我们在上一课时了解了利用代理可以解决目标网站封 IP 的问题，但是如何实时高效地获取到大量可用的代理又是一个问题。

首先在互联网上有大量公开的免费代理，当然我们也可以购买付费的代理 IP，但是代理不论是免费的还是付费的，都不能保证是可用的，因为可能此 IP 已被其他人使用来爬取同样的目标站点而被封禁，或者代理服务器突然发生故障或网络繁忙。一旦我们选用了一个不可用的代理，这势必会影响爬虫的工作效率。

所以，我们需要提前做筛选，将不可用的代理剔除掉，保留可用代理。那么这个怎么来实现呢？这里就需要借助于一个叫作代理池的东西了。

接下来本课时我们就介绍一下如何搭建一个高效易用的代理池。

准备工作

在这里代理池的存储我们需要借助于 Redis，因此这个需要额外安装。总体来说，本课时需要的环境如下：

- 安装并成功运行和连接一个 Redis 数据库，安装方法见：https://cuiqingcai.com/5219.html。
- 安装好 Python3（至少为 Python 3.6 版本），并能成功运行 Python 程序。

安装好一些必要的库，包括 aiohttp、requests、redis-py、pyquery、Flask 等。

建议使用 Python 虚拟环境安装，参考安装命令如下：

- pip3 install -r https://raw.githubusercontent.com/Python3WebSpider/ProxyPool/master/requirements.txt

