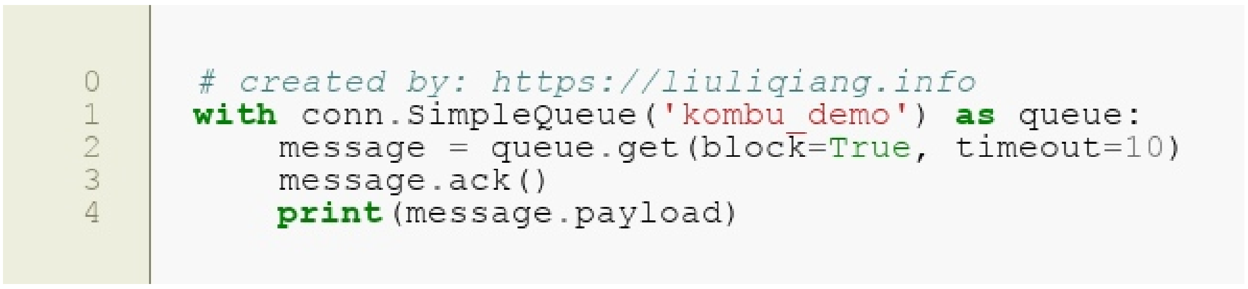
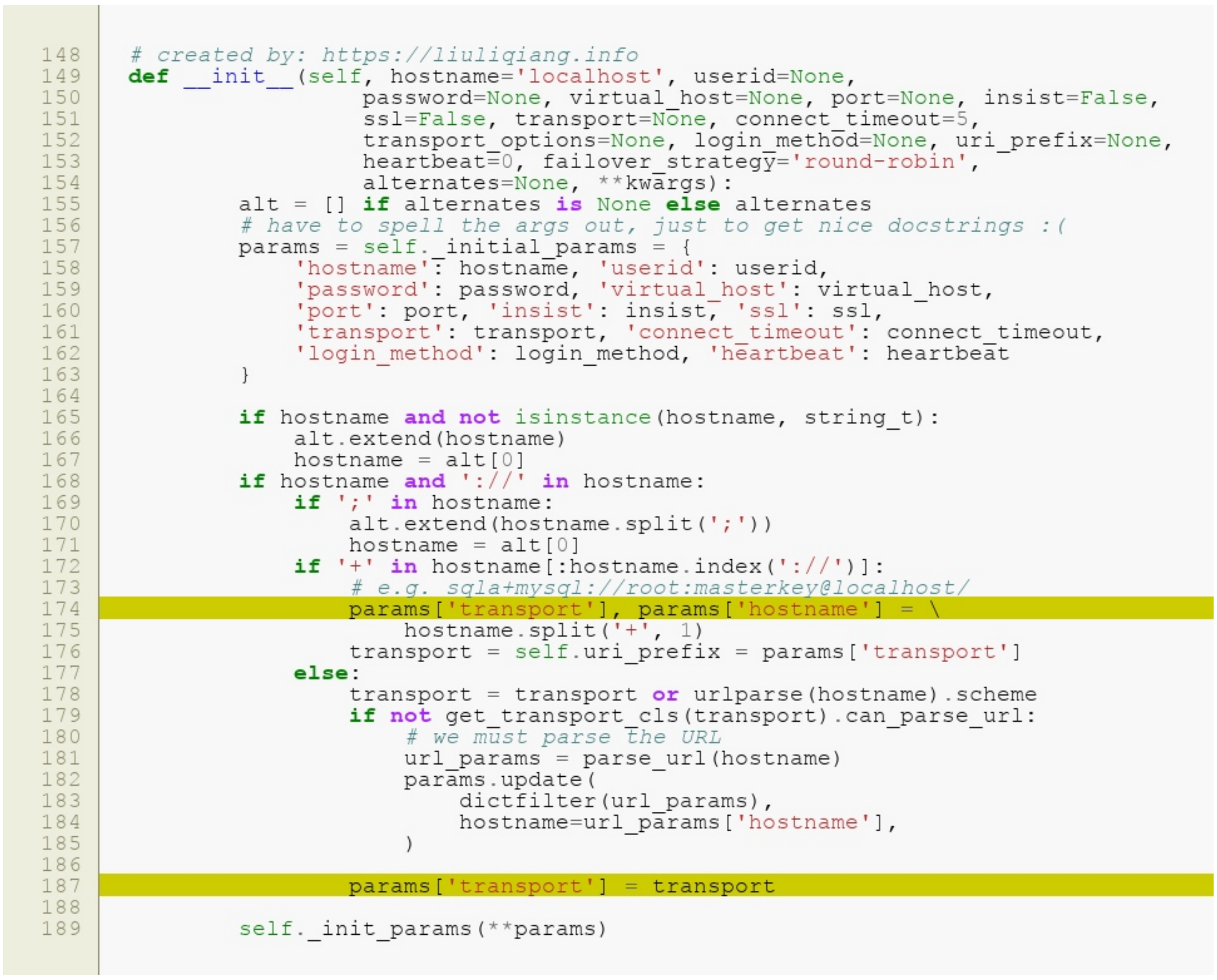
Kombu 源码解析二

