put\_nowait方法来实现的, 在put\_nowait方法中, 使用的是redis中的rpush来完成的. 从队列中取出来元素是使用get\_nowait方法来实现的, 具体代码是使用redis.lpop来完成的.

想要实现后进先出LIFO队列, 只需要把put\_nowait修改为使用redis.lpush, 把get\_nowait修改为使用redis.lpop来实现即可. 或者put\_nowait用redis.rpush, get\_nowait使用redis.rpop来实现.

**def** put\_nowait(self, obj):

**if** self.lazy\_limit **and** self.last\_qsize < self.maxsize:

**pass**

**elif** self.full():

**raise** self.Full

self.last\_qsize = self.redis.rpush(self.name, umsgpack.packb(obj))

**return** True

**def** get\_nowait(self):

ret = self.redis.lpop(self.name)

**if** ret **is** None:

**raise** self.Empty

**return** umsgpack.unpackb(ret)

请求对象的序列化

在put\_nowait方法中, redis.rpush() 中的self.name对应于redis中的key, 而umsgpack.packb(obj)则实现了对obj对象的序列化操作. redis中只能存储二进制类型的数据, 一个请求对象是实例对象, 不能直接保存到redis中, 必须要先进行序列化, 把实例对象转换为二进制的字符串, 才能保存到redis中.

在 get\_nowait 方法中, 取出来的数据又经过umsgpack.unpackb(ret)进行处理, 是反序列化的操作, 把二进制字符串转换为原来的python对象.

umsgpack是一个第3方的模块, 可以使用python中内置的pickle模块来代替umsgpack实现序列化的操作.

与json的使用方法类似, 可以使用pickle.dumps来实现序列化的操作, 而使用pickle.loads实现反序列化的操作.

help(pickle.dumps)

Help on built-in function dumps in module \_pickle:

dumps(obj, protocol=None, \*, fix\_imports=True)

Return the pickled representation of the object as a bytes object.

The optional \*protocol\* argument tells the pickler to use the given

protocol; supported protocols are 0, 1, 2, 3 and 4. The default

protocol is 3; a backward-incompatible protocol designed for Python 3.

Specifying a negative protocol version selects the highest protocol

version supported. The higher the protocol used, the more recent the

version of Python needed to read the pickle produced.

If \*fix\_imports\* is True and \*protocol\* is less than 3, pickle will

try to map the new Python 3 names to the old module names used in

Python 2, so that the pickle data stream is readable with Python 2.

help(pickle.loads)

Help on built-in function loads in module \_pickle:

loads(data, \*, fix\_imports=True, encoding='ASCII', errors='strict')

Read and return an object from the given pickle data.

The protocol version of the pickle is detected automatically, so no

protocol argument is needed. Bytes past the pickled object's

representation are ignored.

Optional keyword arguments are \*fix\_imports\*, \*encoding\* and \*errors\*,

which are used to control compatibility support for pickle stream

generated by Python 2. If \*fix\_imports\* is True, pickle will try to

map the old Python 2 names to the new names used in Python 3. The

\*encoding\* and \*errors\* tell pickle how to decode 8-bit string

instances pickled by Python 2; these default to 'ASCII' and 'strict',

respectively. The \*encoding\* can be 'bytes' to read these 8-bit

string instances as bytes objects.

#### 实现redis队列的基类

把pyspider中的redis\_queue.py中的代码作为基类来使用, 实现先进先出和后进先出的队列.

在项目根目录中创建redis\_queue的目录, 新建base.py, 把pyspider中的redis\_queue.py中的代码复制进来.

修改类名为BaseRedisQueue, 修改umsgpack相应的代码, 使用pickle相应的代码来完成. 删除掉最后面的 Queue = RedisQueue

*#!/usr/bin/env python*

*# -\*- encoding: utf-8 -\*-*

*# vim: set et sw=4 ts=4 sts=4 ff=unix fenc=utf8:*

*# Author: Binux<roy@binux.me>*

*# http://binux.me*

*# Created on 2015-04-27 22:48:04*

*# 修改自pyspider redis\_queue.py*

**import** **time**

**import** **pickle**

**import** **redis**

**from** **six.moves** **import** queue **as** BaseQueue

**class** **BaseRedisQueue**(object):

*"""*

*A Queue like message built over redis*

*使用redis的list类型实现数据存取*

*"""*

Empty = BaseQueue.Empty

Full = BaseQueue.Full

max\_timeout = 0.3

**def** \_\_init\_\_(self, name, host='localhost', port=6379, db=0,

maxsize=0, lazy\_limit=True, password=None, cluster\_nodes=None):

*"""*

*Constructor for RedisQueue*

*maxsize: an integer that sets the upperbound limit on the number of*

*items that can be placed in the queue.*

*lazy\_limit: redis queue is shared via instance, a lazy size limit is used*

*for better performance.*

*"""*

self.name = name

**if**(cluster\_nodes **is** **not** None):

**from** **rediscluster** **import** StrictRedisCluster

self.redis = StrictRedisCluster(startup\_nodes=cluster\_nodes)

**else**:

self.redis = redis.StrictRedis(host=host, port=port, db=db, password=password)

self.maxsize = maxsize

self.lazy\_limit = lazy\_limit

self.last\_qsize = 0

**def** qsize(self):

self.last\_qsize = self.redis.llen(self.name)

**return** self.last\_qsize

**def** empty(self):

**if** self.qsize() == 0:

**return** True

**else**:

**return** False

**def** full(self):

**if** self.maxsize **and** self.qsize() >= self.maxsize:

**return** True

**else**:

**return** False

**def** put\_nowait(self, obj):

**if** self.lazy\_limit **and** self.last\_qsize < self.maxsize:

**pass**

**elif** self.full():

**raise** self.Full

self.last\_qsize = self.redis.rpush(self.name, pickle.dumps(obj))

**return** True

**def** put(self, obj, block=True, timeout=None):

**if** **not** block:

**return** self.put\_nowait(obj)

start\_time = time.time()

**while** True:

**try**:

**return** self.put\_nowait(obj)

**except** self.Full:

**if** timeout:

lasted = time.time() - start\_time

**if** timeout > lasted:

time.sleep(min(self.max\_timeout, timeout - lasted))

**else**:

**raise**

**else**:

time.sleep(self.max\_timeout)

**def** get\_nowait(self):

ret = self.redis.lpop(self.name)

**if** ret **is** None:

**raise** self.Empty

**return** pickle.loads(ret)

**def** get(self, block=True, timeout=None):

**if** **not** block:

**return** self.get\_nowait()

start\_time = time.time()

**while** True:

**try**:

**return** self.get\_nowait()

**except** self.Empty:

**if** timeout:

lasted = time.time() - start\_time

**if** timeout > lasted:

time.sleep(min(self.max\_timeout, timeout - lasted))

**else**:

**raise**

**else**:

time.sleep(self.max\_timeout)

#### 实现redis先进先出队列

新建redis\_queue/fifo\_redis\_queue.py, 实现先进先出redis队列. 因为pyspider中实现的就是先进先出的队列, 只需要导入base.BaseRedisQueue即可.

**from** **.base** **import** BaseRedisQueue

FifoRedisQueue = BaseRedisQueue

#### 实现redis后进先出队列

新建redis\_queue/lifo\_redis\_queue.py, 实现后进先出redis队列. 只需要继承并重写BaseRedisQueue中的get\_nowait或put\_nowait即可.

**import** **pickle**

**from** **.base** **import** BaseRedisQueue

**class** **LifoRedisQueue**(BaseRedisQueue):

**def** get\_nowait(self):

ret = self.redis.rpop(self.name)

**if** ret **is** None:

**raise** self.Empty

**return** pickle.loads(ret)

#### 实现redis优先级队列 RedisPriorityQueue

在保存数据时提供数据的优先级或权重值, 在取数据时就可以取权重值最大或最小的数据. 可以使用redis中的有序集合来实现优先级队列.

连接到redis-cli中进行测试

# 保存数据

ZADD key [NX|XX] [CH] [INCR] score member [score member ...]

添加到有序set的一个或多个成员，或更新的分数，如果它已经存在

zadd pqueue 10 "value10"

zadd pqueue 20 "value20"

zadd pqueue 30 "value30"

zadd pqueue 1 "value1"

# 使用Redis Desktop Manager连接到同一个redis数据库中, 刷新查看数据库中保存的数据. 可以看到数据是按照score也就是设定的权重值从小到大保存的.

# 再添加一个权重值为-1的值

zadd pqueue -1 "value-1"

# 在Redis Desktop Manager中查看

# 注意有序集合中元素的value值不能重复. 如果添加了一个value值相同的元素, 后添加的元素的权重会覆盖掉之前同value元素的权重

zadd pqueue 100 "value100"

zadd pqueue 1000 "value100"

# 取数据

ZRANGE key start stop [WITHSCORES]

根据指定的index返回，返回sorted set的成员列表

# 使用zrange从有序集合中取出来权重在一定区间中的数据, 指定开始和结束的序号, 如1 5 表示取索引值为1到5之间的数据, 包含1和5. 0 0 表示取索引值为0的数据, -1 -1 表示取最后一个数据.

zrange pqueue 1 5

zrange pqueue 0 0

zrange pqueue -1 -1

# 删除取出来的数据.

# 使用list中的lpop取出来数据时会自动把数据删除, 但使用zrange取出来数据后不会删除原数据, 需要手动删除原数据.

ZREM key member [member ...]

从排序的集合中删除一个或多个成员

zrem pqueue (zrange pqueue 0 0)

使用有序集合来实现优先级队列时的注意点为:

1. 使用zadd添加数据, 在添加数据时需要对python对象进行序列化操作

2. 使用zadd添加相同value的数据时会覆盖掉原数据的权重值, 所以有些情况下需要考虑权重的问题.

3. 使用zrange取值, 取值时需要进行反序列化得到python对象

4. 取值后需要使用zrem删除掉取到的数据

redis优先级队列的代码实现

新建redis\_queue/priority\_redis\_queue.py, 实现redis优先级队列. 在FIFO, LIFO队列中, 是使用redis中的列表数据结构实现数据的保存和读取的, 这里是使用redis的有序集合来实现数据的保存和读取的, 只需要把redis\_queue/base.py中相应的列表操作修改为有序集合的操作就可以了. 查找self.redis.

需要修改qsize, put\_nowait, get\_nowait方法.

**import** **pickle**

**import** **threading**

**from** **.base** **import** BaseRedisQueue

**class** **PriorityRedisQueue**(BaseRedisQueue):

*'''利用redis的有序集合来实现数据的存取'''*

**def** qsize(self):

*# zcard获取有序集合中成员的数量*

self.last\_qsize = self.redis.zcard(self.name)

**return** self.last\_qsize

**def** put\_nowait(self, obj):

*'''*

*:param obj: (score, value), obj对象的结果必须是一个元组, 第一个元素为元素的权重*

*:return:*

*'''*

**if** self.lazy\_limit **and** self.last\_qsize < self.maxsize:

**pass**

**elif** self.full():

**raise** self.Full

*# 使用zadd时需要给定一个权重, 查看zadd的源码, 可以使用位置参数或关键字参数*

*# redis.zadd('my-key', 1.1, 'name1', 2.2, 'name2', name3=3.3, name4=4.4)*

self.last\_qsize = self.redis.zadd(self.name, obj[0], pickle.dumps(obj[1]))

**return** True

**def** get\_nowait(self):

*'''*

*-1,-1: 默认取权重值最大的*

*0,0：取权重最小的*

*如果想要用户控制取出来的是最大值或最小值, 还需要修改其它相应的方法.*

*'''*

*# zrange(self, name, start, end, desc=False, withscores=False, score\_cast\_func=float)*

ret = self.redis.zrange(self.name, -1, -1)

*# 测试ret的结果*

*# print("测试zrange的返回值", ret)*

*# 如果有序集合为空, 取出来的值为空, 就raise一个错误提示.*

*# 列表使用 == 来判断, if ret == []:*

**if** **not** ret:

**raise** self.Empty

*# 取出来数据时要使用zrem来删除取出来的数据*

*# 使用zrange查询时返回的是一个包含1个或多个元素的列表, 所以还要使用下标来取出来其中的元素*

*# zrem(self, name, \*values)*

self.redis.zrem(self.name, ret[0])

**return** pickle.loads(ret[0])

在项目根目录中新建 priority\_redis\_queue\_demo.py, 对redis优先级队列进行测试

**from** **redis\_queue.priority\_redis\_queue** **import** PriorityRedisQueue

*# 因为redis是使用容器来实现的, 所以这里的host是容器中的地址*

pqueue = PriorityRedisQueue("pqueue", host="172.17.0.2", db=15)

*# 以元组的结果向pqueue中添加数据*

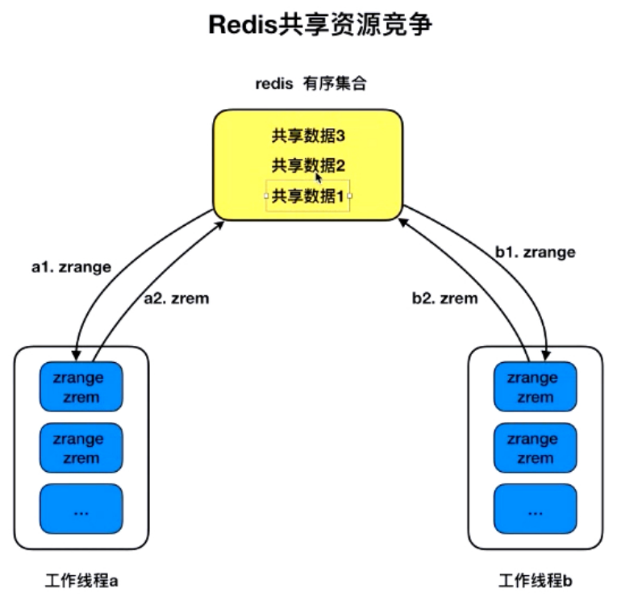
pqueue.put((100,"value100"))

*# 从pqueue中取出来所有的数据, get默认是处于堵塞状态的, 使用block=False取消堵塞状态*

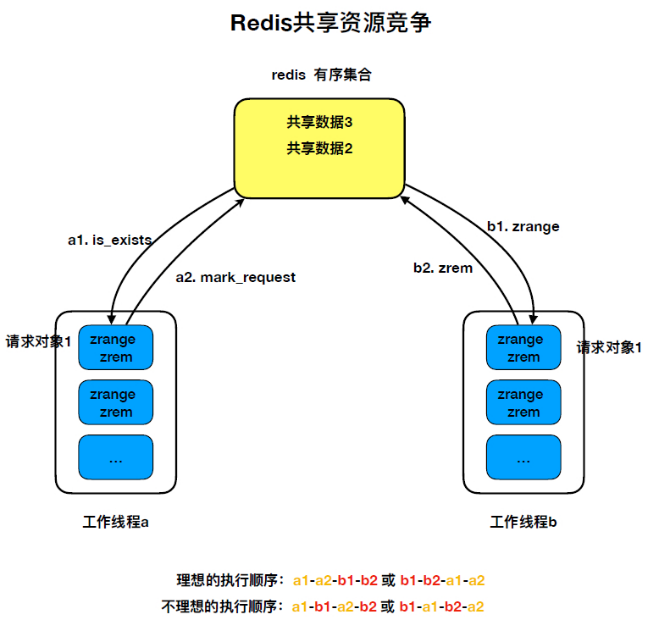
**print**(pqueue.get(block=False))

#### redis共享资源竞争的问题

在FIFO和LIFO队列中, 使用lpop或rpop取出来并删除数据时, 取数据和删除数据是使用一个命令来完成的, 而在priority\_redis\_queue中, 取数据是使用zrange来实现的, 删除数据是使用zrem来实现的, 两个命令之间会存在或多或少的时间间隔, 有可能会出现一些问题.