做好了如上准备工作，我们便可以开始实现或运行本课时所讲的代理池了。

代理池的目标

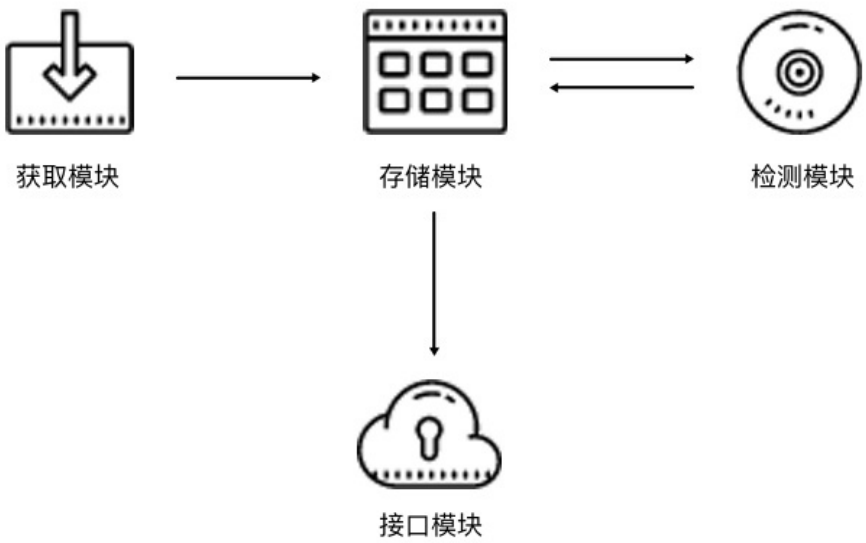
我们需要做到下面的几个目标，来实现易用高效的代理池。

- 基本模块分为 4 块：存储模块、获取模块、检测模块、接口模块。
- 存储模块：负责存储抓取下来的代理。首先要保证代理不重复，要标识代理的可用情况，还要动态实时处理每个代理，所以一种比较高效和方便的存储方式就是使用 Redis 的 Sorted Set，即有序集合。
- 获取模块：需要定时在各大代理网站抓取代理。代理可以是免费公开代理也可以是付费代理，代理的形式都是 IP 加端口，此模块尽量从不同来源获取，尽量抓取高匿代理，抓取成功之后将可用代理保存到数据库中。
- 检测模块：需要定时检测数据库中的代理。这里需要设置一个检测链接，最好是爬取哪个网站就检测哪个网站，这样更加有针对性，如果要做一个通用型的代理，那可以设置百度等链接来检测。另外，我们需要标识每一个代理的状态，如设置分数标识，100 分代表可用，分数越少代表越不可用。检测一次，如果代理可用，我们可以将分数标识立即设置为 100 满分，也可以在原基础上加 1 分；如果代理不可用，可以将分数标识减 1 分，当分数减到一定阈值后，代理就直接从数据库移除。通过这样的标识分数，我们就可以辨别代理的可用情况，选用的时候会更有针对性。
- 接口模块：需要用 API 来提供对外服务的接口。其实我们可以直接连接数据库来获取对应的数据，但是这样就需要知道数据库的连接信息，并且要配置连接，而比较安全和方便的方式就是提供一个 Web API 接口，我们通过访问接口即可拿到可用代理。另外，由于可用代理可能有多个，那么我们可以设置一个随机返回某个可用代理的接口，这样就能保证每个可用代理都可以取到，实现负载均衡。

以上内容是设计代理的一些基本思路。接下来我们设计整体的架构，然后用代码实现代理池。

代理池的架构

根据上文的描述，代理池的架构如图所示。



代理池分为 4 个模块：存储模块、获取模块、检测模块、接口模块。

- 存储模块使用 Redis 的有序集合，用来做代理的去重和状态标识，同时它也是中心模块和基础模块，将其他模块串联起来。
- 获取模块定时从代理网站获取代理，将获取的代理传递给存储模块，并保存到数据库。
- 检测模块定时通过存储模块获取所有代理，并对代理进行检测，根据不同的检测结果对代理设置不同的标识。
- 接口模块通过 Web API 提供服务接口，接口通过连接数据库并通过 Web 形式返回可用的代理。

代理池的实现

接下来我们分别用代码来实现一下这四个模块。

> 注：完整的代理池代码量较大，因此本课时的代码不必一步步跟着编写，最后去了解源码即可。

存储模块

这里我们使用 Redis 的有序集合，集合的每一个元素都是不重复的，对于代理池来说，集合的元素就变成了一个个代理，也就是 IP 加端口的形式，如 60.207.237.111:8888，这样的一个代理就是集合的一个元素。另外，有序集合的每一个元素都有一个分数字段，分数是可以重复的，可以是浮点数类型，也可以是整数类型。该集合会根据每一个元素的分数对集合进行排序，数值小的排在前面，数值大的排在后面，这样就可以实现集合元素的排序了。

对于代理池来说，这个分数可以作为判断一个代理是否可用的标志，100 为最高分，代表最可用，0 为最低分，代表最不可用。如果要获取可用代理，可以从代理池中随机获取分数最高的代理，注意是随机，这样可以保证每个可用代理都会被调用到。

分数是我们判断代理稳定性的重要标准，设置分数规则如下所示。

- 分数 100 为可用，检测器会定时循环检测每个代理可用情况，一旦检测到有可用的代理就立即置为 100，检测到不可用就将分数减 1，分数减至 0 后代理移除。
- 新获取的代理的分数为 10，如果测试可行，分数立即置为 100，不可行则分数减 1，分数减至 0 后代理移除。