OK，现在真正进入到 Kombu 的源码解析部分，我们还是从 Kombu 源码解析一 中的例子讲起，我们来看下对于一个简单的使用，在 Kombu 内部是如何实现的，首先从最开始的这段代码说起：



```python
# created by: https://liuliqiang.info
with conn.SimpleQueue('kombu_demo') as queue:
    message = queue.get(block=True, timeout=10)
    message.ack()
    print(message.payload)
```

我们先来关注一下，`conn.SimpleQueue` 发生了什么事情，然后再来看看 queue 是啥，最后应该看看 message 的内容。conn 的类型应该是 `Connection`，而我们这里用的是 **Redis**，那么应该对应的是 **Redis** 的 `Connection` 才对，具体如何，我们跟进代码看一下 (kombu/connection.py line 49)：



```python
148  # created by: https://liuliqiang.info
149  def __init__(self, hostname='localhost', userid=None,
150                   password=None, virtual_host=None, port=None, insist=False,
151                   ssl=False, transport=None, connect_timeout=5,
152                   transport_options=None, login_method=None, uri_prefix=None,
153                   heartbeat=0, failover_strategy='round-robin',
154                   alternates=None, **kwargs):
155          alt = [] if alternates is None else alternates
156          # have to spell the args out, just to get nice docstrings :(
157          params = self._initial_params = {
158              'hostname': hostname, 'userid': userid,
159              'password': password, 'virtual_host': virtual_host,
160              'port': port, 'insist': insist, 'ssl': ssl,
161              'transport': transport, 'connect_timeout': connect_timeout,
162              'login_method': login_method, 'heartbeat': heartbeat
163          }
164
165          if hostname and not isinstance(hostname, string_t):
166              alt.extend(hostname)
167              hostname = alt[0]
168          if hostname and '://' in hostname:
169              if ';' in hostname:
170                  alt.extend(hostname.split(';'))
171                  hostname = alt[0]
172              if '+' in hostname[:hostname.index('://')]:
173                  # e.g. sqla+mysql://root:masterkey@localhost/
174                  params['transport'], params['hostname'] = \
175                      hostname.split('+', 1)
176                  transport = self.uri_prefix = params['transport']
177              else:
178                  transport = transport or urlparse(hostname).scheme
179                  if not get_transport_cls(transport).can_parse_url:
180                      # we must parse the URL
181                      url_params = parse_url(hostname)
182                      params.update(
183                          dictfilter(url_params),
184                          hostname=url_params['hostname'],
185                      )
186
187                  params['transport'] = transport
188
189          self._init_params(**params)
```

跟踪 `Connection` 的构造函数我们可以发现在 Line 174 和 Line 187 里面都只是将 `transport` 的类型记住，然后在 **Line 189** 那里做了一个参数初始化，我们查看一下代码

可以发现在 **Line 246** 中又直接赋值给 `transport_cls`，这在以后有大用处：

```
230  # created by: https://liuliqiang.info
231  def _init_params(self, hostname, userid, password, virtual_host, port,
232                   insist, ssl, transport, connect_timeout,
233                   login_method, heartbeat):
234      transport = transport or 'amqp'
235      if transport == 'amqp' and supports_librabbitmq():
236          transport = 'librabbitmq'
237      self.hostname = hostname
238      self.userid = userid
239      self.password = password
240      self.login_method = login_method
241      self.virtual_host = virtual_host or self.virtual_host
242      self.port = port or self.port
243      self.insist = insist
244      self.connect_timeout = connect_timeout
245      self.ssl = ssl
246      self.transport_cls = transport
247      self.heartbeat = heartbeat and float(heartbeat)
```