有两个工作线程a, b, 同时去取redis有序集合中保存的数据, 会以共享数据1, 2, 3 的顺序去取其中的数据,

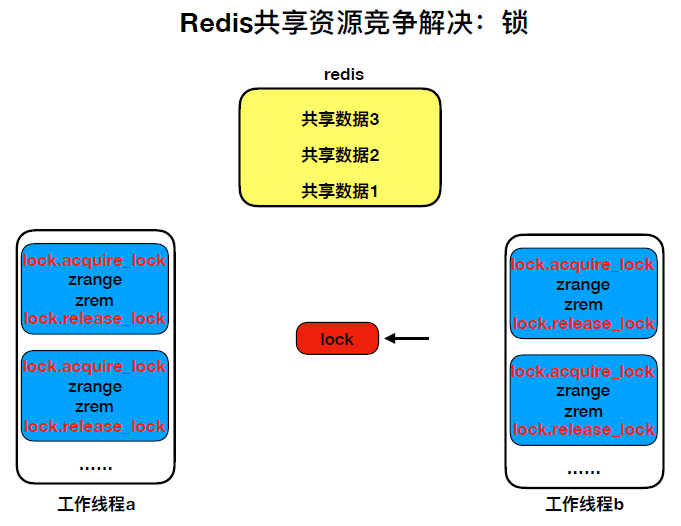


理想的执行顺序: a1-a2-b1-b2 或 b1-b2-a1-a2

不理想的执行顺序: a1-b1-a2-b2或b1-a2-b2-a2

在不理想的执行顺序中, 两个线程同时处理同一份数据, 如果是爬虫, 同一个请求就会发起多次, 就会造成资源的浪费和数据的冲突.

必须要把a1-a2及 b1-b2的操作放在一起, 作为一个原子型的操作. 也就是a1和a2两个操作作为一个原子型的操作, 在这两个操作对某一个数据的操作完成之前, 不能再有其它的线程来对同一个数据进行操作, 或说不能有其它的线程来竞争同一个资源. 可以使用redis的事务操作来实现同一个资源只有一个线程进行操作的需求, 但redis中的事务也并不能百分百的保证一个资源同一时间内只能被一个线程访问到. 可以在用户的代码中使用锁机制来严格实现, 一个线程竞争到某一个资源之后就对它加锁, 在它解锁之前其它的线程都不能对同一个资源进行操作.



说明：

1. 使用了锁机制后，能确保在同一份数据只会被某一个线程获取到，而不会被多个线程同时获取，从而保证了数据不会被处理多次的情况发生

2. 此处相当于实现了同一个线程内部zrange与zrem是一个原子性操作

注意：

1. 一般的内存中的锁只能解决单进程中多个线程间的资源共享问题；

2. 如果是不同进程间甚至不同服务器上线程间资源共享问题，则需要考虑使用如redis分布式锁来实现

3. redis虽有事务机制，但仍不足以保证前面理想的执行结果百分百出现

1. 创建一个每一个线程都能访问的公共锁Lock, 各个线程使用的是同一把锁.

2. 锁有2个方法, acquire\_lock和release\_lock. 对资源操作前先上锁, 操作完再解锁, 保证同一时间只能一个线程对资源进行操作

3. 加锁之后就会牺牲掉多线程多任务的特性, 但如果加锁和解锁之间没有很多耗时的操作, 对性能的影响也可以很小. 所以加锁和解锁之间尽量不要有耗时的操作.

4. 当多个线程在同一个进程中时, 可以实现锁的共享, 但如果是不同进程之间的线程, 或不同宿主机或不同容器中的线程, 就不能实现锁的共享了, 可以使用redis分布式锁来实现不同进线中的线程使用同一把锁的目的.

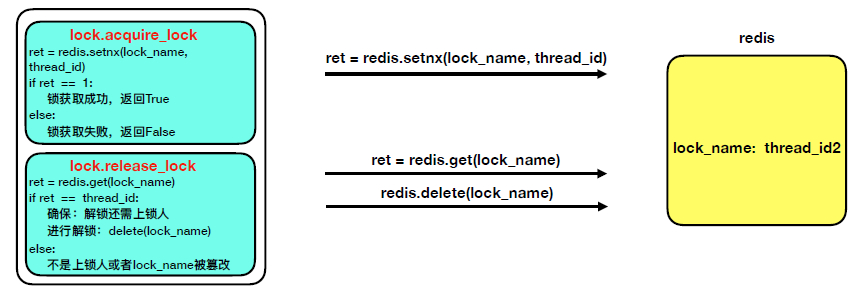
#### Redis分布式锁——实现原理

锁的状态：上锁状态(等着被解锁)、解锁状态(等着被上锁)

锁的动作：解锁、上锁

锁的特征：解锁还需上锁人

死锁的处理



创建一个锁对象lock, 创建锁的上锁和解锁方法lock.acquire\_lock, lock.release\_lock.

上锁时, 只能对处理解锁状态的锁上锁. 先判断, 如果是处于锁定的状态, 就上锁, 并返回True, 表示上锁成功. 如果判断锁是处于上锁的状态, 会上锁失败, 就返回False, 表示上锁失败.

解锁时, 也需要先对锁的状态进行判断, 只能对上锁状态的锁进行解锁. 并且解锁时需要判断解锁和上锁的是不是同一个线程, 只能对自己上锁的锁进行解锁.

使用redis来实现分布式锁.

首先要使用redis来记录锁的状态.

SET key value [EX seconds] [PX milliseconds] [NX|XX]

给一个key设置value值, 如果key值已经存在, 就会覆盖掉原来的value值.

SETNX key value

给一个key设置value值，只有当该键不存在时才能设置成功, 当key已经存在时会设置失败并返回0.

上锁过程: 给setnx设置key-value值

ret = redis.setnx(lock\_name, thread\_id)

if ret == 1:

锁获取成功，返回True

else:

锁获取失败，返回False

解锁过程: 获取setnx中对应key的value值. 并删除key

ret = redis.get(lock\_name)

if ret == thread\_id:

确保：解锁还需上锁人

进行解锁：delete(lock\_name)

else:

不是上锁人或者lock\_name被篡改

**如何实现解锁还需上锁人**

每个线程都有一个thread\_id, 并且是唯一的. 在代码中手动给线程设置一个唯一的id值, 在上锁时把thread\_id作为value值添加到setnx中, 在解锁时判断thread\_id是否与锁的value值相等, 只有相等时才表示解锁和上锁的是同一个线程, 才能执行解锁的操作.

**死锁的产生和处理**

产生:

1. 上锁的线程挂掉了.

2. 强行修改了锁的thread\_id值, 此时锁会一直无法打开

死锁的解决:

1. 强行删除了lock\_name的key, 相当于手动删除了锁.

2. 利用redis的过期时间, 如代码的运行过程不会超过1min, 就给锁设置一个1min的过程时间, 如果到了1min锁依旧没有被解开, 上锁的线程就可能出现了问题, 此时就自动删除这把锁.

说明：

1. 获取锁：利用redis的setnx命令特征

1.1 如果key不存在则执行操作，返回值将是1，此时表明锁获取成功，即上锁；

1.2 如果key存在则不执行任何操作，返回值将是0，此时表明锁获取失败，因为已经被上锁了

2. 释放锁：获取设置的值，判断是否是当前线程设置的值

2.1 get命令获取对应key的值

2.2 判断值是否和预先设置的一样(thread\_id)，保证不是其他线程解开的锁

2.3 如果一致，就把该key删除，表示释放锁，此时其他线程便可以获取到锁

注意：

1. 同一把锁，注意lock\_name一致

2. 使用同一把锁的各个线程，必须维护好各自的thread\_id，不能重复。否则可能出现，如a线程上的锁却被b线程解开了这样的bug

3. 为防止死锁问题（如a线程上了锁，但在解开锁前a线程挂了），应当给lock\_name这个数据设置一定过期时间，具体时间，依实际情况定.

#### redis分布式锁的代码实现

在项目根目录中新建 redis\_lock.py

**import** **pickle**

**import** **time**

**import** **redis**

**class** **RedisLock**(object):

**def** \_\_init\_\_(self, lock\_name, host="localhost", port=6379, db=0):

*'''*

*创建redis客户端*

*'''*

self.redis = redis.StrictRedis(host=host, port=port, db=db)

self.lock\_name = lock\_name

**def** \_get\_thread\_id(self):

*'''*

*生成每个线程唯一的thread\_id值*

*'''*

**import** **threading**

**import** **socket**

**import** **os**

*# thread\_id = "服务器号" + "进程号" + threading.current\_thread().name*

*# socket.gethostname()获取的是主机号, 对于未命名的容器, 获取的是容器的id*

*# os.getpid(), 在容器中进程号是从1开始的.*

*# return socket.gethostname() + str(os.getpid())+ threading.current\_thread().name*

**return** "{}\_{}\_{}".format(socket.gethostname(), str(os.getpid()), threading.current\_thread().name)

**def** acquire\_lock(self, thread\_id=None, expire=10, block=True):

*'''*

*上锁*

*:param thread\_id: 表明每个线程的唯一标识值，用来判断解锁*

*:param block: 在上锁时指定这个锁是否是阻塞状态的, 默认是阻塞的状态.*

*# 如果block=True, 在上锁时, 如果锁已经处于上锁的状态, 就一直循环等待.*

*# 如果block=False, 在上锁时, 如果锁已经处于上锁的状态, 不进行等待, 直接上锁失败.*

*'''*

*# 如果在上锁的时候用户没有手动指定thread\_id的值, 就调用\_get\_thread\_id方法自动生成.*

**if** thread\_id **is** None:

thread\_id = self.\_get\_thread\_id()

*# 如果在上锁时设定锁是阻塞状态的, 就一直执行循环中的内容, 一直尝试上锁的操作, 只有当上锁成功时才跳出循环.*

**while** block:

*# 如果lock\_name存在，ret=0，否则ret=1*

ret = self.redis.setnx(self.lock\_name, pickle.dumps(thread\_id))

*# 如果ret==1, 表示上锁成功, 上锁成功, 就跳出循环, 不再执行.*

**if** ret == 1:

*# 为了解决死锁的问题, 在上锁成功后给锁添加一个过期时间*

self.redis.expire(self.lock\_name, expire)

**print**("上锁成功: **%s**" % self.lock\_name)

**return** True

*# 如果上锁不成功, 就一直执行循环中的语句.*

time.sleep(0.01)

*# 如果lock\_name存在，ret=0，否则ret=1, 注意要先把文本进行序列化, 换为二进制后作为value保存到redis中.*

ret = self.redis.setnx(self.lock\_name, pickle.dumps(thread\_id))

**if** ret == 1:

self.redis.expire(self.lock\_name, expire)

**print**("上锁成功: **%s**" % self.lock\_name)

**return** True

**else**:

**print**("上锁失败: **%s**" % self.lock\_name)

**return** False

**def** release\_lock(self, thread\_id=None):

*'''*

*解锁*

*'''*

**if** thread\_id **is** None:

thread\_id = self.\_get\_thread\_id()

ret = self.redis.get(self.lock\_name)

*# pickle.loads(ret) == thread\_id的作用是确保 解锁线程还是上锁的线程, 因为thread\_id在保存的时候是以二进制形式保存的, 所以要先进行反序列化, 转换成文本, 再进行判断*

*# ret is not None, 如果在上锁前就执行解锁的操作, 或者对一个不存在的锁进行解锁时, ret的值为None, 就会报错.*

**if** ret **is** **not** None **and** pickle.loads(ret) == thread\_id:

self.redis.delete(self.lock\_name)

**print**("解锁成功: **%s**" % self.lock\_name)

**return** True

**else**:

**print**("解锁失败: **%s**" % self.lock\_name)

**return** False

**if** \_\_name\_\_ == '\_\_main\_\_':

**import** **threading**

redis\_lock = RedisLock("redis\_lock2", host="172.17.0.2")

*# 在没有上锁之前强行的进行解锁.*

*# redis\_lock.release\_lock(thread\_id)*

*# 使用过期时间, 解决死锁*

**if** redis\_lock.acquire\_lock(expire=10):

**print**("执行对应的操作")

redis\_lock.release\_lock()

*# 如果设置block=True时, 在上锁时, 如果锁已经处于上锁的状态, 就一直循环等待.*

*# 如果设置block=False, 在上锁时, 如果锁已经处于上锁的状态, 不进行等待, 直接上锁失败.*

*# 设置block=True, 第一次执行代码后接着再次执行代码, 程序就会处于等待状态, 直到10s后死锁自动删除, 才上锁成功.*

**if** redis\_lock.acquire\_lock(expire=10, block=True):

**print**("执行对应的操作")

redis\_lock.release\_lock()

#### 使用redis分布式锁解决redis优先级队列中资源竞争的问题

修改redis\_queue/priority\_redis\_queue.py中的get\_nowait中的代码, 使用redis分布式锁保证一个线程在执行redis.zrange和redis.zrem这两个操作时其它线程都无法操作. 即要使用redis分布式锁把redis.zrange和redis.zrem之间的代码锁起来. 要想在redis\_queue/priority\_redis\_queue.py中使用redis\_lock.py中的内容, 首先要导入, 为了使用的方法, 把redis\_lock.py移动到redis\_queue目录中, 再把redis\_queue目录修改为redis\_tools,

在redis\_tools/priority\_redis\_queue.py中需要导入RedisLock, 在get\_nowait()方法中实例化时需要填写\_\_init\_\_(self, lock\_name, host="localhost", port=6379, db=0)中的参数, 参数可以在get\_nowait()中传递进来, 由于get\_nowait是从BaseRedisQueue类中继承而来的, 在BaseRedisQueue中, get方法调用了get\_nowait方法, 要修改priority\_redis\_queue.py中的get\_nowait(), 就要同时也重写get方法, 为了不方法, 可以在BaseRedisQueue的初始化方法中定义一个名为redis\_lock\_config的字典型参数, PriorityRedisQueue继承了BaseRedisQueue, 初始化PriorityRedisQueue时就会把RedisLock的参数redis\_lock\_config传递过来, get\_nowait中的RedisLock在实例化时就能接收和使用这些参数了.

