原 使用Python进行分布式系统协调（ZooKeeper，Consul， etcd ）

发表于3个月前(2015-05-05 15:06) 阅读（89）| 评论（0） 1 人收藏此文章, 取消收藏

目录[-]

笔者之前的博文提到过，随着大数据时代的到来，分布式是解决大数据问题的一个主要手段，随着越来越多的分布式的服务，如何在分布式的系统中对这些服务做协调变成了一个很棘手的问题。今天我们就来看看如何使用Python，利用开源对分布式服务做协调。

在对分布式的应用做协调的时候，主要会碰到以下的应用场景：

- 业务发现（service discovery）

  找到分布式系统中存在那些可用的服务和节点

- 名字服务（name service）

  通过给定的名字知道到对应的资源

- 配置管理（configuration management）

  如何在分布式的节点中共享配置文件，保证一致性。

- 故障发现和故障转移（failure detection and failover）

  当某一个节点出故障的时候，如何检测到并通知其它节点， 或者把想用的服务转移到其它的可用节点

- 领导选举（leader election）

  如何在众多的节点中选举一个领导者，来协调所有的节点

- 分布式的锁（distributed exclusive lock）

  如何通过锁在分布式的服务中进行同步

- 消息和通知服务（message queue and notification）

  如何在分布式的服务中传递消息，以通知的形式对事件作出主动的响应

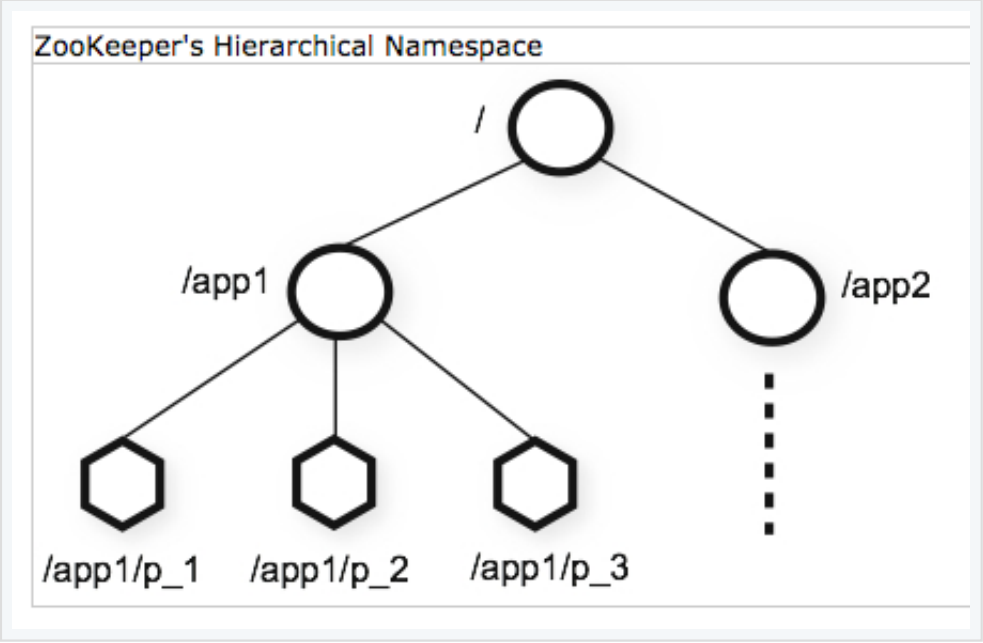
有许多的开源软件试图解决以上的全部或者部分问题，例如ZooKeeper，consul，doozerd等等，我们现在就看看它们是如何做的。

ZooKeeper

ZooKeeper是使用最广泛，也是最有名的解决分布式服务的协调问题的开源软件了，它最早和Hadoop一起开发，后来成为了Apache的顶级项目，很多开源的项目都在使用ZooKeeper，例如大名鼎鼎的Kafka。

Zookeeper本身是一个分布式的应用，通过对共享的数据的管理来实现对分布式应用的协调。

ZooKeeper使用一个树形目录作为数据模型，这个目录和文件目录类似，目录上的每一个节点被称作ZNodes。



ZooKeeper提供基本的API来操纵和控制Znodes，包括对节点的创建，删除，设置和获取数据，获得子节点等。

除了这些基本的操作，ZooKeeper还提供了一些配方（Recipe），其实就是一些常见的用例，例如锁，两阶段提交，领导选举等等。

ZooKeeper本身是用Java开发的，所以对Java的支持是最自然的。它同时还提供了C语言的绑定。

Kazoo是一个非常成熟的Zookeeper Python客户端，我们这就看看如果使用Python来调用ZooKeeper。（注意，运行以下的例子，需要在本地启动ZooKeeper的服务）