OK，小插曲一段，回来看看 `SimpleQueue`，在 **Line 712** 我们可以看到很简单的一项：

```
711  # created by: https://liuliqiang.info
712  def SimpleQueue(self, name, no_ack=None, queue_opts=None,
713                  exchange_opts=None, channel=None, **kwargs):
714      """Simple persistent queue API.
715
716      Create new :class:`~kombu.simple.SimpleQueue`, using a channel
717      from this connection.
718
719      If ``name`` is a string, a queue and exchange will be automatically
720      created using that name as the name of the queue and exchange,
721      also it will be used as the default routing key.
722
723      Arguments:
724          name (str, kombu.Queue): Name of the queue/or a queue.
725          no_ack (bool): Disable acknowledgments. Default is false.
726          queue_opts (Dict): Additional keyword arguments passed to the
727              constructor of the automatically created :class:`~kombu.Queue`.
728          exchange_opts (Dict): Additional keyword arguments passed to the
729              constructor of the automatically created
730              :class:`~kombu.Exchange`.
731          channel (ChannelT): Custom channel to use. If not specified the
732              connection default channel is used.
733      """
734      from .simple import SimpleQueue
735      return SimpleQueue(channel or self, name, no_ack, queue_opts,
736                         exchange_opts, **kwargs)
```

我们可以发现，代码就这么简单，看看上面的注释，注意看下注释里面参数的意思，可以发现，这里提供了好几个默认参数：

- channel：就是这个 connection 了
- queue：就是我们传递的参数
- no_ack：None
- queue_opts：None
- exchange_opts：None

好，我们就跟进去看看这个 `SimpleQueue` 是啥，位置应该在：**kombu/simple.py** ，我们可以在 **Line 118** 上看到定义的代码，说实话，看到这个还是蛮欣喜的，因为感觉看到头了：

```python
# created by: https://liuliqiang.info
def __init__(self, channel, name, no_ack=None, queue_opts=None,
             exchange_opts=None, serializer=None,
             compression=None, **kwargs):
    queue = name
    queue_opts = dict(self.queue_opts, **queue_opts or {})
    exchange_opts = dict(self.exchange_opts, **exchange_opts or {})
    if no_ack is None:
        no_ack = self.no_ack
    if not isinstance(queue, entity.Queue):
        exchange = entity.Exchange(name, **exchange_opts)
        queue = entity.Queue(name, exchange, name, **queue_opts)
        routing_key = name
    else:
        name = queue.name
        exchange = queue.exchange
        routing_key = queue.routing_key
    consumer = messaging.Consumer(channel, queue)
    producer = messaging.Producer(channel, exchange,
                                  serializer=serializer,
                                  routing_key=routing_key,
                                  compression=compression)
    super(SimpleQueue, self).__init__(channel, producer,
                                      consumer, no_ack, **kwargs)
```

可以看到在这个 SimpleQueue 中，已经创建了 Consumer 和 Producer 了，这里我们暂时不关注他们的代码，而是根据我上一篇文章的描述理解先。我们可以关注一下 Consumer 和 Producer 的参数，他们都是默认帮我们设置的，然后这就告一段落了。

接下来，是时候看下获取消息的实现了，还是在这个文件，但是，我们可以发现 **SimpleQueue** 的代码真的很 Simple，它自己没有重载 `get` 方法，所以我们可以在它的父类 `SimpleBase` 中找到，应该在 **Line 35**：

```python
34  # created by: https://liuliqiang.info
35      def get(self, block=True, timeout=None):
36          if not block:
37              return self.get_nowait()
38
39          self._consume()
40
41          time_start = monotonic()
42          remaining = timeout
43          while True:
44              if self.buffer:
45                  return self.buffer.popleft()
46
47              if remaining is not None and remaining <= 0.0:
48                  raise self.Empty()
49
50              try:
51                  # The `drain_events` method will
52                  # block on the socket connection to rabbitmq. if any
53                  # application-level messages are received, it will put them
54                  # into `self.buffer`.
55                  # * The method will block for UP TO `timeout` milliseconds.
56                  # * The method may raise a socket.timeout exception; or...
57                  # * The method may return without having put anything on
58                  #    `self.buffer`.  This is because internal heartbeat
59                  #    messages are sent over the same socket; also POSIX makes
60                  #    no guarantees against socket calls returning early.
61                  self.channel.connection.client.drain_events(timeout=remaining)
62              except socket.timeout:
63                  raise self.Empty()
64
65              if remaining is not None:
66                  elapsed = monotonic() - time_start
67                  remaining = timeout - elapsed
```