修改redis\_tools/base.py, 在\_\_init\_\_方法中添加RedisLock的参数. 添加是否使用锁的开关, 默认情况下use\_lock=False即不使用锁, 用户传入use\_lock=True来使用锁. 同时设定初始时的锁为None.

**class** **BaseRedisQueue**(object):

*"""*

*A Queue like message built over redis*

*使用redis的list类型实现数据存取*

*"""*

Empty = BaseQueue.Empty

Full = BaseQueue.Full

max\_timeout = 0.3

**def** \_\_init\_\_(self, name, host='localhost', port=6379, db=0,

maxsize=0, lazy\_limit=True, password=None, cluster\_nodes=None,

redis\_lock\_config={}, use\_lock=False):

*"""*

*Constructor for RedisQueue*

*maxsize: an integer that sets the upperbound limit on the number of*

*items that can be placed in the queue.*

*lazy\_limit: redis queue is shared via instance, a lazy size limit is used*

*for better performance.*

*"""*

self.name = name

**if**(cluster\_nodes **is** **not** None):

**from** **rediscluster** **import** StrictRedisCluster

self.redis = StrictRedisCluster(startup\_nodes=cluster\_nodes)

**else**:

self.redis = redis.StrictRedis(host=host, port=port, db=db, password=password)

self.maxsize = maxsize

self.lazy\_limit = lazy\_limit

self.last\_qsize = 0

self.redis\_lock\_config = redis\_lock\_config

self.lock = None

self.use\_lock = use\_lock

修改redis\_tools/priority\_redis\_queue.py, 添加使用分布式锁的代码

**import** **pickle**

**import** **threading**

**from** **.base** **import** BaseRedisQueue

**class** **PriorityRedisQueue**(BaseRedisQueue):

*'''利用redis的有序集合来实现数据的存取'''*

**def** qsize(self):

*# zcard获取有序集合中成员的数量*

self.last\_qsize = self.redis.zcard(self.name)

**return** self.last\_qsize

**def** put\_nowait(self, obj):

*'''*

*:param obj: (score, value), obj对象的结果必须是一个元组, 第一个元素为元素的权重*

*:return:*

*'''*

**if** self.lazy\_limit **and** self.last\_qsize < self.maxsize:

**pass**

**elif** self.full():

**raise** self.Full

*# 使用zadd时需要给定一个权重, 查看zadd的源码, 可以使用位置参数或关键字参数*

*# redis.zadd('my-key', 1.1, 'name1', 2.2, 'name2', name3=3.3, name4=4.4)*

self.last\_qsize = self.redis.zadd(self.name, obj[0], pickle.dumps(obj[1]))

**return** True

**def** get\_nowait(self):

*'''*

*-1,-1: 默认取权重值最大的*

*0,0：取权重最小的*

*如果想要用户控制取出来的是最大值或最小值, 还需要修改其它相应的方法.*

*'''*

*# zrange(self, name, start, end, desc=False, withscores=False, score\_cast\_func=float)*

ret = self.redis.zrange(self.name, -1, -1)

*# 测试ret的结果*

*# print("测试zrange的返回值", ret)*

*# 如果有序集合为空, 取出来的值为空, 就raise一个错误提示.*

*# 列表使用 == 来判断, if ret == []:*

**if** **not** ret:

**raise** self.Empty

*# 取出来数据时要使用zrem来删除取出来的数据*

*# 使用zrange查询时返回的是一个包含1个或多个元素的列表, 所以还要使用下标来取出来其中的元素*

*# zrem(self, name, \*values)*

self.redis.zrem(self.name, ret[0])

**return** pickle.loads(ret[0])

**def** get\_nowait(self):

*'''*

*-1,-1: 默认取权重值最大的*

*0,0：取权重最小的*

*'''*

*# 判断用户是否修改了use\_lock的值, 如果use\_lock为True, 表示使用锁.*

**if** self.use\_lock **is** True:

**from** **.redis\_lock** **import** RedisLock

*# 所有的线程使用的是同一把锁, 锁只用创建一次, 只有当lock不存在时才会去创建锁. 在初始化时设定锁为None, 在创建锁时进行判断, 如果之前没有创建过锁, 才进行创建. 在第2次调用get\_nowait时, 就不会再执行创建锁的操作了. 这样就保证了使用的是同一把锁.*

**if** self.lock **is** None:

*# 由于redis\_lock\_config的参数是字典的格式, 所以要使用两个\*\*来进行接收和解包*

self.lock = RedisLock(\*\*self.redis\_lock\_config)

**if** self.lock.acquire\_lock():

ret = self.redis.zrange(self.name, -1, -1)

**if** **not** ret:

**raise** self.Empty

self.redis.zrem(self.name, ret[0])

self.lock.release\_lock()

**return** pickle.loads(ret[0])

*# 如果use\_lock的值为Fasle, 表示不使用锁*

**else**:

*# zrange(self, name, start, end, desc=False, withscores=False, score\_cast\_func=float)*

ret = self.redis.zrange(self.name, -1, -1)

**if** **not** ret:

**raise** self.Empty

*# 取出来数据时要使用zrem来删除取出来的数据*

*# 使用zrange查询时返回的是一个包含1个或多个元素的容器, 所以还要使用下标来取出来其中的元素*

self.redis.zrem(self.name, ret[0])

**return** pickle.loads(ret[0])

修改项目根目录中的 priority\_redis\_queue\_demo.py, 对使用分布式锁的redis优先级队列进行测试

**from** **redis\_tools.priority\_redis\_queue** **import** PriorityRedisQueue

*# 因为redis是使用容器来实现的, 所以这里的host是容器中的地址*

pqueue = PriorityRedisQueue(

"pqueue",

host="172.17.0.2",

db=15,

use\_lock=True,

redis\_lock\_config={"lock\_name":"pqueue\_lock", "host":"172.17.0.2"}

)

*# 以元组的结果向pqueue中添加数据*

pqueue.put((100,"value100"))

**print**(pqueue.get())

运行文件进行测试, 把use\_lock修改为False, 再次运行, 查看结果的不同.

由于redis分布式锁redis\_lock.py, 基于redis实现的3种优先级队列fifo\_redis\_queue.py, lifo\_redis\_queue.py, priority\_redis\_queue.py都作为redis\_tools包中的模块了, 为方便外部的调用, 为4个模块提供一个统一的接口.

修改redis\_tools/\_\_init\_\_.py

**def** get\_redis\_queue\_cls(queue\_type):

**if** queue\_type == "fifo":

**from** **.fifo\_redis\_queue** **import** FifoRedisQueue

**return** FifoRedisQueue

**elif** queue\_type == "lifo":

**from** **.lifo\_redis\_queue** **import** LifoRedisQueue

**return** LifoRedisQueue

**elif** queue\_type == "priority":

**from** **.priority\_redis\_queue** **import** PriorityRedisQueue

**return** PriorityRedisQueue

**else**:

**raise** **Exception**("只支持fifo、lifo、priority三种队列类型")

**def** get\_redis\_lock\_cls():

**from** **.redis\_lock** **import** RedisLock

**return** RedisLock

在使用时, 只需要修改get\_redis\_queue\_cls()中传递的参数, 修改redis队列的配置就可以了.

**from** **redis\_tools** **import** get\_redis\_queue\_cls

*# 在使用某种redis队列时, 只需要传入对应的参数, 就能选择相应的redis队列了.*

*# 只支持fifo、lifo、priority三种队列类型*

RedisQueue = get\_redis\_queue\_cls("priority")

*# redis队列的参数配置单独提取出来*

REDIS\_QUEUE\_CONFIG = {

"name": "pqueue",

"host": "172.17.0.2",

"db": 0,

"use\_lock": False,

"redis\_lock\_config": {

"lock\_name": "pqueue\_lock",

"host": "172.17.0.2"

}

}

*# 为了使用的统一, RedisQueue*

pqueue = RedisQueue(\*\*REDIS\_QUEUE\_CONFIG)

*# 以元组的结果向pqueue中添加数据*

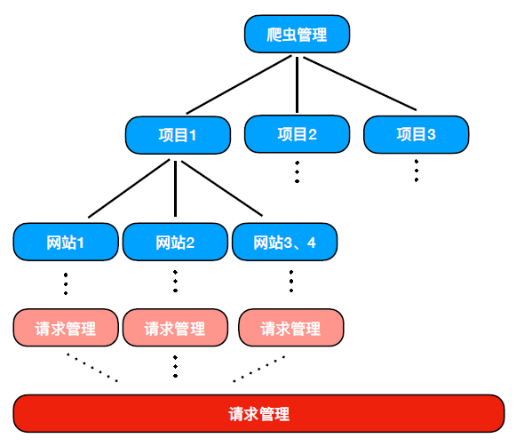
pqueue.put((100,"value100"))

**print**(pqueue.get())

#### 请求缓冲与请求调度管理

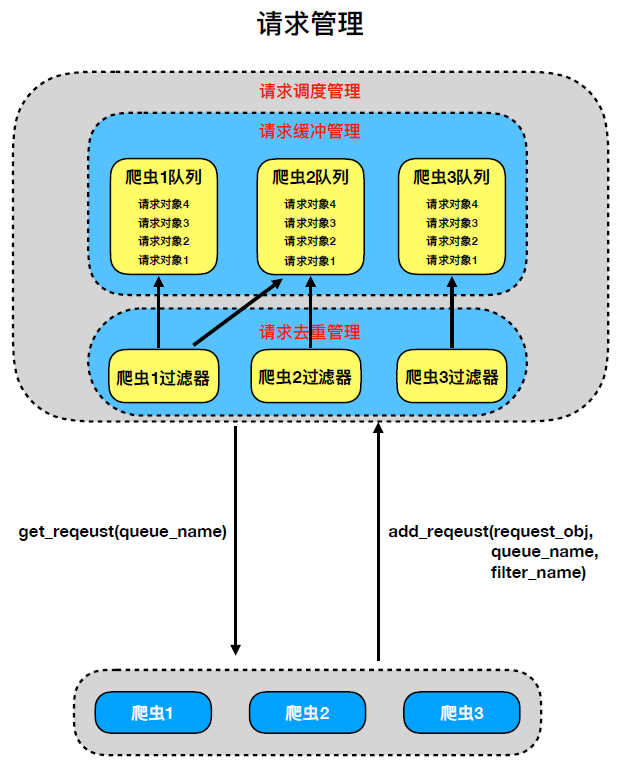
现在就完成了请求缓冲与调度管理中的3种队列, 接下来就要实现完整的请求去重, 缓冲, 调度管理, 实现一个完整的请求管理业务功能.

最重要的不是代码如何去写, 而是如何用代码来实现已经设计好的业务逻辑思路和结构, 思路和逻辑结构是多样的. 代码是思路的实现, 不同的思路可以实现相同的功能.



如图所示, 一个需求对应一个项目, 一个项目会对应多个网站, 每个网站的具体情况不同, 如反爬措施严格与否, 数据量大与否, 即每一个网站都会对应一个不同的请求调度的策略, 也就意味着每个项目中不同网站的爬虫要解耦分开来编写, 如果有多个项目, 多个网站, 到最后就需要管理大量网站的大量的爬虫, 为了方便后期维护, 在开始项目设计和代码编写阶段就考虑把多个项目放在一起进行管理. 不同项目, 不同网站的爬虫, 代码上是分开的, 但运行和管理则是统一进行的.

每个网站都会对应着自己的请求队列, 请求去重过滤器, 对应着自己的请求管理, 就需要把多个网站的请求管理进行统一的管理. 以下的请求管理设计图就是为了实现同时统一管理多个项目和多个网站多个爬虫的请求管理而设计的.



每个爬虫的请求对应自己的队列和过滤器, 在对某个爬虫进行维护时, 只需要修改其对应的过滤器, 队列和爬虫规则即可, 不同爬虫的功能之间相互独立, 彼此之间不受影响.

对所有的队列和过滤器进行统一的管理, 对外提供一个统一的接口来添加请求和获取请求. 在添加某个爬虫的请求时, 如爬虫代码中从页面中获取到了新的请求, 需要把新的请求添加到队列中, 此时不仅要提供请求对象, 还要提供队列的名称, 指定请求要添加到哪个队列中, 也要提供过滤器的名称, 指定使用哪个过滤器来进行过滤.

一个过滤器可以供给多个爬虫队列来使用, 例如想要爬虫1, 爬虫使用相同的过滤器, 只需要提供同名的过滤器就可以了.

在从队列中获取请求时, 如在爬虫代码中获取请求, 对请求进行处理时, 需要指定队列的名称. 就可以根据队列的优先级从队列中取出来请求了.

最重要的不是代码, 而是学习如何使用代码来实现思路, 如何从设计图一步步的使用代码来实现功能.

1. 从整体上来看, 整个请求管理是一个对象, 提供两个方法, add\_request, get\_request, 定义一个类, 设计两个方法.

