12.11 实现消息发布/订阅模型¶

问题¶

你有一个基于线程通信的程序，想让它们实现发布/订阅模式的消息通信。

解决方案¶

要实现发布/订阅的消息通信模式， 你通常要引入一个单独的“交换机”或“网关”对象作为所有消息的中介。 也就是说，不直接将消息从一个任务发送到另一个，而是将其发送给交换机， 然后由交换机将它发送给一个或多个被关联任务。下面是一个非常简单的交换机实现例子：

```
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]
```

一个交换机就是一个普通对象，负责维护一个活跃的订阅者集合，并为绑定、解绑和发送消息提供相应的方法。 每个交换机通过一个名称定位，`get_exchange()` 通过给定一个名称返回相应的 `Exchange` 实例。

下面是一个简单例子，演示了如何使用一个交换机：

```
# Example of a task.  Any object with a send() method

class Task:
    ...
    def send(self, msg):
        ...

task_a = Task()
task_b = Task()

# Example of getting an exchange
exc = get_exchange('name')

# Examples of subscribing tasks to it
exc.attach(task_a)
exc.attach(task_b)

# Example of sending messages
exc.send('msg1')
exc.send('msg2')

# Example of unsubscribing
exc.detach(task_a)
exc.detach(task_b)
```

尽管对于这个问题有很多的变种，不过万变不离其宗。 消息会被发送给一个交换机，然后交换机会将它们发送给被绑定的订阅者。

讨论¶

通过队列发送消息的任务或线程的模式很容易被实现并且也非常普遍。 不过，使用发布/订阅模式的好处更加明显。

首先，使用一个交换机可以简化大部分涉及到线程通信的工作。 无需去写通过多进程模块来操作多个线程，你只需要使用这个交换机来连接它们。 某种程度上，这个就跟日志模块的工作原理类似。 实际上，它可以轻松的解耦程序中多个任务。

其次，交换机广播消息给多个订阅者的能力带来了一个全新的通信模式。 例如，你可以使用多任务系统、广播或扇出。 你还可以通过以普通订阅者身份绑定来构建调试和诊断工具。 例如，下面是一个简单的诊断类，可以显示被发送的消息：

```
class DisplayMessages:
    def __init__(self):
        self.count = 0
    def send(self, msg):
        self.count += 1
        print('msg[{}]: {!r}'.format(self.count, msg))

exc = get_exchange('name')
d = DisplayMessages()
exc.attach(d)
```

最后，该实现的一个重要特点是它能兼容多个“task-like”对象。 例如，消息接受者可以是actor（12.10小节介绍）、协程、网络连接或任何实现了正确的 `send()` 方法的东西。

关于交换机的一个可能问题是对于订阅者的正确绑定和解绑。 为了正确的管理资源，每一个绑定的订阅者必须最终要解绑。 在代码中通常会是像下面这样的模式：

```
exc = get_exchange('name')
exc.attach(some_task)
try:
    ...
finally:
    exc.detach(some_task)
```

某种意义上，这个和使用文件、锁和类似对象很像。 通常很容易会忘记最后的 `detach()` 步骤。 为了简化这个，你可以考虑使用上下文管理器协议。 例如，在交换机对象上增加一个 `subscribe()` 方法，如下：

```
from contextlib import contextmanager
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):
        for task in tasks:
            self.attach(task)
        try:
            yield
        finally:
            for task in tasks:
                self.detach(task)

    def send(self, msg):
```

```
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]

# Example of using the subscribe() method
exc = get_exchange('name')
with exc.subscribe(task_a, task_b):
     ...
     exc.send('msg1')
     exc.send('msg2')
     ...

# task_a and task_b detached here
```

最后还应该注意的是关于交换机的思想有很多种的扩展实现。 例如，交换机可以实现一整个消息通道集合或提供交换机名称的模式匹配规则。 交换机还可以被扩展到分布式计算程序中（比如，将消息路由到不同机器上面的任务中去）。