在 **Line 36** 很庆幸，是 block 的，所以我们不需要看更多的代码了，然后这里有个很应景的 `_consume`，进去看一下：

```python
# created by: https://liuliqiang.info
 def _consume(self):
     if not self._consuming:
         self.consumer.consume(no_ack=self.no_ack)
         self._consuming = True
```

很简单，其实就是调用 **Consumer** 的 consume，然后返回，注意，这里是**非阻塞的**，那么我们要怎么拿到消息呢，继续看下去，可以看到，下面有个循环，然后着重看一下**注释**，如果有消息进来，那么就会被放到 `self.buffer` 里面，没有消息这里就会阻塞住了，同时，如果阻塞超过我们设定的超时时间，那么就会跑出异常啦。
那我们就按照正常的逻辑走吧，在 **Line 45** 就是正常拿到数据之后，然后返回去了，我们看看拿到的是啥，回到原来的代码，其实也就是我们自己写的 Sample 中：

```python
# created by: https://liuliqiang.info
message = queue.get(block=True, timeout=10)
message.ack()
```

返回的是一个 Message，我们在前面跟踪的代码里面没有体现这个 message 是什么类型，那么它是什么类型？其实你找遍 Kombu 的代码，发现其实只有一种在 **kombu/message.py** 中的 Message 类型，之前提过了，这是 Kombu 中生产/消费的基本单位，我们快速得看一下代码：

```python
# created by: https://liuliqiang.info
def __init__(self, body=None, delivery_tag=None,
             content_type=None, content_encoding=None, delivery_info={},
             properties=None, headers=None, postencode=None,
             accept=None, channel=None, **kwargs):
    self.errors = [] if self.errors is None else self.errors
    self.channel = channel
    self.delivery_tag = delivery_tag
    self.content_type = content_type
    self.content_encoding = content_encoding
    self.delivery_info = delivery_info
    self.headers = headers or {}
    self.properties = properties or {}
    self._decoded_cache = None
    self._state = 'RECEIVED'
    self.accept = accept

    compression = self.headers.get('compression')
    if not self.errors and compression:
        try:
            body = decompress(body, compression)
        except Exception:
            self.errors.append(sys.exc_info())

    if not self.errors and postencode and isinstance(body, text_t):
        try:
            body = body.encode(postencode)
        except Exception:
            self.errors.append(sys.exc_info())
    self.body = body
```

其实无非就是消息内容的封装，但是，内容还是比较丰富的，因为没啥讲究的必要，所以，内容就不讲了，我们回到后面一句对消息的处理：`message.ack` 看看这里发生了什么事情：

```python
# created by: https://liuliqiang.info
def ack(self, multiple=False):
    """Acknowledge this message as being processed.

    This will remove the message from the queue.

    Raises:
        MessageStateError: If the message has already been
            acknowledged/requeued/rejected.
    """
    if self.channel is None:
        raise self.MessageStateError(
            'This message does not have a receiving channel')
    if self.channel.no_ack_consumers is not None:
        try:
            consumer_tag = self.delivery_info['consumer_tag']
        except KeyError:
            pass
        else:
            if consumer_tag in self.channel.no_ack_consumers:
                return
    if self.acknowledged:
        raise self.MessageStateError(
            'Message already acknowledged with state: {0._state}'.format(
                self))
    self.channel.basic_ack(self.delivery_tag, multiple=multiple)
    self._state = 'ACK'
```

其实你会发现也没做啥，其实就是判断需不需要确认，确认过了没，然后这一段比较重要的是：`self.channel.basic_ack` 这里确认的方式是交给 channel（Connection） 来执行，然后就完了。

这里就是一段我们跟踪简单实现的一种方式，整个环节还是比较简单的，但是我们已经知道了一些东西。

- Connection 里面包含了 Transport，并且是抽象的，根据我们的参数确定具体的 Transport 是什么
- SimpleQueue 里面不仅仅只有 queue，还包含了 connection, consumer 和 producer
- Consumer 的 consume 是非阻塞的，真实是在 `drain_events` 中获取，然后塞到成员变量中的，但是具体怎么塞的，我们还没看到
- 消息的确认是通过 connection 来确认的，但是我们没看到有持久化之类的