2. 从内部具体的功能来看, 需要实现请求过滤, 请求队列, 请求调度三个功能.

请求过滤器使用request\_manager/request\_filter来实现, 提供一个请求对象, 通过is\_exists和mark\_request来判断请求是否存在并对请求进行标记. 需要对多个队列维护多个过滤器, 所以在类中还需要使用一个容器来保存多个过滤器,

请求缓冲管理可以使用队列来完成, 可以使用基于内存, 磁盘和数据库的队列实现. 这里要把请求管理设计成一个支持分布式管理的请求对象, 要实现分布式的管理, 这里就要选择基于redis数据库的队列. 把redis\_tools目录移动到request\_manager/utils目录下, 使其成为request\_manager请求管理中的一个功能模块. 同样, 队列也有多个, 需要使用一定的容器来存储和管理多个队列.

综上分析, 代码的整体结构如下, 一个请求管理类, 其中实现get\_reqeust和add\_reqeust两个方法. 其中创建2个基本的容器, 如字典, 来保存多个过滤器和多个队列. 在添加请求add\_request方法中就可以传入队列名称和过滤器名称来区分不同的请求对象.

因为请求管理类是在request\_manager中实现的, 所以可以在request\_manager/\_\_init\_\_.py中来实现.

## 3. 消息队列与Python

### 3.1 消息队列介绍

* 什么是消息队列？
* 消息队列和一般的队列有什么区别和联系？

消息队列是基于生产者-消费者模型和队列的特征实现的数据传递的方法. 常见的消息队列的形式有kafka和rabbitmq

消息队列对数据进行了预处理, 添加了如资源竞争的处理, 可以直接使用.

消息队列是以服务的形式进行使用的, 使用客户端来使用消息队列. 客户端就是生产者消费者模式的客户端, 生产都产生消息, 消费者从消息队列中获取数据. 想要实现多路消息队列, 只需要进行简单的配置即可.

### 3.2 消息队列之Kafka工作原理 -- broker

生产者和消费都是客户端, 在kafka中, 服务端称为broker, broker通过9092端口对外提供服务. broker会把数据保存到磁盘中实现持久化的存储, 能够存储海量的数据.

可以启动多个broker, 作为一个集群统一对外进行服务, 从而提高kafka的性能.



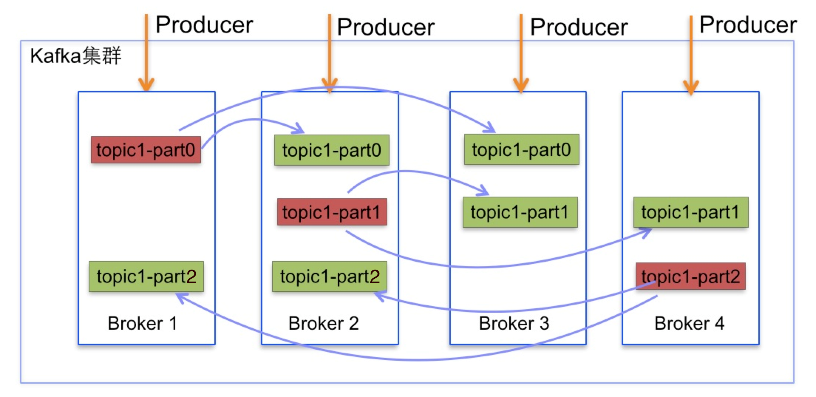
### 3.3 消息队列之Kafka工作原理 -- topic



在请求管理中, 使用add\_request方法向队列中添加请求时, 需要指定队列名称的参数, 指定向哪个队列中添加请求, 在使用get\_request方法从队列中获取请求时, 也要指定队列名称, 指定从哪个队列中获取请求. 在kafka中, 生产者产生的数据在保存到kafka中时必须要指定一个topic, 消费者在消费数据时也要指定是从哪个topic中取出来数据. 所以kafka中的消息是按照topic进行区分的, 一个topic可以看成是一条队列.

kafka 可以以集群cluster的方式提示服务, 每个集群cluster中都有多个broker

### 3.4 消息队列之Kafka工作原理 – partition



分区partition的概念.

kafka中不同生产者产生的数据是保存在不同的topic中的, 所以topic是kafka中数据保存的基本单元. 而在每一个topic中, 数据又是分区partition进行存储的.

每个kafka集群作为一个整体对外提供消息服务, 每个kafka中都可以包含多个broker, 而作为客户端的生产者产生数据, 数据是通过kafka集群中的多个broker进行保存的, 多个broker在保存数据时是按照partition分区进行存储的. 如在保存topic1的数据时, 用3个broker保存在3个分区中. 如果有100条数据, 这100条数据会大致均匀的保存在这3个分区中.

在启动kafka集群时, 还需要指定创建的副本的数量, 每个topic每个分区的数据都会在其它broker的分区中创建副本, 副本和备份是以topic的分区为单位的, 副本不会保存到与原始数据相同的broker中, 创建的总的副本数量会小于broker的数量.

### 3.5 消息队列之Kafka安装介绍

* 版本
  + Apache Kafka 与 Confluent Platform
  + Apache Kafka <https://kafka.apache.org/>
  + Confluent Platform <https://www.confluent.io/>
* Docker镜像
  + Confluent kafka 的docker镜像
  + <https://github.com/confluentinc/cp-docker-images>
  + 使用说明:

<https://docs.confluent.io/current/installation/docker/docs/index.html>

* + examples: examples/kafka-single-node/ docker-compose.yml

---

version: '2'

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

environment:

ZOOKEEPER\_CLIENT\_PORT: 2181

ZOOKEEPER\_TICK\_TIME: 2000

kafka:

# "`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-

# An important note about accessing Kafka from clients on other machines:

# -----------------------------------------------------------------------

#

# The config used here exposes port 9092 for \_external\_ connections to the broker

# i.e. those from \_outside\_ the docker network. This could be from the host machine

# running docker, or maybe further afield if you've got a more complicated setup.

# If the latter is true, you will need to change the value 'localhost' in

# KAFKA\_ADVERTISED\_LISTENERS to one that is resolvable to the docker host from those

# remote clients

#

# For connections \_internal\_ to the docker network, such as from other services

# and components, use kafka:29092.

#

# See https://rmoff.net/2018/08/02/kafka-listeners-explained/ for details

# "`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-'"`-.\_,-

#

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

ports:

- 9092:9092

environment:

KAFKA\_BROKER\_ID: 1

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT\_HOST://localhost:9092

KAFKA\_LISTENER\_SECURITY\_PROTOCOL\_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT\_HOST:PLAINTEXT

KAFKA\_INTER\_BROKER\_LISTENER\_NAME: PLAINTEXT

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

kafka的集群要依赖于zookeeper集群管理器, 所以要先启动zookeeper服务.

* 客户端工具
  + Apache Kafka的Python客户端：kafka-python. 不太稳定, 可能会丢包.
  + Confluent kafka的Python客户端： confluent-kafka-python, 比较稳定.
  + git地址

<https://github.com/confluentinc/confluent-kafka-python>

在安装confluent-kafaka-python时需要先安装librdkafka, librdkafka是基于c/c++编译的, 所以先要安装gcc, 再安装librdkafka, 然后使用pip install conflent-kafaka-python

<https://github.com/edenhill/librdkafka>

* + 使用文档

<https://docs.confluent.io/current/clients/confluent-kafka-python/index.html>

也可以参考github中给出的实例.

在项目根目录中新建 kafka\_demo/docker-compose.yml

version: '2'

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

*# 指定网络模式为host, 即直接使用宿主机的网络.*

network\_mode: host

*# zookeeper和kafka启动时内部都有一些配置项, 使用enviroment来更改启动时的配置项.*

environment:

*# zookeeper提供服务的端口*

ZOOKEEPER\_CLIENT\_PORT: 32181

ZOOKEEPER\_TICK\_TIME: 2000

extra\_hosts:

- "moby:127.0.0.1"

kafka:

image: confluentinc/cp-kafka:latest

*# 如果kafka的网络模式也指定为host, 服务是运行在宿主机上的, kafka的地址与宿主机的地址完全一样, 此时. KAFKA\_ZOOKEEPER\_CONNECT中的地址可以使用localhost, 表示与宿主机中的一样, 如果使用的是其它的网络模式, 如bridge, KAFKA\_ZOOKEEPER\_CONNECT中的地址就要使用zookeeper.*

network\_mode: host

depends\_on:

- zookeeper

*# 对外部提供的服务是暴露在29092端口的, 对内提供的服务时使用的是9092端口. 在host网络模式下, 可以不开放端口.*

ports:

- 29092:29092

*# 修改配置文件中对应项的值, 百度一下kafka配置项, 前面加上KAFKA\_的前缀, 点号变为下划线, 全部大写.*

environment:

KAFKA\_BROKER\_ID: 1

*# host模式下, 这里就要使用localhost或者是主机地址, 不能使用域名zookeeper*

*# kafka中zookeeper中的端口要与上面zookeeper中的配置一致.*

KAFKA\_ZOOKEEPER\_CONNECT: localhost:32181

*# 这里的地址是客户端生产者和消费者连接kafka时的地址,*

*# host模式下，地址可以写localhost, 此时只能在本地访问kafka服务，但如果想要在host模式下提供外部访问，那么必须使用宿主机的地址, 并且开放相应的端口*

*# 在bridge模式下, kafka, zookeeper和宿主机的网络已经隔离开, 通过ip地址和端口访问, 此时必须要对外开放这里的端口并进行端口映射, 作为客户端的生产者和消费者才能连接服务..*

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://10.211.55.3:29092

*# 每个分区数据保存的总的份数. 1时无副本, 2时副本数为1. 注意指定的数据保存的总份数要小于等于启动的kafka服务的数量*

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

extra\_hosts:

- "moby:127.0.0.1"

yml文件会自动上传到远程的linux-server中, 如果没有自动上传, 执行手动上传. 在linux中执行构建容器的操作.

docker-compose -f docker-compose.yml up -d

# 查看运行的容器, 会运行cp-kafka, cp-zookeeper两个容器

docker ps -a

kafka默认使用9092端口进行通信, 如果有防火墙, 必须要打开9092端口.

# 查看开放的端口

firewall-cmd --list-ports

# 开放9092端口

firewall-cmd --zone=public --add-port=9092/tcp --permanent

# 开放29092端口

firewall-cmd --zone=public --add-port=29092/tcp --permanent

# 重启防火墙

service firewalld restart

# 结束docker-compose的运行, 并重新生成容器

docker-compose -f docker-compose.yml down

docker-compose -f docker-compose.yml up -d

安装confluent-kafka-python

因为python开发环境是基于docker的, 所以使用Dockerfile来构建. 修改python-dev/Dockerfile, 添加安装kafka的代码.

# 配置基本的python开发环境

# 注意：如果pip安装出现如certificate verify failed的错误，pip后加上 --trusted-host files.pythonhosted.org 试试

FROM python:alpine3.6

RUN apk add --no-cache gcc musl-dev

RUN apk add --no-cache libffi-dev && pip install ipython

# filter-class依赖环境

RUN apk add --no-cache openssl-dev &&\

pip install redis &&\

pip install pymysql &&\

pip install sqlalchemy &&\

pip install six &&\

pip install w3lib &&\

pip install simhash &&\

pip install requests

# 编译安装 confluent-kafka-python

RUN apk add --no-cache libc-dev pcre-dev zlib-dev jemalloc-dev &&\

apk --update add git openssh && \

apk --update --no-cache add bash &&\

apk add --update --no-cache g++ &&\

apk add --update --no-cache make &&\

rm -rf /var/lib/apt/lists/\* && \

rm /var/cache/apk/\*

RUN git clone https://github.com/edenhill/librdkafka.git &&\

cd librdkafka &&\

./configure &&\

make &&\

make install &&\

pip install confluent-kafka

使用confluent-kafka-python

新建kafka\_demo/kafka\_producer.py, kafka\_demo/kafka\_consumer.py, 分别把示例代码复制进去.

kafka\_producer.py

**from** **confluent\_kafka** **import** Producer

*# 运行kafka服务的地址和端口*

*# p = Producer({'bootstrap.servers': 'mybroker1,mybroker2'})*

p = Producer({'bootstrap.servers': '10.211.55.3:29092'})

*# 处理错误与处理的结果*

**def** delivery\_report(err, msg):

*""" Called once for each message produced to indicate delivery result.*

*Triggered by poll() or flush(). """*

**if** err **is** **not** None:

**print**('Message delivery failed: {}'.format(err))

**else**:

**print**('Message delivered to {} [{}]: {}'.format(msg.topic(), msg.partition(), msg.value()))

*# 生产者产生一系列的数据*

*# for data in some\_data\_source:*

**for** data **in** ["eeee**%d**"%i **for** i **in** range(30)]:

*# Trigger any available delivery report callbacks from previous produce() calls*

p.poll(0)

*# Asynchronously produce a message, the delivery report callback*

*# will be triggered from poll() above, or flush() below, when the message has*

*# been successfully delivered or failed permanently.*

*# 生产者生产数据要提供一个topic的参数, 数据是二进制方式的, 所以要使用encode来进行编码. 产生完数据之后会调用callback中的回调函数*

p.produce('mytopic', data.encode('utf-8'), callback=delivery\_report)

*# Wait for any outstanding messages to be delivered and delivery report*

*# callbacks to be triggered.*

p.flush()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

kafka\_consumer.py

**from** **confluent\_kafka** **import** Consumer, KafkaError, TopicPartition

c = Consumer({

*# 运行服务的地址*

'bootstrap.servers': '10.211.55.3:29092',

*# 每个消费者都有一个分组,*

'group.id': 'mygroup2',

*# 消费者的配置*

'default.topic.config': {

'auto.offset.reset': 'smallest' *# largest*

}

})

*# tp = TopicPartition("mytopic", 2, 0)*

*# c.assign([tp])*

*# c.seek(tp)*

*# 生产者中指定了消息的topic, 消费时也要指定topic*

c.subscribe(['mytopic'])

**while** True:

*# 从kafka中获取数据, 判断数据, 如果不是None, 不是错误, 就把消息打印出来.*

msg = c.poll(1.0)

**if** msg **is** None:

**continue**

**if** msg.error():

**if** msg.error().code() == KafkaError.\_PARTITION\_EOF:

**continue**

**else**:

**print**(msg.error())

**break**

**print**('Received message: {} [{}]: {}'.format(msg.topic(), msg.partition(), msg.value().decode('utf-8')))

c.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

运行kafka\_producer.py, 就会看到消息保存到mytopic中的提示

运行kafka\_consumer.py, 稍等片刻等待服务器发送数据. 就会看到接收到的消息.