基本操作

以下的例子现实了对Znode的基本操作，首先要创建一个客户端的连接，并启动客户端。然后我们可以利用该客户端对Znode做增删改，取内容的操作。最后推出客户端。

```
from kazoo.client import KazooClient

import logging
logging.basicConfig()

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()

# Ensure a path, create if necessary
zk.ensure_path("/test/zk1")

# Create a node with data
zk.create("/test/zk1/node", b"a test value")

# Determine if a node exists
if zk.exists("/test/zk1"):
    print "the node exist"

# Print the version of a node and its data
data, stat = zk.get("/test/zk1")
print("Version: %s, data: %s" % (stat.version, data.decode("utf-8")))

# List the children
children = zk.get_children("/test/zk1")
print("There are %s children with names %s" % (len(children), children))

zk.stop()
```

通过对ZNode的操作，我们可以完成一些分布式服务协调的基本需求，包括名字服务，配置服务，分组等等。

故障检测(Failure Detection)

在分布式系统中，一个最基本的需求就是当某一个服务出问题的时候，能够通知其它的节点或者某个管理节点。

ZooKeeper提供ephemeral Node的概念，当创建该Node的服务退出或者异常中止的时候，该Node会被删除，所以我们就可以利用这种行为来监控服务运行状态。

以下是worker的代码

```
from kazoo.client import KazooClient
import time

import logging
logging.basicConfig()

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()

# Ensure a path, create if necessary
zk.ensure_path("/test/failure_detection")

# Create a node with data
zk.create("/test/failure_detection/worker",
          value=b"a test value", ephemeral=True)

while True:
    print "I am alive!"
    time.sleep(3)

zk.stop()
```

以下的monitor 代码，监控worker服务是否运行。

```
from kazoo.client import KazooClient

import time

import logging
logging.basicConfig()

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()

# Determine if a node exists
while True:
    if zk.exists("/test/failure_detection/worker"):
        print "the worker is alive!"
    else:
        print "the worker is dead!"
        break
    time.sleep(3)

zk.stop()
```

领导选举

Kazoo直接提供了领导选举的API，使用起来非常方便。

```
from kazoo.client import KazooClient
import time
import uuid

import logging
logging.basicConfig()

my_id = uuid.uuid4()

def leader_func():
    print "I am the leader {}".format(str(my_id))
    while True:
        print "{} is working! ".format(str(my_id))
        time.sleep(3)

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()

election = zk.Election("/electionpath")

# blocks until the election is won, then calls
# leader_func()
election.run(leader_func)

zk.stop()
```

你可以同时运行多个worker，其中一个会获得Leader，当你杀死当前的leader后，会有一个新的leader被选出。

分布式锁

锁的概念大家都熟悉，当我们希望某一件事在同一时间只有一个服务在做，或者某一个资源在同一时间只有一个服务能访问，这个时候，我们就需要用到锁。

```
from kazoo.client import KazooClient
import time
import uuid

import logging
logging.basicConfig()

my_id = uuid.uuid4()
```

```
def work():
    print "{} is working! ".format(str(my_id))

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()

lock = zk.Lock("/lockpath", str(my_id))

print "I am {}".format(str(my_id))

while True:
    with lock:
        work()
    time.sleep(3)

zk.stop()
```

当你运行多个worker的时候，不同的worker会试图获取同一个锁，然而只有一个worker会工作，其它的worker必须等待获得锁后才能执行。

监视

ZooKeeper提供了监视（Watch）的功能，当节点的数据被修改的时候，监控的function会被调用。我们可以利用这一点进行配置文件的同步，发消息，或其他需要通知的功能。

```
from kazoo.client import KazooClient
import time

import logging
logging.basicConfig()

zk = KazooClient(hosts='127.0.0.1:2181')
zk.start()

@zk.DataWatch('/path/to/watch')
def my_func(data, stat):
    if data:
        print "Data is %s" % data
        print "Version is %s" % stat.version
    else :
        print "data is not available"

while True:
    time.sleep(10)

zk.stop()
```