这只是一种解决方案，当然可能还有更合理的方案。之所以设置此方案有如下几个原因。

- 在检测到代理可用时，分数立即置为 100，这样可以保证所有可用代理有更大的机会被获取到。你可能会问，为什么不将分数加 1 而是直接设为最高 100 呢？设想一下，有的代理是从各大免费公开代理网站获取的，常常一个代理并没有那么稳定，平均 5 次请求可能有两次成功，3 次失败，如果按照这种方式来设置分数，那么这个代理几乎不可能达到一个高的分数，也就是说即便它有时是可用的，但是筛选的分数最高，那这样的代理几乎不可能被取到。如果想追求代理稳定性，可以用上述方法，这种方法可确保分数最高的代理一定是最稳定可用的。所以，这里我们采取 “可用即设置 100” 的方法，确保只要可用的代理都可以被获取到。
- 在检测到代理不可用时，分数减 1，分数减至 0 后，代理移除。这样一个有效代理如果要被移除需要连续不断失败 100 次，也就是说当一个可用代理如果尝试了 100 次都失败了，就一直减分直到移除，一旦成功就重新置回 100。尝试机会越多，则这个代理拯救回来的机会越多，这样就不容易将曾经的一个可用代理丢弃，因为代理不可用的原因很可能是网络繁忙或者其他人用此代理请求太过频繁，所以在这里将分数为 100。
- 新获取的代理的分数设置为 10，代理如果不可用，分数就减 1，分数减到 0，代理就移除，如果代理可用，分数就置为 100。由于很多代理是从免费网站获取的，所以新获取的代理无效的比例非常高，可能可用的代理不足 10%。所以在这里我们将分数设置为 10，检测的机会没有可用代理的 100 次那么多，这也可以适当减少开销。

上述代理分数的设置思路不一定是最优思路，但据个人实测，它的实用性还是比较强的。

在这里首先给出存储模块的实现代码，见：https://github.com/Python3WebSpider/ProxyPool/tree/master/proxypool/storages，建议直接对照源码阅读。

在代码中，我们定义了一个类来操作数据库的有序集合，定义一些方法来实现分数的设置、代理的获取等。其核心实现代码实现如下所示：

```
import redis
from proxypool.exceptions import PoolEmptyException
from proxypool.schemas.proxy import Proxy
from proxypool.setting import REDIS_HOST, REDIS_PORT, REDIS_PASSWORD, REDIS_KEY, PROXY_SCORE_MAX, PROXY_SCORE_MIN, \
    PROXY_SCORE_INIT
from random import choice
from typing import List
from loguru import logger
from proxypool.utils.proxy import is_valid_proxy, convert_proxy_or_proxies
REDIS_CLIENT_VERSION = redis.__version__
IS_REDIS_VERSION_2 = REDIS_CLIENT_VERSION.startswith('2.')
class RedisClient(object):
    """
    redis connection client of proxypool
    """

    def __init__(self, host=REDIS_HOST, port=REDIS_PORT, password=REDIS_PASSWORD, **kwargs):
        """
        init redis client
        :param host: redis host
        :param port: redis port
        :param password: redis password
        """
        self.db = redis.StrictRedis(host=host, port=port, password=password, decode_responses=True, **kwargs)
```

```
def add(self, proxy: Proxy, score=PROXY_SCORE_INIT) -> int:
    """
    add proxy and set it to init score
    :param proxy: proxy, ip:port, like 8.8.8.8:88
    :param score: int score
    :return: result
    """
    if not is_valid_proxy(f'{proxy.host}:{proxy.port}'):
        logger.info(f'invalid proxy {proxy}, throw it')
        return
    if not self.exists(proxy):
        if IS_REDIS_VERSION_2:
            return self.db.zadd(REDIS_KEY, score, proxy.string())
        return self.db.zadd(REDIS_KEY, {proxy.string(): score})

def random(self) -> Proxy:
    """
    get random proxy
    firstly try to get proxy with max score
    if not exists, try to get proxy by rank
    if not exists, raise error
    :return: proxy, like 8.8.8.8:8
    """
    # try to get proxy with max score
    proxies = self.db.zrangebyscore(REDIS_KEY, PROXY_SCORE_MAX, PROXY_SCORE_MAX)
    if len(proxies):
        return convert_proxy_or_proxies(choice(proxies))
    # else get proxy by rank
    proxies = self.db.zrevrange(REDIS_KEY, PROXY_SCORE_MIN, PROXY_SCORE_MAX)
    if len(proxies):
        return convert_proxy_or_proxies(choice(proxies))
    # else raise error
    raise PoolEmptyException

def decrease(self, proxy: Proxy) -> int:
    """
    decrease score of proxy, if small than PROXY_SCORE_MIN, delete it
    :param proxy: proxy
    :return: new score
    """
    score = self.db.zscore(REDIS_KEY, proxy.string())
    # current score is larger than PROXY_SCORE_MIN
    if score and score > PROXY_SCORE_MIN:
        logger.info(f'{proxy.string()} current score {score}, decrease 1')
        if IS_REDIS_VERSION_2:
            return self.db.zincrby(REDIS_KEY, proxy.string(), -1)
        return self.db.zincrby(REDIS_KEY, -1, proxy.string())
    # otherwise delete proxy
    else:
        logger.info(f'{proxy.string()} current score {score}, remove')
        return self.db.zrem(REDIS_KEY, proxy.string())

def exists(self, proxy: Proxy) -> bool:
    """
    if proxy exists
    :param proxy: proxy
    :return: if exists, bool
    """
    return not self.db.zscore(REDIS_KEY, proxy.string()) is None

def max(self, proxy: Proxy) -> int:
    """
    set proxy to max score
    :param proxy: proxy
    :return: new score
    """
    logger.info(f'{proxy.string()} is valid, set to {PROXY_SCORE_MAX}')
    if IS_REDIS_VERSION_2:
        return self.db.zadd(REDIS_KEY, PROXY_SCORE_MAX, proxy.string())
    return self.db.zadd(REDIS_KEY, {proxy.string(): PROXY_SCORE_MAX})

def count(self) -> int:
    """
    get count of proxies
    :return: count, int
    """
    return self.db.zcard(REDIS_KEY)

def all(self) -> List[Proxy]:
    """
    get all proxies
    :return: list of proxies
    """
```

```
        return convert_proxy_or_proxies(self.db.zrangebyscore(REDIS_KEY, PROXY_SCORE_MIN, PROXY_SCORE_MAX))

    def batch(self, start, end) -> List[Proxy]:
        """
        get batch of proxies
        :param start: start index
        :param end: end index
        :return: list of proxies
        """
        return convert_proxy_or_proxies(self.db.zrevrange(REDIS_KEY, start, end - 1))
if __name__ == '__main__':
    conn = RedisClient()
    result = conn.random()
    print(result)
```