再次在kafka\_producer中生产数据, kafka\_consumer中几乎会立即接收到消息.

桥接模式下使用kafka发送和接收数据

修改docker-compose.yml文件

version: '2'

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

*# 指定网络模式为host, 即直接使用宿主机的网络. 注释掉network\_mode, 就会自动使用bridge网络*

*# network\_mode: host*

*# zookeeper和kafka启动时内部都有一些配置项, 使用enviroment来更改启动时的配置项.*

environment:

*# zookeeper提供服务的端口*

ZOOKEEPER\_CLIENT\_PORT: 32181

ZOOKEEPER\_TICK\_TIME: 2000

*# extra\_hosts:*

*# - "moby:127.0.0.1"*

kafka:

image: confluentinc/cp-kafka:latest

*# 如果kafka的网络模式也指定为host, 服务是运行在宿主机上的, kafka的地址与宿主机的地址完全一样, 此时. KAFKA\_ZOOKEEPER\_CONNECT中的地址可以使用localhost, 表示与宿主机中的一样, 如果使用的是其它的网络模式, 如bridge, KAFKA\_ZOOKEEPER\_CONNECT中的地址就要使用zookeeper.*

*# network\_mode: host*

depends\_on:

- zookeeper

*# 对外部提供的服务是暴露在29092端口的, 对内提供的服务时使用的是9092端口.*

ports:

- 29092:29092

*# 修改配置文件中对应项的值, 百度一下kafka配置项, 前面加上KAFKA\_的前缀, 点号变为下划线, 全部大写.*

environment:

KAFKA\_BROKER\_ID: 1

*# host模式下, 这里就要使用localhost或者是主机地址, 不能使用域名zookeeper*

*# kafka中zookeeper中的端口要与上面zookeeper中的配置一致.*

*# 在bridge桥接模式下, 因为zookeeper和kafka之间的服务是相对隔离的, 这里就不能使用localhost了, 必须要指定ip地址或域名zookeeper*

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:32181

*# 这里的地址是客户端生产者和消费者连接kafka时的地址,*

*# host模式下，地址可以写localhost, 此时只能在本地访问kafka服务，但如果想要在host模式下提供外部访问，那么必须使用宿主机的地址, 并且开放相应的端口*

*# 在bridge模式下, kafka, zookeeper和宿主机的网络已经隔离开, 通过ip地址和端口访问, 此时必须要对外开放这里的端口并进行端口映射, 作为客户端的生产者和消费者才能连接服务.*

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://10.211.55.3:29092

*# 每个分区数据保存的总的份数. 1时无副本, 2时副本数为1. 注意指定的数据保存的总份数要小于等于启动的kafka服务的数量*

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

*# extra\_hosts:*

*# - "moby:127.0.0.1"*

在单节点上运行多个kafka服务

修改服务名, 修改服务端口并在防火墙中开放端口, 修改kafka\_docker\_id

version: '2'

services:

zookeeper:

image: confluentinc/cp-zookeeper:latest

*# 指定网络模式为host, 即直接使用宿主机的网络. 注释掉network\_mode, 就会自动使用bridge网络*

*# network\_mode: host*

*# zookeeper和kafka启动时内部都有一些配置项, 使用enviroment来更改启动时的配置项.*

environment:

*# zookeeper提供服务的端口*

ZOOKEEPER\_CLIENT\_PORT: 32181

ZOOKEEPER\_TICK\_TIME: 2000

*# extra\_hosts:*

*# - "moby:127.0.0.1"*

kafka:

image: confluentinc/cp-kafka:latest

*# 如果kafka的网络模式也指定为host, 服务是运行在宿主机上的, kafka的地址与宿主机的地址完全一样, 此时. KAFKA\_ZOOKEEPER\_CONNECT中的地址可以使用localhost, 表示与宿主机中的一样, 如果使用的是其它的网络模式, 如bridge, KAFKA\_ZOOKEEPER\_CONNECT中的地址就要使用zookeeper.*

*# network\_mode: host*

depends\_on:

- zookeeper

*# 对外部提供的服务是暴露在29092端口的, 对内提供的服务时使用的是9092端口.*

ports:

- 29092:29092

*# 修改配置文件中对应项的值, 百度一下kafka配置项, 前面加上KAFKA\_的前缀, 点号变为下划线, 全部大写.*

environment:

KAFKA\_BROKER\_ID: 100

*# host模式下, 这里就要使用localhost或者是主机地址, 不能使用域名zookeeper*

*# kafka中zookeeper中的端口要与上面zookeeper中的配置一致.*

*# 在bridge桥接模式下, 因为zookeeper和kafka之间的服务是相对隔离的, 这里就不能使用localhost了, 必须要指定ip地址或域名zookeeper*

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:32181

*# 这里的地址是客户端生产者和消费者连接kafka时的地址,*

*# host模式下，地址可以写localhost, 此时只能在本地访问kafka服务，但如果想要在host模式下提供外部访问，那么必须使用宿主机的地址, 并且开放相应的端口*

*# 在bridge模式下, kafka, zookeeper和宿主机的网络已经隔离开, 通过ip地址和端口访问, 此时必须要对外开放这里的端口并进行端口映射, 作为客户端的生产者和消费者才能连接服务.*

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://10.211.55.3:29092

*# 每个分区数据保存的总的份数. 1时无副本, 2时副本数为1. 注意指定的数据保存的总份数要小于等于启动的kafka服务的数量*

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

*# extra\_hosts:*

*# - "moby:127.0.0.1"*

*# 启动多个kafka服务时, 需要修改服务名*

kafka2:

image: confluentinc/cp-kafka:latest

depends\_on:

- zookeeper

*# 因为kafka1已经使用了29091端口, 这里就要使用不同的端口了. 注意要打开防火墙中对应的端口.*

ports:

- 29093:29093

environment:

*# 多个kafka服务的broker\_id一定不要相同*

KAFKA\_BROKER\_ID: 200

KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:32181

KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://10.211.55.3:29093

KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR: 1

停止容器, 再次运行, 再运行生产者和消费者进行测试.

### 3.6 消息队列之Kafka使用介绍

* Kafka启动：
  + 单节点单broker
  + 单节点多broker
* Kafka使用时的显著特征
  + **分区**之间是无序的，但**分区内的消息**是有序的.

在每个topic中的数据都保存在一个分区的情况下, 运行生产者和消费者, 生产者中生产的消息和消费者中消费的数据的顺序是相同的, 分区号使用[0]来表示. 此时所有的消息都保存在一个分区中, 即一个分区内的消息是有序的. 修改docker-compose.yml, 指定每个topic使用的分区数量. KAFKA\_NUM\_PARTITIONS: 3. 停止docker-compose, 再次运行docker-compose, 运行生产者生产数据, 此时所有的消息是随机保存到每个分区中的, 也就是说消息在各个分区之间是无序的. 在保存消息时, 是随机逐个分区进行消息的保存的, 一个分区中的所有消息都保存结束后才向另一个分区中保存消息, 选择向哪个分区中保存数据也是无序的, 每个分区中消息的数量几乎相等. 运行消费者从分区中读取数据. 也是随机逐个分区进行消息的读取的, 并且每个分区中消息的顺序与保存时的顺序一致. 即消息在每个分区内是有序的.

* + 对于topic的消费，**消费者的数量** 应 不多于 该**topic分区的数量**，否则多余的消费者将必定无法接收到消息
  + 一个消费者可同时消费多个topic
  + 在订阅消费时，Kafka保证每条消息在同一个**Consumer Group**里只会被某一个Consumer消费

#### 向指定分区生产和消费数据

在存在多分区的情况下, kafka生产和消费数据时数据的顺序就会不同, 为了达到生产和消费数据时顺序的一致, 可以使得生产和消费时使用同一个分区, 由于分区内的消息是有序的, 这样就能保证生产和消费的数据的顺序一致.

在生产消息和消费消息时指定特定的分区.

查看文档说明.

<https://docs.confluent.io/current/clients/confluent-kafka-python/index.html>

# produce(topic[, value][, key][, partition][, on\_delivery][, timestamp][, headers])

修改kafka\_producer.py, 把数据保存到分区0中.

*# 指定生产的消息保存到特定的分区中.*

p.produce('mytopic', data.encode('utf-8'), partition=0, callback=delivery\_report)

运行生产者, 在不修改kafka\_consumer.py的情况下, 运行kafka\_consumer.py, 因为只有0号分区中有数据, 所以数据全部来源自0号分区.

如果所有的3个分区中都有数据, 此时数据的消费顺序也是无序的. 停止生产者和消费者的运行, 取消掉kafka\_producer.py中的保存到特定分区的语句, 还把数据保存到3个分区中, 然后再次修改代码, 生产数据并只保存到分区0中. 然后运行kafka\_consumer.py进行消费, 此时就会从所有的3个分区中消费数据, 但是消费的分区的顺序是随机的, 是由kafka随机设定的. 想要指定分区进行消费, 就要在kafka\_consumer.py中使用c.seek()结合TopicPartition来指定消费的分区.

修改kafka\_consumer.py, 指定分区进行消费.

**from** **confluent\_kafka** **import** Consumer, KafkaError, TopicPartition

c = Consumer({

*# 运行服务的地址*

'bootstrap.servers': '10.211.55.3:29092',

*# 每个消费者都有一个分组,*

'group.id': 'mygroup2',

*# 消费者的默认配置*

'default.topic.config': {

*# 'auto.offset.reset'重置每个分区中进行消费时的偏移量, smallest相当于把偏移量设计为0. 此时从0号数据开始进行消费, 消费的数据包含历史数据和新产生的数据. 如果设置为largest, 表示把偏移量设置为最大值, 此时消费的数据不包含历史数据, 只消费新增数据.*

*# 如果使用c.seek()来进行消费, TopicPartition中定义的offset就会覆盖这里默认的偏移量.*

'auto.offset.reset': 'smallest' *# largest*

}

})

*# 创建TopicPartition的参数, 指定消费哪个分区中的数据.*

*# 每一个分区内部数据是有序的, 既然是有序的, 就有一个类似于下标的内容, offset就表示分区内数据的下标或索引.*

*# 在TopicPartition中可以手动指定offset的值, 指明从分区中的哪一个数据开始进行消费. offset为大于等于0的整数. 设置为0表示smallest, 每次消费都从索引为0的数据开始消息, 消费时既包含历史数据, 又包含新产生的数据. 这里的offset设置会覆盖掉上面auto.offset.reset中的值. 而在使用c.subscribe的时候, 不用手动设置偏移量, 会自动使用上面auto.offset.reset中默认的设置.*

*# TopicPartition(topic\_name, partition\_id, offset)*

tp = TopicPartition("mytopic", 2, 0)

*# 把tp放在列表中传递给c.assign, 也就是上面可以设置多个tp*

c.assign([tp])

*# 使用c.seek()来指定读取数据时从哪个分区中读. 需要传递TopicPartition的对象为参数.*

*# 使用c.seek()时就不需要使用c.subscribe来读取消息了.*

c.seek(tp)

*# 生产者中指定了消息的topic, 消费时也要指定topic. subscribe中没有指定分区进行消费的选项.*

*# 使用c.subscribe时就不使用c.seek读取消息了.*

*# c.subscribe表示消费者指定消费哪个主题的数据, 但subscribe没有指定分区的功能, 要使用TopicPartition来完成指定分区进行消费的功能.*

*# 在使用c.subscribe的时候, 会自动使用上面auto.offset.reset中默认的设置.*

*# c.subscribe(['mytopic'])*

**while** True:

*# 从kafka中获取数据, 判断数据, 如果不是None, 不是错误, 就把消息打印出来.*

msg = c.poll(1.0)

**if** msg **is** None:

**continue**

**if** msg.error():

**if** msg.error().code() == KafkaError.\_PARTITION\_EOF:

**continue**

**else**:

**print**(msg.error())

**break**

**print**('Received message: {} [{}]: {}'.format(msg.topic(), msg.partition(), msg.value().decode('utf-8')))

c.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

修改并运行kafka\_producer.py, 向指定的分区中写入消息, 再运行kafka\_consumer.py, 从指定的分区中读取消息. 注意, 如果tp的代码中设置了offset=0, tp = TopicPartition("mytopic", 2, 0), 那么每次读取的数据都是从索引0处开始读取的, 每次都会读取一个分区中所有的数据. 如果指定offset的值为负值, 则只读取新的数据. 而使用c.subscribe(), 则只会读取新的数据, 不会从0索引处开始读取.

#### 消费者组的概念.

在订阅消费subscribe消费模式下, kafka保证每条消息在同一个consumer group里只会被某一个consumer消费.

在kafka\_consumer.py中实例化Consumer() 对象时, 有一个group.id的参数.

c = Consumer({

*# 运行服务的地址*

'bootstrap.servers': '10.211.55.3:29092',

*# 每个消费者都有一个分组,*

'group.id': 'mygroup2',

*# 消费者的默认配置*

'default.topic.config': {

'auto.offset.reset': 'smallest' *# largest*

}

})

这里的group.id中设置的就是消费者所处的组. 停止docker-compose, 再次运行, 停止之前所有的消费者和生产者, 修改消费者消费的方式为subscribe, 以"mygroup"为组名启动两个kafka\_consumer, 修改group.id为"mygroup2", 再次启动一个kafka\_consumer, 那么, 前两个consumer都属于"mygroup"组的消费者, 后一个属于"mygroup2"组的消费者.

同一个消费者组中的数据不会重复, 只会对同一条数据消费一次. 而在不同的组中, 所有的数据都会重新进行消费. 不指定消息保存的分区, 启动生产者, 向所有3个分区中写入数据. 查看相同消费者组中的消费者, 其中没有重复的数据, 一个分区只会在相同消费者组中的某一个消费者中进行消费, 不会同时在同一个消费者组中的两个或多个消费者中进行消费. 同一消费者组中所有消费者消费数据的总和才等于生产者生产的所有数据.