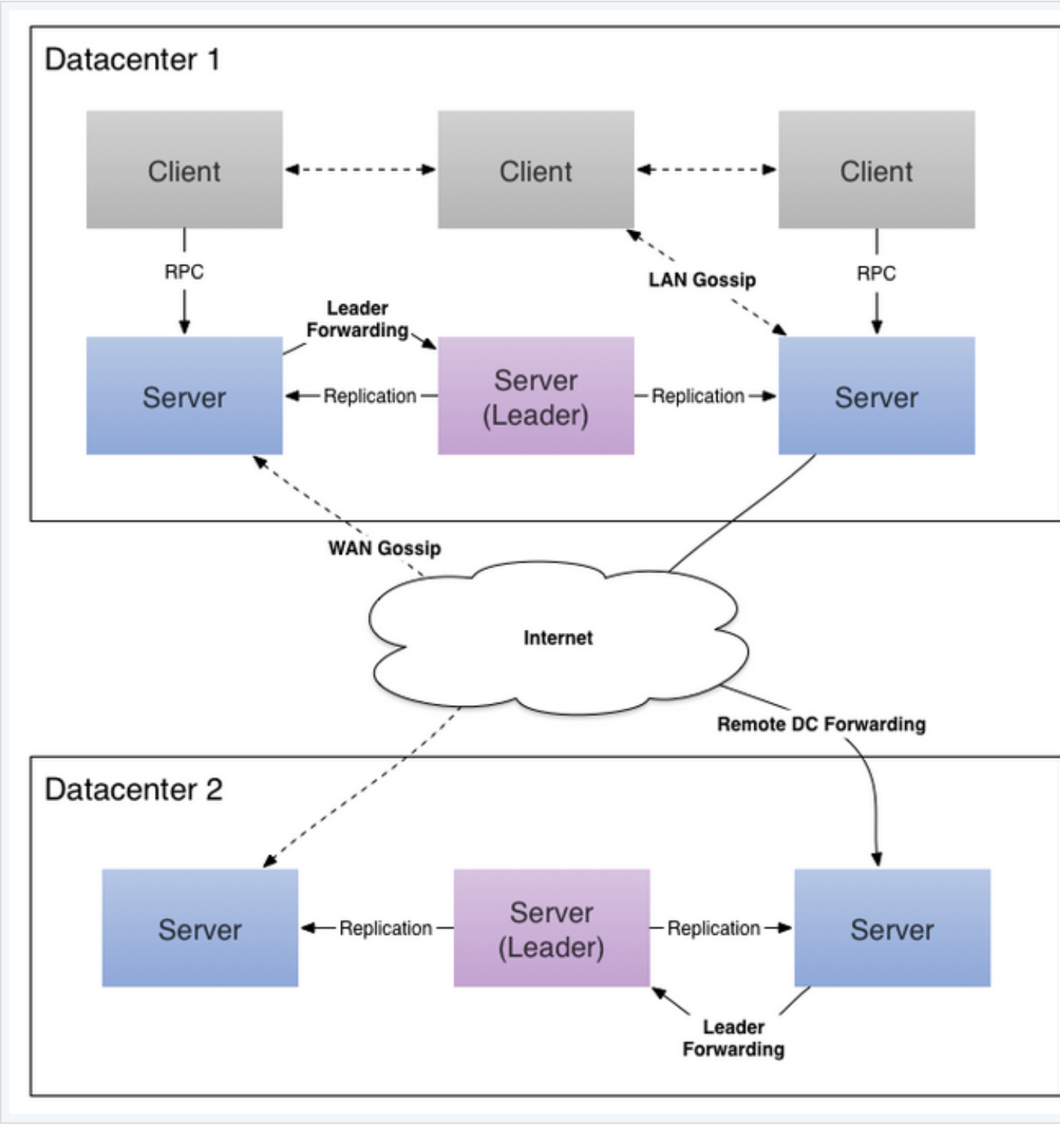
除了我们上面列举的内容外，Kazoo还提供了许多其他的功能，例如：计数，租约，队列等等，大家有兴趣可以参考它的文档

Consul

Consul是用Go开发的分布式服务协调管理的工具，它提供了服务发现，健康检查，Key / Value存储等功能，并且支持跨数据中心的功能。

Consul提供ZooKeeper类似的功能，它的基于HTTP的API可以方便的和各种语言进行绑定。自然Python也在列。

与Zookeeper有所差异的是Consul通过基于Client / Server架构的Agent部署来支持跨Data Center的功能。



Consul在Cluster伤的每一个节点都运行一个Agent，这个Agent可以使Server或者Client模式。Client负责到Server的高效通信，相对为无状态的。 Server负责包括选举领导节点，维护cluster的状态，对所有的查询做响应，跨数据中心的通信等等。