首先我们定义了一些常量，如 PROXY_SCORE_MAX、PROXY_SCORE_MIN、PROXY_SCORE_INIT 分别代表最大分数、最小分数、初始分数。REDIS_HOST、REDIS_PORT、REDIS_PASSWORD 分别代表了 Redis 的连接信息，即地址、端口、密码。REDIS_KEY 是有序集合的键名，我们可以通过它来获取代理存储所使用的有序集合。

RedisClient 这个类可以用来操作 Redis 的有序集合，其中定义了一些方法来对集合中的元素进行处理，它的主要功能如下所示。

- __init__ 方法是初始化的方法，其参数是 Redis 的连接信息，默认的连接信息已经定义为常量，在 __init__ 方法中初始化了一个 StrictRedis 的类，建立 Redis 连接。
- add 方法向数据库添加代理并设置分数，默认的分数是 PROXY_SCORE_INIT 也就是 10，返回结果是添加的结果。
- random 方法是随机获取代理的方法，首先获取 100 分的代理，然后随机选择一个返回。如果不存在 100 分的代理，则此方法按照排名来获取，选取前 100 名，然后随机选择一个返回，否则抛出异常。
- decrease 方法是在代理检测无效的时候设置分数减 1 的方法，代理传入后，此方法将代理的分数减 1，如果分数达到最低值，那么代理就删除。
- exists 方法可判断代理是否存在集合中。
- max 方法将代理的分数设置为 PROXY_SCORE_MAX，即 100，也就是当代理有效时的设置。
- count 方法返回当前集合的元素个数。
- all 方法返回所有的代理列表，供检测使用。