如果使用c.seek()的方式进行消费, 就不会再遵守这个规律.

#### 一个消费者消费多个topic

在kafka\_consumer.py中, 如果使用c.subscribe()方式进行消费, 其中可以以列表的形式传递多个topic, 如:

c.subscribe(['mytopic', 'mytopic2'])

而在c.seek()消费方式中, 可以构建多个TopicPartition对象, 使用c.assign指定多个TP对象, 然后进行消费. 如:

tp = TopicPartition("mytopic", 2, 0)

tp2 = TopicPartition("mytopic2", 2, 0)

c.assign([tp, tp2])

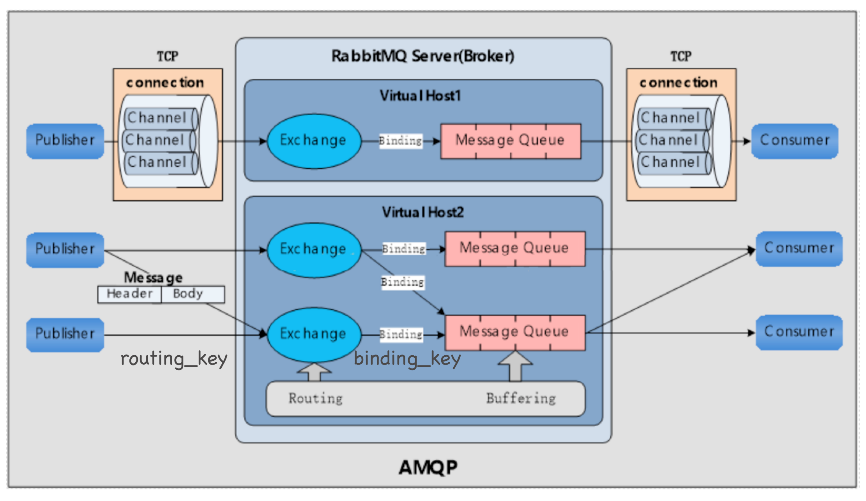
c.seek(tp)

对于topic的消费，消费者的数量 应 不多于 该topic分区的数量，否则多余的消费者将必定无法接收到消息

在docker-compose.yml中使用KAFKA\_NUM\_PARTITIONS: 3指定每个topic使用的分区数, 而在运行kafka\_consumer.py时, 多次执行, 或者创建多个消费者对象进行消费, 如果消费者的数量大于topic分区的数量, 因为同一组中的消费者之间的数据不会重复, 就会有多余的消费者无法接收到数据.

修改代码, 多次运行kafka\_consumer.py, 使用subscribe方式从多个分区中读取数据. 生产数据和读取数据时都不指定分区. 一个topic的消费者运行单个消费者实例, 另一个topic的消费者运行大于3个实例的消费者. 运行kafka\_producer.py, 向多个分区中写入2个不同topic的数据, 此时会有多余的消费者没有接收到数据. 并且, 分区是和消费者进行绑定的, 一旦某个消费都第一次从某个分区中读取了数据, 它就绑定了该分区, 其它的消费者就不能再从该分区中读取数据了.

### 3.7 消息队列之RabbitMQ – 工作原理



AMQP高级消息队列传输协议

Rbmq是基于AMQP协议构建的消息队列管理程序.

Rbmq是基于生产者消费者模式实现的, 服务端Rb Server, 也称为Broker, 但Rb本身没有集成类似于kafka中的集群功能, 想要实现Rb的功能, 必须要借助为其它的集群工具, 在集群工具中搭建Rb功能, 才能实现Rb的集群功能. 但就单个Broker服务来说, Rb与kafka的功能类似, 都是对外提供消息队列服务的.

生产者Publisher和消费者Consumer都通过Rb的5672端口来连接Rb服务, Rb服务连接后返回的是连接对象connection, 连接对象connection不能直接向Rb服务写入或读取消息/数据, 必须还要创建通信频道channel, 由频道channel来完成具体的操作. 一个channel可以理解为一个单独的长连接对象, 也就是说connection连接对象只是存放了连接的参数, 而channel才是真正创建了连接.

在RabbitMQ Server即Broker的内部, 划分出来多个Virtual Host, 可以理解为虚拟的宿主机, 类似于Mysql中创建的数据库, 每一个宿主机可以连接特定的生产者和消费者, 向其它写入和读取数据, 这里特定的生产者和消费者类似于给mysql数据库分配的特定的用户. 特定的生产者和消费都只能连接到自己对应的Virtual Host中进行操作, 这样就使用了数据的分离, 权限的分离和用户的分离. 在Rb服务运行之后, 会自动创建出来一个默认的Virtual Host, 如果想要使用另外的VH, 就要手动进行创建和配置.

在每一个VH的内部, 都包含一个Exchange交换机和Message Queue消息队列, 这里的消息队列Message Queue和之前我们手动创建的基于内存和redis的队列功能是相同的, 都是用来存储数据的. 所以RabbitMQ中数据实际上也是存储在Message Queue队列中的. 可以通过一个参数的配置来实现RabbitMQ中的数据是否以持久化的方式存储的, 即是保存到内存中还是磁盘中. Message Queue是与消费者绑定的, 一个Message Queue至少有一个消费者与之对应, 而一个消费者则可以从多个Message Queue中读取数据.

Exchange交换机实现的功能是向消息队列Message Queue中分发数据. 生产者生产的数据先交给交换机, 再由交换机提交给Message Queue进行保存, Exchange交换机本向没有保存消息的功能. 生产者生成的数据需要指定提交给哪个交换机, 交换机拿到数据之后再根据一定的规则提交给消息队列进行保存. 在单个Virtual Host虚拟主机中可以存在多个交换机, 多个Message Queue队列, 生产者和交换机, 交换机和消息队列之间就需要有一定的对应关系, 生产者和交换机之间通过routing\_key来进行关联, 确定它们之间的对应关系, 即使用routing\_key来确定哪个生产者绑定哪个交换机. 而交换机和消息队列之间则通过binding\_key进行关联的, 由于消息队列是和消费者绑定的, 所以binding\_key也相当于确定了交换机与消费机之间的对应关系.

### 3.8 消息队列之RabbitMQ – 环境

* RabbitMQ 服务端的安装
  + [RabbitMQ的docker镜像](https://github.com/docker-library/docs/tree/master/rabbitmq)

https://github.com/docker-library/docs/tree/master/rabbitmq

使用alpine版的rabbitmq,

kafka镜像配置的内容很多, 所以是使用dockerfile生产容器的, 而rabbitmq只需要运行一个容器就可以了, 所以就不用使用dockerfile构建了.

# 运行rabbitmq守护进程. 因为要对外提供服务, 在不知道ip地址的情况下, 可以通过设置的hostname来进行, 但在多个网段的情况下, hostname也不太好用. 此时可以通过配置端口映射来解决. 如果不想使用端口映射, 就必须知道容器的运行地址.

# docker run -d --hostname my-rabbit --name some-rabbit rabbitmq:3

docker run -d --hostname my-rabbit --name some-rabbit -p 5672:5672 rabbitmq:alpine

# 开放防火墙的5672端口. 默认是使用bridge桥接网络, 桥接网络的容器和宿主机是隔离开的, 想要进行通信, 二者之间的防火墙必须要打通.

firewall-cmd --list-ports

firewall-cmd --zone=public --add-port=5672/tcp --permanent

service firewalld restart

新建producer.py生产者py文件, 运行, 如果消息成功保存到服务器中, 表示服务器的配置和运行都是成功的.

**import** **pika**

*# 此处的host中的地址为运行rabbitmq的linux-server的ip地址*

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = 'queue'

channel.queue\_declare(queue=queue\_name)

**for** i **in** range(100):

message = "data**%d**"%i

channel.basic\_publish(exchange='',

routing\_key=queue\_name,

body=message)

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

* RabbitMQ的Python客户端：pika
  + [Git地址](https://github.com/pika/pika)
  + [详细文档](https://pika.readthedocs.io/en/0.10.0/examples.html)

<https://github.com/pika/pika>

pip install pika

# 可以通过实例来学习RabbitMQ.

<https://pika.readthedocs.io/en/0.10.0/examples.html>

### 3.9 消息队列之RabbitMQ – 重要概念详解

* pika中的生产者与消费者. 基于pika构建出生产者和消息者.
* queue常用属性
  + durable：表示持久化队列，rabbitmq服务重启后队列不丢失
  + exclusive：表示队列是否对当前连接特有，其它连接不能使用，当前连接断开后队列会消失。exclusive和durable是互斥的
  + auto\_delete：表示当消费者不再使用队列使用时会自动删除
* 消费者的Ack确认机制
* routing\_key与binding\_key
* Exchange常用属性(durable、auto\_delete)及其类型
  + 直连交换机：direct转发消息到RK指定的队列(BK严格匹配RK)
  + 扇形交换机：fanout 转发消息到所有绑定队列
  + 主题交换机：topic按RK规则转发(BK模糊匹配RK)
  + 首部交换机：headers根据发送消息中的headers属性进行匹配

#### 基于pika实现生产者和消费者

不使用交换机

Rb的一个特色就是交换机, 它能够很方便的实现一些复杂的特殊功能, 不使用交换机是一种特殊类型的连接, 数据会直接由生产者交给队列, 再由队列交给消费者, 一般很少使用.

生产者

新建rabbitmq\_demo/no\_exchange/basic\_demo/producer.py,

**import** **pika**

*# 构建出连接对象,*

*# 此处的host中的地址为运行rabbitmq的linux-server的ip地址*

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

*# 基于channel对象来完成具体的操作*

channel = connection.channel()

*# 在不使用交换机的情况下, 是直接使用消息队列来进行通信和数据传递的, 生产的消息是直接保存在消息队列中的, 所以要产生定义一个队列, 并且使用 channel 的 queue\_declare 方法来进行声明, 指定使用的是这个队列.*

queue\_name = 'queue'

channel.queue\_declare(queue=queue\_name)

**for** i **in** range(100):

message = "data**%d**"%i

*# 生产者和消费者都会创建连接对象, 但生产者是使用 channel 的 basic\_publish 方法来发布消息, 而消费者则是使用 basic\_consume 来接收消息.*

*# 所以生产者和消费者是通过channel不同的方法来进行区分的, 而kafka中是使用生产者类和消费者类来进行区分的.*

channel.basic\_publish(

exchange='',

*# 使用 routing\_key 指定生产者如何与交换机进行匹配的, 在不使用交换机的情况下, routing\_key 要指定向哪个消息队列中发送消息. routing\_key 要指定为队列的名称.*

routing\_key=queue\_name,

body=message

)

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建rabbitmq\_demo/no\_exchange/basic\_demo/consumer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 消费者中要使用和生产者中相同的消息队列*

queue\_name = "queue"

channel.queue\_declare(queue=queue\_name)

**print**(' [\*] Waiting for messages. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] Received **%r**" % body)

**print**(" [x] Done")

*# basic\_consume 方法只是进行了消费消息的配置, 并没有真正开始对消息进行消费, 需要调用 channel 的 start\_consuming 方法开始对消息进行消息.*

channel.basic\_consume(

callback,

*# 使用queue来确定是从哪个队列中来接收消息.*

queue=queue\_name,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

#### rabbitmq中队列的特征.

queue常用属性

durable：表示持久化队列，rabbitmq服务重启后队列不丢失. 使用持久化队列时, 重启rabbitmq服务时, 队列不丢失, 但消息和数据不一定会丢失, 是通过消息和数据的持久化来配置的. 而使用非持久化队列时, 重启rabbitmq服务后, 队列和消息/数据都会丢失.

exclusive：表示队列是否对当前连接特有，其它连接不能使用，如果是exclusive=True, 表示队列是当前连接特有的, 只能用当前连接中的消息者来连接, 如果使用其它的消息者连接同一个队列, 就会冲突报错.

使用exclusive=True的队列, 当前连接断开后队列会消失, 队列中未被消费的数据则会丢失。启动消费者, 生产数据并保存到队列中, 然后断开消费者的连接, 此时队列会消失, 其中还未被消费的数据也会丢失.

exclusive和durable参数是互斥的. durable表示持久化的队列, exclusive=True的队列是非持久化的, 所以二者是互斥的, 同时定义了durable=True后, 再使用exclusive=True, 则durable=True的设置会被覆盖掉, 不起作用. 相当于durable=False.

通常使用exclusive=True来构建一个临时队列.

auto\_delete：表示当消费者不再使用队列使用时会自动删除. 相当于exclusive=True中的一个功能, 它与durable=True也是互斥的. 即删除了队列无法实现持久化的功能.

#### 不使用交换机的持久化队列

生产者

新建rabbitmq\_demo/noexchange/queue\_usage/producer.py,

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = 'queue'

*# duable=True持久化的队列. 队列持久化不等同于消息持久化. 想要实现消息的持久化, 还要在channel.basic\_publish中配置.*

*# durable, auto\_delete, exclusive的默认值都是False*

channel.queue\_declare(queue=queue\_name, durable=True)

**for** i **in** range(100):

message = "data**%d**" % i

channel.basic\_publish(

exchange='',

routing\_key=queue\_name,

body=message,

*# 在队列持久化的基础上, 实现消息的持久化. 相当于把消息保存到磁盘中.*

properties=pika.BasicProperties(

*# delivery\_mode 默认值为1, 也就是消息和数据不是持久化的. rabbitmq在重启之后, delivery\_mode=1的消息都会被清空, 队列还会存在.*

delivery\_mode = 2, *# make message persistent*

))

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = "queue"

channel.queue\_declare(queue=queue\_name, durable=True)

**print**(' [\*] Waiting for messages. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] Received **%r**" % body)

**print**(" [x] Done")

channel.basic\_consume(

callback,

queue=queue\_name,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

修改producer.py, 给队列添加auto\_delete=True的参数. 删除容器, 重新运行, 或者删除原来的队列, 重新运行生产者.