KV基本操作

类似于Zookeeper，Consul支持对KV的增删查改的操作。

```
import consul

c = consul.Consul()

# set data for key foo
c.kv.put('foo', 'bar')

# poll a key for updates
index = None
while True:
```

```
    index, data = c.kv.get('foo', index=index)
    print data['Value']

c.kv.delete('foo')
```

这里和ZooKeeper对Znode的操作几乎是一样的。

服务发现（Service Discovery）和健康检查（Health Check）

Consul的另一个主要的功能是用于对分布式的服务做管理，用户可以注册一个服务，同时还提供对服务做健康检测的功能。

首先，用户需要定义一个服务。

```
{
  "service": {
    "name": "redis",
    "tags": ["master"],
    "address": "127.0.0.1",
    "port": 8000,
    "checks": [
      {
        "script": "/usr/local/bin/check_redis.py",
        "interval": "10s"
      }
    ]
  }}
```

其中，服务的名字是必须的，其它的字段可以自选，包括了服务的地址，端口，相应的健康检查的脚本。当用户注册了一个服务后，就可以通过Consul来查询该服务，获得该服务的状态。

Consul支持三种Check的模式：

- 调用一个外部脚本（Script），在该模式下，consul定时会调用一个外部脚本，通过脚本的返回内容获得对应服务的健康状态。
- 调用HTTP，在该模式下，consul定时会调用一个HTTP请求，返回2XX，则为健康；429 （Too many request）是警告。其它均为不健康
- 主动上报，在该模式下，服务需要主动调用一个consul提供的HTTP PUT请求，上报健康状态。

Python API提供对应的接口，大家可以参考 http://python-consul.readthedocs.org/en/latest/

- Consul.Agent.Service
- Consul.Agent.Check

Consul的Health Check和Zookeeper的Failure Detection略有不同，ZooKeeper可以利用ephemeral Node来检测服务的状态，Consul的Health Check，通过调用脚本，HTTP或者主动上报的方式检查服务的状态，更为灵活，可以获得等多的信息，但是也需要做更多的工作。

故障检测(Failure Detection)

Consul提供Session的概念，利用Session可以检查服务是否存活。

对每一个服务我们都可以创建一个session对象，注意这里我们设置了ttl，consul会以ttl的数值为间隔时间，持续的对session的存活做检查。对应的在服务中，我们需要持续的renew session，保证session是合法的。

```
import consul
import time

```

```
c = consul.Consul()

s = c.session.create(name="worker",behavior='delete',ttl=10)

print "session id is {}".format(s)

while True:
    c.session.renew(s)
    print "I am alive ..."
    time.sleep(3)
```

Moniter代码用于监控worker相关联的session的状态，但发现worker session已经不存在了，就做出响应的处理。

```
import consul
import time

def is_session_exist(name, sessions):
    for s in sessions:
        if s['Name'] == name:
            return True

    return False

c = consul.Consul()

while True:
    index, sessions = c.session.list()
    if is_session_exist('worker', sessions):
        print "worker is alive ..."
    else:
        print 'worker is dead!'
        break
    time.sleep(3)
```

这里注意，因为是基于ttl（最小10秒）的检测，从业务中断到被检测到，至少有10秒的时延，对应需要实时响应的情景，并不适用。Zookeeper使用ephemeral Node的方式时延相对短一点，但也非实时。

领导选举和分布式的锁

无论是Consul本身还是Python客户端，都不直接提供Leader Election的功能，但是这篇文档介绍了如何利用Consul的KV存储来实现Leader Election,利用Consul的KV功能，可以很方便的实现领导选举和锁的功能。

当对某一个Key做put操作的时候，可以创建一个session对象，设置一个acquire标志为该 session，这样就获得了一个锁，获得所得客户则是被选举的leader。

代码如下：

```
import consul
import time

c = consul.Consul()

def request_lead(namespace, session_id):
    lock = c.kv.put(leader_namespace,"leader check", acquire=session_id)
    return lock

def release_lead(session_id):
    c.session.destroy(session_id)

def whois_lead(namespace):
    index,value = c.kv.get(namespace)
    session = value.get('Session')
    if session is None:
        print 'No one is leading, maybe in electing'
    else:
        index, value = c.session.info(session)
```

```
        print '{} is leading'.format(value['ID'])

def work_non_block():
    print "working"

def work_block():
    while True:
        print "working"
        time.sleep(3)

leader_namespace = 'leader/test'

## initialize leader key/value node
leader_index, leader_node = c.kv.get(leader_namespace)

if leader_node is None:
    c.kv.put(leader_namespace,"a leader test")

while True:
    whois_lead(leader_namespace)
    session_id = c.session.create(ttl=10)
    if request_lead(leader_namespace,session_id):
        print "I am now the leader"
        work_block()
        release_lead(session_id)
    else:
        print "wait leader elected!"
    time.sleep(3)
```