定义好了这些方法，我们可以在后续的模块中调用此类来连接和操作数据库。如想要获取随机可用的代理，只需要调用 random 方法即可，得到的就是随机的可用代理。

获取模块

获取模块主要是为了从各大网站抓取代理并调用存储模块进行保存，代码实现见：https://github.com/Python3WebSpider/ProxyPool/tree/master/proxypool/crawlers。

获取模块的逻辑相对简单，比如我们可以定义一些抓取代理的方法，示例如下：

```
from proxypool.crawlers.base import BaseCrawler
from proxypool.schemas.proxy import Proxy
import re
MAX_PAGE = 5
BASE_URL = 'http://www.ip3366.net/free/?stype=1&page={page}'
class IP3366Crawler(BaseCrawler):
    """
    ip3366 crawler, http://www.ip3366.net/
    """
    urls = [BASE_URL.format(page=i) for i in range(1, 8)]

    def parse(self, html):
        """
        parse html file to get proxies
        :return:
        """
        ip_address = re.compile('<tr>\s*<td>(.*?)</td>\s*<td>(.*?)</td>')
        # \s * 匹配空格，起到换行作用
        re_ip_address = ip_address.findall(html)
        for address, port in re_ip_address:
            proxy = Proxy(host=address.strip(), port=int(port.strip()))
            yield proxy
```

我们在这里定义了一个代理 Crawler 类，用来抓取某一网站的代理，这里是抓取的 IP3366 的公开代理，通过 parse 方法来解析页面的源码并构造一个个 Proxy 对象返回即可。

另外在其父类 BaseCrawler 里面定义了通用的页面抓取方法，它可以读取子类里面定义的 urls 全局变量并进行爬取，然后调用子类的 parse 方法来解析页面，代码实现如下：

```
from retrying import retry
import requests
from loguru import logger
class BaseCrawler(object):
    urls = []
```

```
    @retry(stop_max_attempt_number=3, retry_on_result=lambda x: x is None)
    def fetch(self, url, **kwargs):
        try:
            response = requests.get(url, **kwargs)
            if response.status_code == 200:
                return response.text
        except requests.ConnectionError:
            return

    @logger.catch
    def crawl(self):
        """
        crawl main method
        """
        for url in self.urls:
            logger.info(f'fetching {url}')
            html = self.fetch(url)
            for proxy in self.parse(html):
                logger.info(f'fetched proxy {proxy.string()} from {url}')
                yield proxy
```

所以，我们如果要扩展一个代理的 Crawler，只需要继承 BaseCrawler 并实现 parse 方法即可，扩展性较好。

因此，这一个个的 Crawler 就可以针对各个不同的代理网站进行代理的抓取。最后有一个统一的方法将 Crawler 汇总起来，遍历调用即可。

如何汇总呢？在这里我们可以检测代码只要定义有 BaseCrawler 的子类就算一个有效的代理 Crawler，可以直接通过遍历 Python 文件包的方式来获取，代码实现如下：

```
import pkgutil
from .base import BaseCrawler
import inspect
# load classes subclass of BaseCrawler
classes = []
for loader, name, is_pkg in pkgutil.walk_packages(__path__):
    module = loader.find_module(name).load_module(name)
    for name, value in inspect.getmembers(module):
        globals()[name] = value
        if inspect.isclass(value) and issubclass(value, BaseCrawler) and value is not BaseCrawler:
            classes.append(value)
__all__ = __ALL__ = classes
```

在这里我们调用了 walk_packages 方法，遍历了整个 crawlers 模块下的类，并判断了它是 BaseCrawler 的子类，那就将其添加到结果中，并返回。

最后只要将 classes 遍历并依次实例化，调用其 crawl 方法即可完成代理的爬取和提取，代码实现见：https://github.com/Python3WebSpider/ProxyPool/blob/master/proxypool/processors/getter.py。

检测模块

我们已经成功将各个网站的代理获取下来了，现在就需要一个检测模块来对所有代理进行多轮检测。代理检测可用，分数就设置为 100，代理不可用，分数减 1，这样就可以实时改变每个代理的可用情况。如要获取有效代理只需要获取分数高的代理即可。