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = 'queue'

*# duable=True持久化的队列. 队列持久化不等同于消息持久化. 想要实现消息的持久化, 还要在channel.basic\_publish中配置.*

*# durable, auto\_delete, exclusive的默认值都是False*

*# channel.queue\_declare(queue=queue\_name, durable=True)*

*# 如果设置了auto\_delete=True, 则消息就不是持久化的了.*

*# 在没有设置 auto\_delete=True 参数之前运行了生产者, 就会在 rabbitmq 服务器中产生名为 "queue" 的队列, 在修改完队列的参数之后, 想要使用同名的队列, 直接运行 producer.py 会报错. 也就是队列一旦生成之后就不能再进行修改, 想要使用同名的队列, 必须要先删除之前同名的队列. 可以使用 queue\_delete() 方法来删除, 也可以删除容器, 重新运行.*

channel.queue\_declare(queue=queue\_name, durable=True, auto\_delete=True)

**for** i **in** range(100):

message = "data**%d**" % i

channel.basic\_publish(

exchange='',

routing\_key=queue\_name,

body=message,

*# 在队列持久化的基础上, 实现消息的持久化. 相当于把消息保存到磁盘中.*

properties=pika.BasicProperties(

*# delivery\_mode 默认值为1, 也就是消息和数据不是持久化的. rabbitmq在重启之后, delivery\_mode=1的消息都会被清空, 队列还会存在.*

delivery\_mode = 2, *# make message persistent*

))

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

修改consumer.py, 给队列添加auto\_delete=True的参数.

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = "queue"

*# channel.queue\_declare(queue=queue\_name, durable=True)*

*# 如果生产者中使用了auto\_delete=True的参数, 消费者中也必须使用相同的参数, 即生产者和消费者中队列的参数必须一致, 否则就会报错.*

channel.queue\_declare(queue=queue\_name, durable=True, auto\_delete=True)

**print**(' [\*] Waiting for messages. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] Received **%r**" % body)

**print**(" [x] Done")

channel.basic\_consume(

callback,

queue=queue\_name,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

#### 消费者的Ack确认机制

rabblitmq中的客户端与服务器之间有一着主次的关系, 服务器占主动地位, 客户端占被动地位, 服务器会尽可能的保证自己不被阻塞, 也就会尽可能多的向客户端消费者发送消息, 如果消费者接收的消息没有限制, 服务器就会把所有的消息全部一次性的发送给消费者, 而不去考虑客户端接收到消息后如何处理或者多长时间能处理完的问题. 如果客户端接收到服务器发送的所有数据后, 在处理消息的过程中因为种种原因退出了, 再次启动客户时因为服务器已经发送了所有的消息, 不会再次向客户端发送消息, 而客户端因为异常退出也导致之前接收到的消息丢失, 这样就造成了数据的缺失.

为了确定服务器发送的消息是否已经被客户端接收到, 就可以在服务器和客户端通信时启用Ack确认机制, 客户端消费者在接收到服务器发送的一条数据后, 就向服务器发送一条确认消息, 服务器只有在收到客户端发送的确认消息后, 才会认为这条数据已经被客户端接收. 如果客户端在处理的过程中出现异常退出, 因为服务器没有收到某条数据的确认消息, 就会认为此消息没有被客户端接收到, 就会重新发送本条消息, 这样就能保证消息的完整性.

而对于kafka, 程序的阻塞会发生在服务器端, 服务器不会一次性的把数据全部发送给客户端, 而是客户端处理一条, 服务器发送一条. 而rabbitmq中, 服务器会一次性把数据发送给消费者, 所以程序的阻塞会发生在消费者客户端的.

不使用ack确认机制时

生产者

新建rabbitmq\_demo/no\_exchange/no\_ack/producer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = 'queue'

channel.queue\_declare(queue=queue\_name)

**for** i **in** range(100):

message = "data**%d**" % i

channel.basic\_publish(

exchange='',

routing\_key=queue\_name,

body=message

)

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建rabbitmq\_demo/no\_exchange/no\_ack/consumer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = "queue"

channel.queue\_declare(queue=queue\_name)

**print**(' [\*] Waiting for messages. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] Received **%r**" % body)

**import** **time**

*# 消费者每消费1个消息, 暂停1s. 模拟客户端处理消息的速度较慢.*

time.sleep(1)

**print**(" [x] Done")

channel.basic\_consume(

callback,

queue=queue\_name,

*# 消费者中指定 no\_ack=True, 表示不使用 ack 确认机制. 默认为 False, 表示启用 ack 确认机制.*

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

先运行consumer.py, 再运行producer.py, 生产100条数据, 由于客户端消费者没有设置一次性接收消息的数量, 服务器会把这100条数据全部一次性的发送给消费者. 在consumer.py运行的开始, 消费者已经接收到了这100条数据, 但因为消费者消费数据的速度很慢, 1s处理一条数据, 在处理到一定时间时停止consumer消费者程序的运行, 再次运行consumer.py, 由于第一次运行消费者时服务器一次性的把所有数据都发送给了消费者, 在第2次运行消费者时服务器就不再发送数据了. 并且在消费者程序中止运行时, 所有未处理的消息都随着程序的停止而丢失了, 所以第2次运行consumer.py时, 不会接收到任何消息. 这就算成了数据的丢失.

启用ack确认机制

生产者

新建 rabbitmq\_demo/no\_exchange/use\_ack/producer.py

**import** **pika**

**import** **sys**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = 'queue'

channel.queue\_declare(queue=queue\_name)

**for** i **in** range(100):

message = "data**%d**"%i

channel.basic\_publish(

exchange='',

routing\_key=queue\_name,

body=message

)

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建 rabbitmq\_demo/no\_exchange/use\_ack/consumer.py

**import** **pika**

**import** **time**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

queue\_name = "queue"

channel.queue\_declare(queue=queue\_name)

**print**(' [\*] Waiting for messages. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] Received **%r**" % body)

time.sleep(1)

**print**(" [x] Done")

*# 除了在 channel.basic\_consumer 中启用 ack 确认机制, 还需要在回调函数时定义向服务器发送的确认消息.*

*# ch.basic\_ack, 收到消息之后向服务器发送确认消息, 告诉服务器已经收到了上一条消息, 可以继续发送下一条消息了. 如果不发送消息, 服务器就认为客户端没有收到本条消息, 就会持续不断的发送本条消息.*

*# ch表示当前的channel对象. 在channel.basic\_consume方法中调用callback回调函数时, 会自动把channel对象作为第1个参数传递过去.*

ch.basic\_ack(delivery\_tag=method.delivery\_tag)

*# 指定服务端一次性发出多少条数据给消费者.*

*# 如果每次发送1条消息, 客户端接收1条确认发一条确认消息, 如果在中间执行过程中客户端中断连接, 服务器没有收到回复的确认信息, 就会在下一次启动客户端时重新发送此条数据, 这样启动ack后就不会造成消息的丢失.*

*# channel.basic\_qos(prefetch\_count=1)*

*# 如果指定服务器每次发送10条数据, 服务器会一次性的把10条数据都发送给客户端, 客户端在收到10条数据后会一条条的处理, 处理一条就向服务器发送一条回复数据. 当10条数据都处理完并且回复信息也全部由客户端发送给服务器时, 服务器才会发送下一批的10条数据. 这样就能减少系统资源的占用, 提高网络发送数据的性能.*

channel.basic\_qos(prefetch\_count=10)

*# 消费者中默认是启动了ack确认机制的, 即 no\_ack 的默认值为False, 如果省略, 就表示启动了ack确认机制.*

channel.basic\_consume(

callback,

queue=queue\_name,

no\_ack=False

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

启动producer.py, consumer.py, 在处理到一定时间会中止consumer.py的运行, 然后再次启动, 由于启动了ack机制, 服务器就会在上一次数据处理中断的地方重新发送数据. 这样就避免了数据的丢失, 保证了数据的完整性.

#### Exchange交换机

* Exchange常用属性(durable、auto\_delete)及其类型
  + 直连交换机：direct转发消息到RK指定的队列(BK严格匹配RK)
  + 扇形交换机：fanout 转发消息到所有绑定队列
  + 主题交换机：topic按RK规则转发(BK模糊匹配RK)
  + 首部交换机：headers根据发送消息中的headers属性进行匹配

##### 直连交换机

direct转发消息到RK指定的队列(BK严格匹配RK)

routing\_key的作用: 生产者产生的消息会提交给交换机, 消息在由生产者提交给交换机时会提供一个routing\_key的参数, 告诉交换机本消息要按照什么样的方式进行发布, 只是定义了发布的方式, 并不会真正进行发布, 只有消费者声明了一个队列, 同时这个队列使用binding\_key与交换机进行了绑定, 一旦routing\_key和binding\_key能够匹配成功, 这条消息才会被发送到对应的队列中.

直边交换机的routing\_key和binding\_key是严格相等的, 如都是同一个字符串.

生产者

新建 rabbitmq\_demo/exchange\_direct/producer.py

**import** **pika**

**import** **sys**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 类似于不使用交换机时的队列, 产生要使用 channel.exchange\_declare 方法来声明一个交换机. 指定交换机的名称和类型.*

*# exchange\_declare方法中的参数与队列声明 queue\_declare 中参数的功能类似.*

exchange\_name = "direct\_ex"

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='direct'

)

*# 定义消息的 routing\_key, 生产者中定义了消息的 routing\_key 之后, 通过在 channel.basic\_publish 方法中指定此 routing\_key, 生产者发布的每条消息都会带上这个参数. 消费者就可能通过消息中的 routing\_key 参数来有选择性的接收不同的消息.*

routing\_key = 'info'

**for** i **in** range(10):

message = "data**%d**-**%s**"%(i,routing\_key)

*# 在发布消息时给消息绑定交换机, 指定routing\_key参数, 这样发送的每条消息都会携带 routing\_key 的参数, 并且消息会发给绑定的交换机.*

*# 通过定义和指定交换机, 就完成了生产者和交换机的绑定.*

channel.basic\_publish(

exchange=exchange\_name,

routing\_key=routing\_key,

body=message

)

**print**(" [x] Sent **%r**:**%r**" % (routing\_key, message))

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建rabbitmq\_demo/exchange\_direct/consumer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 在直接模式中, 消费者中也要声明一个与生产者同名的交换机, 这样才能接收到生产者提交到这个交换机中的消息.*

exchange\_name = "direct\_ex"

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='direct'

)

*# 消费者中必须要声明消息队列, 生产者中不需要声明消息队列, 因为生产者产生的数据是提交给交换机的, 与消息队列没有直接的联系.*

*# 如果 channel.queue\_declare 中不写 queue 参数，那么服务端会产生一个随机队列*

*# 如果声明时指定 exclusive=True，表示生成的这个随机队列只能由当前的连接使用, 相当于实现了临时队列，该队列只有此连接存活期间有效, 只要连接关闭，该随机队列也会被删除. 通过这个方法就让每个消费者有一个自己独有的临时性队列.*

result = channel.queue\_declare(exclusive=True)

*# 指定临时性队列的名称, 在绑定队列时指定 queue 为此临时性队列的名称*

queue = result.method.queue

*# 给消费者绑定了队列之后, 为了使它与某一个交换机进行匹配, 还需要指定此队列根据什么样的规则来接收消息, 所以还需要指定 binding\_key*

*# 指当前队列要接受 routing\_key 指为 debug、info 或 warning 的消息. 相当于当前的数据是由不同级别的日志组成的, 而当前的队列只接收和消息绑定的3种级别的消息.*

binding\_keys = ["debug", "info", "warning"]

*# 多次循环绑定队列, 当前的队列和名称为 exchange\_name 的交换机绑定了多个不同名称的 binding\_key 消息. 只要生产者生产了 binding\_key 为上面列表中的消息, 这个消费者都能够接收到. 如生产者生产了 binding\_key="info" 的消息, 消费者就能够接收到.*

**for** binding\_key **in** binding\_keys:

*# 消费者在绑定消息队列时指定交换机, 队列名称和 routing\_key*

channel.queue\_bind(

exchange=exchange\_name,

queue=queue,

routing\_key=binding\_key

)

**print**(' [\*] Waiting for logs. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] **%r**:**%r**" % (method.routing\_key, body))

channel.basic\_consume(

callback,

queue=queue,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

先启动消费者, 因为没有消息, 消费者会挂起.

启动生产者, 生产10条消息, 只要生产者中指定的routing\_key 为 info, debug, warning中的一种或多种, 消费者都能够接收到.

修改生产者中的routing\_key为其它的字符串, 如 "error", 再次运行生产者生产10条数据, 消费者就不会接收到消息了.

修改消费者中的 binding\_keys 列表, 添加上 "error", 此时就能够接收到生产者生产的 routing\_key 为 "error" 的消息了.

在不关闭之间消费者的情况下启动一个修改后的消费者, 相当于有两个消费者绑定了同一个交换机, 但启动的新的消费者并没有接收到之前生产的 routing\_key 为 "error" 的消息, 这是为什么呢? 在之前生产 routing\_key 为 "error" 的消息时, 有消费者产生的临时队列与交换机绑定, 生产者生产的数据提交给交换机之后, 交换机会把所有的数据全部传递给与之绑定的队列中, 但由于之前的消费者都不能处理 routing\_key 为 "error" 的消息, 此消息就会保存在此临时队列中, 生成的新的能处理 routing\_key="error" 的消费者会生成一个新的临时性队列与交换机连接, 所以新的消费者不能从之前的消费者的临时性队列中消费数据, 并且此时服务器已经把消息传递给了消费者, 所以服务器中也不存在 routing\_key为 "error" 的消息了. 所以即使有新的能处理 routing\_key="error" 的消费者产生的临时性队列连接到交换机上了, 也不会再接收到之前生产的 routing\_key="error" 的消息.

由于在使用临时性的队列的情况下, 只要没有能够处理某个 routing\_key的消费者生成的临时队列连接到交换机, 即使有其它消费者创建的临时性队列连接到交换机中, 交换机中该routing\_key的消息要么都会被丢弃掉, 要么会首先被其它临时性队列接收到, 所以以后在接收消息时, 需要先启动消费者连接到交换机, 再启动生产者生产能够被消费者消费的消息.

保持 binding\_key能处理 "error" 的消费者连接到交换机中, 再次生产 routing\_key 为 "error" 的消息, 此时消费者就能接收到消息了.

直接模式的特点就是生产者生产消息时必须指定 routing\_key, 消费者在消费消息时必须要指定 binding\_key, 并且消费者只能接收到 routing\_key 与自己 的binding\_key 相同的消息.