利用同样的机制，可以方便的实现锁，信号量等分布式的同步操作。

监视

Consul的Agent提供了Watch的功能，然而Python客户端并没有相应的接口。

etcd

etcd是另一个用GO开发的分布式协调应用，它提供一个分布式的Key / Value存储来进行共享的配置管理和服务发现。

同样的etcd使用基于HTTP的API，可以灵活的进行不同语言的绑定，我们用的是这个客户端 https://github.com/jplana/python-etcd

基本操作

```
import etcd

client = etcd.Client()
client.write('/nodes/n1', 1)
print client.read('/nodes/n1').value
```

etcd对节点的操作和ZooKeeper类似，不过etcd不支持ZooKeeper的ephemeral Node的概念，要监控服务的状态似乎比较麻烦。

分布式锁

etcd支持分布式锁，以下是一个例子。

```
import sys

sys.path.append("../../")

import etcd

```

```
import uuid
import time

my_id = uuid.uuid4()

def work():
    print "I get the lock {}".format(str(my_id))

client = etcd.Client()

lock = etcd.Lock(client, '/customerlock', ttl=60)

with lock as my_lock:
    work()
    lock.is_locked()  # True
    lock.renew(60)
lock.is_locked()  # False
```

老版本的etcd支持leader election，但是在最新版该功能被deprecated了，参见 https://coreos.com/etcd/docs/0.4.7/etcd-modules/

其它

我们针对分布式协调的功能讨论了三个不同的开源应用，其实还有许多其它的选择，我这里就不一一介绍，大家有兴趣可以访问以下的链接：

- eureka https://github.com/Netflix/eureka

  Netflix开发的定位服务，应用于fail over和load balance的功能

- curator http://curator.apache.org/

  基于ZooKeeper的更高层次的封装

- doozerd https://github.com/ha/doozerd

  基于GO的高可靠，分布式的数据存储，过去两年已经不活跃

- openreplica http://openreplica.org/

  基于Python开发的，面向对象的接口的分布式应用协调的工具

- serf http://www.serfdom.io/

  serf提供轻量级的cluster成员管理，故障检测（failure detection）和协调。开发基于GO语言。Consul使用了serf提供的功能

- noah https://github.com/lusis/Noah

  基于ruby的ZooKeeper实现，过去三年不活跃

- copy cat https://github.com/kuujo/copycat

  基于日志的分布式协调的框架，使用Java开发

总结

ZooKeeper无疑是分布式协调应用的最佳选择，功能全，社区活跃，用户群体很大，对所有典型的用例都有很好的封装，支持不同语言的绑定。缺点是，整个应用比较重，依赖于Java，不支持跨数据中心。

Consul作为使用Go语言开发的分布式协调，对业务发现的管理提供很好的支持，他的HTTP API也能很好的和不同的语言绑定，并支持跨数据中心的应用。缺点是相对较新，适合喜欢尝试新事物的用户。

etcd是一个更轻量级的分布式协调的应用，提供了基本的功能，更适合一些轻量级的应用来使用。

参考

如果大家对于分布式系统的协调想要进行更多的了解，可以阅读一下的链接：

http://stackoverflow.com/questions/6047917/zookeeper-alternatives-cluster-coordination-service

http://txt.fliglio.com/2014/05/encapsulated-services-with-consul-and-confd/

http://txt.fliglio.com/2013/12/service-discovery-with-docker-docker-links-and-beyond/

http://www.serfdom.io/intro/vs-zookeeper.html

http://devo.ps/blog/zookeeper-vs-doozer-vs-etcd/

https://www.digitalocean.com/community/articles/how-to-set-up-a-serf-cluster-on-several-ubuntu-vps

http://www.slideshare.net/JyrkiPulliainen/taming-pythons-with-zoo-keeper-ep2013?qid=e1267f58-090d-4147-9909-ec673525e76b&v=qf1&b=&from_search=8

http://muratbuffalo.blogspot.com/2014/09/paper-summary-tango-distributed-data.html

https://developer.yahoo.com/blogs/hadoop/apache-zookeeper-making-417.html

http://www.knewton.com/tech/blog/2014/12/eureka-shouldnt-use-zookeeper-service-discovery/

http://codahale.com/you-cant-sacrifice-partition-tolerance/