在上面使用临时性队列的情况下, 队列是随机生成的, 每一个消费者都会生成自己的独有的临时性队列, 多个临时性队列之间是相互独立的, 这就造成了不存在能够消费某个routing\_key的消费者时, 对应的消息会被其它临时性队列接收并丢弃的问题, 为了解决这个问题, 可以让多个消费者使用同一个指定的队列. 修改消费者的代码, 不使用临时队列, 而是手动指定一个队列.

修改rabbitmq\_demo/exchange\_direct/consumer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 在直接模式中, 消费者中也要声明一个与生产者同名的交换机, 这样才能接收到生产者提交到这个交换机中的消息.*

exchange\_name = "direct\_ex"

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='direct'

)

*# 消费者中必须要声明消息队列, 生产者中不需要声明消息队列, 因为生产者产生的数据是提交给交换机的, 与消息队列没有直接的联系.*

*# 如果 channel.queue\_declare 中不写 queue 参数，那么服务端会产生一个随机队列*

*# 如果声明时指定 exclusive=True，表示生成的这个随机队列只能由当前的连接使用, 相当于实现了临时队列，该队列只有此连接存活期间有效, 只要连接关闭，该随机队列也会被删除. 通过这个方法就让每个消费者有一个自己独有的临时性队列.*

*# result = channel.queue\_declare(exclusive=True)*

*# 指定临时性队列的名称, 在绑定队列时指定 queue 为此临时性队列的名称*

*# queue = result.method.queue*

*# 不对每个消费者使用自己的临时性队列, 而是所有消费者都使用同一个手动指定的持久性队列.*

*# 不使用临时性的队列, 而是手动指定一个队列, 因为没有使用 exclusive=True 的参数, 所以此时使用的就是持久性的队列.*

*# 只要开始运行了消费者, 这里指定的队列就会和交换机进行绑定, 而只要把一个队列与交换机进行了绑定, 除非删除这个队列或者手动进行解绑, 否则这个队列与交换机之间的绑定关系会一直存在, 停止消费者的运行, 队列和交换机之间的绑定关系也会存在, 此时生产的数据即使没有消费者进行消费, 也会保存到此队列中, 再次运行消费者时还是能够从队列中接收服务器在消费者运行之前发送的数据的. 这样就实现了持久化的队列与数据的存储.*

queue = "new\_queue"

channel.queue\_declare(queue=queue)

*# 给消费者绑定了队列之后, 为了使它与某一个交换机进行匹配, 还需要指定此队列根据什么样的规则来接收消息, 所以还需要指定 binding\_key*

*# 指当前队列要接受 routing\_key 指为 debug、info 或 warning 的消息. 相当于当前的数据是由不同级别的日志组成的, 而当前的队列只接收和消息绑定的3种级别的消息.*

binding\_keys = ["debug", "info", "warning"]

*# 多次循环绑定队列, 当前的队列和名称为 exchange\_name 的交换机绑定了多个不同名称的 binding\_key 消息. 只要生产者生产了 binding\_key 为上面列表中的消息, 这个消费者都能够接收到. 如生产者生产了 binding\_key="info" 的消息, 消费者就能够接收到.*

**for** binding\_key **in** binding\_keys:

*# 消费者在绑定消息队列时指定交换机, 队列名称和 routing\_key*

channel.queue\_bind(

exchange=exchange\_name,

queue=queue,

routing\_key=binding\_key

)

**print**(' [\*] Waiting for logs. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] **%r**:**%r**" % (method.routing\_key, body))

channel.basic\_consume(

callback,

queue=queue,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

只要开始运行了消费者文件, 代码中指定的队列就会和交换机进行绑定, 而只要把一个队列与交换机进行了绑定, 除非删除这个队列或者手动进行解绑, 否则这个队列与交换机之间的绑定关系会一直存在, 停止消费者的运行, 队列和交换机之间的绑定关系也会存在, 此时生产的数据即使没有消费者进行消费, 也会保存到此队列中, 再次运行消费者时还是能够从队列中接收服务器在消费者运行之前发送的数据的. 这样就实现了持久化的队列与数据的存储.

##### 主题交换机

topic按RK规则转发(BK模糊匹配RK)

主题交换机除了 routing\_key 和 binding\_key 与直接交换机不同之外, 其它都与直接交换机相同.

生产者

新建rabbitmq\_demo/exchange\_topic/producer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

exchange\_name = "topic\_ex"

*# 主题交换机在声明交换机时, 要指定类型为 "topic"*

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='topic'

)

*# routing\_key 可以定义为 "a.b.c" 的形式, 直接交换机模式中如果生产者的 routing\_key 使用 "a.b.c" 模式的话, 消费者也要使用完全相同的字符串, 才能够进行匹配. 而主题交换机中, routing\_key 一般都要使用 "a.b.c" 这种模式来命令, 在消费者中的 binding\_key 则可以对消费者的 routing\_key 进行模糊匹配. routing\_key 中不能使用模糊匹配词, 必须是确定的, 只有 binding\_key 中才能使用模糊匹配.*

routing\_key = 'china.beijing.street'

routing\_key = 'beijing.street'

**for** i **in** range(10):

message = 'street-**%d**'%i

channel.basic\_publish(

exchange=exchange\_name,

routing\_key=routing\_key,

body=message

)

**print**(" [x] Sent **%r**:**%r**" % (routing\_key, message))

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建rabbitmq\_demo/exchange\_topic/consumer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

exchange\_name = "topic\_ex"

*# 主题交换机在声明交换机时, 要指定类型为 "topic"*

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='topic'

)

*# 主题交换机除了 routing\_key 和 binding\_key 与直接交换机不同之外, 其它都与直接交换机相同. 这里在绑定队列时也是使用的临时性队列.*

result = channel.queue\_declare(durable=True)

queue = result.method.queue

*# 在消费者的 binding\_key 中对生产者的 routing\_key 进行模糊匹配.*

*# \*号匹配任意一个单词；# 号匹配0个或多个单词*

*# china.a.b.c > "china.#"*

*# a.street > '\*.street'*

binding\_keys = ['\*.street', "china.#"]

**for** binding\_key **in** binding\_keys:

channel.queue\_bind(

exchange=exchange\_name,

queue=queue,

routing\_key=binding\_key

)

**print**(' [\*] Waiting for logs. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] **%r**:**%r**" % (method.routing\_key, body))

channel.basic\_consume(

callback,

queue=queue,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

##### 扇形交换机

fanout不是通过 routing\_key 与 binding\_key 的匹配来完成消息的转发的. 而是会把消息转发到所有绑定的队列中.

类似于广播的功能, 任何连接的设备都可以收到服务端发送的消息.

生产者

新建rabbitmq\_demo/exchange\_fanout/producer.py

**import** **pika**

**import** **sys**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 声明交换机时指定类型为 扇形交换机*

exchange\_name = "fanout\_ex"

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='fanout'

)

**for** i **in** range(10):

message = "data**%d**"%i

channel.basic\_publish(

exchange=exchange\_name,

*# 因为扇形交换机的消息会发送到与交换机绑定的所有队列中, 被所有连接队列的消费者接收, 所以就不需要设置 routing\_key 参数了. routing\_key 参数不能省略, 所以只能为空字符串*

routing\_key='',

body=message

)

**print**(" [x] Sent **%r**" % message)

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建rabbitmq\_demo/exchange\_fanout/consumer.py

**import** **pika**

**import** **sys**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

exchange\_name = "fanout\_ex"

channel.exchange\_declare(

exchange=exchange\_name,

exchange\_type='fanout'

)

*# 使用exclusive=True参数, 会自动创建随机命名的临时性队列*

result = channel.queue\_declare(exclusive=True)

queue\_name = result.method.queue

*# 在扇形交换机中消费者绑定队列时只需要指定队列和交换机, 不用指定 binding\_key 参数. 所有绑定到交换机上的消费者和队列都能够收到消息.*

channel.queue\_bind(

exchange=exchange\_name,

queue=queue\_name

)

**print**(' [\*] Waiting for logs. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] **%r**" % body)

channel.basic\_consume(

callback,

queue=queue\_name,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

首先运行多个消费者, 因为没有数据, 所有消费者都处于等待状态, 启动生产者, 生产10条数据, 此时所有的消费者都会接收到生产者生产的10条数据.

##### 首部交换机

headers模式的交换机也不是通过 routing\_key 与 binding\_key 的匹配来转发消息的, 而是根据发送消息中的headers属性进行匹配实现消息的转发的.

生产者

新建rabbitmq\_demo/exchange\_headers/producer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 指定交换机的类型为 首部交换机*

exchange\_type = "headers\_ex"

channel.exchange\_declare(

exchange=exchange\_type,

exchange\_type='headers'

)

*# 定义字典格式的 headers*

headers = {

"key1": "value1",

"key2": "value2"

}

*# 在生产消息时通过 properties 参数来携带首部 headers 的信息. 因为不再使用 routing\_key 进行匹配, 所以这里的 routing\_key=''*

**for** i **in** range(10):

message = "data**%d**"%i

channel.basic\_publish(

exchange=exchange\_type,

routing\_key='',

body=message,

properties=pika.BasicProperties(headers=headers)

)

**print**(" [x] Sent **%r**:**%r**" % (headers, message))

connection.close()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

消费者

新建rabbitmq\_demo/exchange\_headers/consumer.py

**import** **pika**

connection = pika.BlockingConnection(pika.ConnectionParameters(host='10.211.55.3'))

channel = connection.channel()

*# 指定交换机的类型为 首部交换机*

exchange\_type = "headers\_ex"

channel.exchange\_declare(

exchange=exchange\_type,

exchange\_type='headers'

)

*# 使用临时性的队列*

result = channel.queue\_declare(exclusive=True)

queue\_name = result.method.queue

*# all表示所有的键值都必须一一匹配，才可以*

*# any表示任意一个键值匹配上，就可以*

match\_type = "any"

*# 定义字典格式的 headers 参数, 必须包含 "x-match" 来指定匹配的类型为全部匹配 all 或任意匹配 any. 如果是 "all" 的匹配方式, 除了 "x-match": match\_type 键值对之外的键值对必须要与消费者中 headers 中定义的键值对完全匹配才能接收到对应的生产者生产的消息. 如果使用的是 "any" 的匹配方式, 则只需要部分匹配就能接收到消息.*

headers = {

"x-match": match\_type, *# "x-match"必须提供，*

'key1': 'value1',

'key2': 'value3'

}

*# 消费者在绑定队列时, 队列交换机类型, 队列名称之外, 还要提供字典格式的 headers的参数*

channel.queue\_bind(

exchange=exchange\_type,

queue=queue\_name,

arguments=headers

)

**print**(' [\*] Waiting for logs. To exit press CTRL+C')

**def** callback(ch, method, properties, body):

**print**(" [x] **%r**:**%r**:**%r**" % (match\_type, properties.headers, body))

channel.basic\_consume(

callback,

queue=queue\_name,

no\_ack=True

)

channel.start\_consuming()

**if** \_\_name\_\_ == '\_\_main\_\_':

**pass**

因为上面消费者中使用的队列也是随机命名的临时队列, 所以这里也必须要先启动消费者, 再启动生产者, 才能接收到消息.

### 3.10 消息队列之总结

一个新的技术, 一是掌握它底层的原理, 二是活用文档.

* 掌握原理. 学习某一个技术时, 必须要掌握的是支持这个技术的原理, 如rabbitmq是基于AMQB协议来实现的技术, 需要掌握流程图中的每一个环节. 只有掌握了原理, 才能在实际使用时得心应手, 也才能在使用中遇到问题时最快地找到问题所在和解决问题的方法. 而只学习api用法的话, 则很难达到这两个效果.
* 活用文档. 在使用时只要查看一下api文档, 或者参考一下示例代码或别人的代码, 就能够学会使用了.
* 多实践. 不能只停留在学会, 看懂的阶段, 还必须要多实践, 多使用.

rabbitmq的适用场景, 适合应用在需要保持消息的顺序, 同时还要保持一定的消息处理速度时使用.

而kafka, 在有多个消费者的情况下, 消息顺序会在一定程度上打乱, 同时构建出一个消费者之后, 从程序的启动到消息的接收还会有一定的延时.

在代码实现的技术选型上, 应尽量选择接近于企业级应用的技术, 如在请求管理中, 临时型的队列, 持久化队列, 以及kafka和rabbitmq之类的消息管理技术都可以使用, 在可能的情况下, 尽可以使用kafka和rabbitmq这种企业级的技术.

## 4. 断点续爬以及增量式设计

断点续爬与增量式爬取都与请求管理紧密相关,

### 4.1 断点续爬设计

* 什么是断点续爬爬虫？
  + 爬虫程序运行中途中断后，下次运行能继续根据之前的请求状态继续运行, 类比如迅雷的断点续传功能. 爬虫的请求在程序中止运行后必须不丢失, 所以请求队列必须要持久化的保存.
* 如何设计断点续爬爬虫？
  + 请求管理：请求去重持久化维护、请求队列持久化维护、请求丢失的容灾维护. 请求丢失的容灾维护. 从请求队列中取出来一个数据后, 队列中这个数据就没有了, 如果数据没有处理完程序就因为各种原因退出了, 这个数据就相当于丢失了. 如果是并发的爬虫的话, 数据的丢失可能会更加严重. 所以还必须要对丢失的请求进行处理. 可以使用类似于rabbitmq的ack机制来完成容灾的维护. 还可以保存两份请求队列, 原始请求队列中的数据在爬虫从中取出来时就会删除, 另一个请求队列中的请求在数据处理完成时才会删除, 如果程序异常退出了, 只需要对比这两个队列中差异的部分, 就能找回丢失的请求了.

程序是思路的体现, 有很多思路能够实现一个具体的功能, 也就有很多种不同的代码. 写代码之前都要通过思考设计自己的思路.

### 4.2 增量式爬虫设计

* 什么是增量式爬虫？
  + 请求的增量. 会从网页中获取到新的请求, 爬虫框架和代码中一般都会通过请求管理来实现.
  + 数据的增量. 数据的增量管理中, 最难的是对哪些数据是新的数据的判断, 也就是增量数据的标准的制定.
* 如何设计增量式爬虫？
  + 请求管理：请求去重持久化维护、请求队列的持久化维护、周期性请求（该请求不做去重判断, 如新闻网站首页的抓取）
  + 数据管理：数据的去重(可以通过类似于队列去重的方法来实现)、响应数据的判断（是否更新）

## 5. 总结