由于代理的数量非常多，为了提高代理的检测效率，我们在这里使用异步请求库 aiohttp 来进行检测。

requests 作为一个同步请求库，我们在发出一个请求之后，程序需要等待网页加载完成之后才能继续执行。也就是这个过程会阻塞等待响应，如果服务器响应非常慢，比如一个请求等待十几秒，那么我们使用 requests 完成一个请求就会需要十几秒的时间，程序也不会继续往下执行，而在这十几秒的时间里程序其实完全可以去做其他的事情，比如调度其他的请求或者进行网页解析等。

对于响应速度比较快的网站来说，requests 同步请求和 aiohttp 异步请求的效果差距没那么大。可对于检测代理来说，检测一个代理一般需要十多秒甚至几十秒的时间，这时候使用 aiohttp 异步请求库的优势就大大体现出来了，效率可能会提高几十倍不止。

所以，我们的代理检测使用异步请求库 aiohttp，实现示例如下所示：

```
import asyncio
import aiohttp
from loguru import logger
from proxypool.schemas import Proxy
from proxypool.storages.redis import RedisClient
from proxypool.setting import TEST_TIMEOUT, TEST_BATCH, TEST_URL, TEST_VALID_STATUS
from aiohttp import ClientProxyConnectionError, ServerDisconnectedError, ClientOSError, ClientHttpProxyError
from asyncio import TimeoutError
EXCEPTIONS = (
    ClientProxyConnectionError,
    ConnectionRefusedError,
    TimeoutError,
    ServerDisconnectedError,
    ClientOSError,
    ClientHttpProxyError
)
```

```
class Tester(object):
    """
    tester for testing proxies in queue
    """

    def __init__(self):
        """
        init redis
        """
        self.redis = RedisClient()
        self.loop = asyncio.get_event_loop()

    async def test(self, proxy: Proxy):
        """
        test single proxy
        :param proxy: Proxy object
        :return:
        """
        async with aiohttp.ClientSession(connector=aiohttp.TCPConnector(ssl=False)) as session:
            try:
                logger.debug(f'testing {proxy.string()}')
                async with session.get(TEST_URL, proxy=f'http://{proxy.string()}', timeout=TEST_TIMEOUT,
                                       allow_redirects=False) as response:
                    if response.status in TEST_VALID_STATUS:
                        self.redis.max(proxy)
                        logger.debug(f'proxy {proxy.string()} is valid, set max score')
                    else:
                        self.redis.decrease(proxy)
                        logger.debug(f'proxy {proxy.string()} is invalid, decrease score')
            except EXCEPTIONS:
                self.redis.decrease(proxy)
                logger.debug(f'proxy {proxy.string()} is invalid, decrease score')

    @logger.catch
    def run(self):
        """
        test main method
        :return:
        """
        # event loop of aiohttp
        logger.info('stating tester...')
        count = self.redis.count()
        logger.debug(f'{count} proxies to test')
        for i in range(0, count, TEST_BATCH):
            # start end end offset
            start, end = i, min(i + TEST_BATCH, count)
            logger.debug(f'testing proxies from {start} to {end} indices')
            proxies = self.redis.batch(start, end)
            tasks = [self.test(proxy) for proxy in proxies]
            # run tasks using event loop
            self.loop.run_until_complete(asyncio.wait(tasks))
if __name__ == '__main__':
    tester = Tester()
    tester.run()
```

这里定义了一个类 Tester，__init__ 方法中建立了一个 RedisClient 对象，供该对象中其他方法使用。接下来定义了一个 test 方法，这个方法用来检测单个代理的可用情况，其参数就是被检测的代理。注意，test 方法前面加了 async 关键词，代表这个方法是异步的。方法内部首先创建了 aiohttp 的 ClientSession 对象，可以直接调用该对象的 get 方法来访问页面。

测试的链接在这里定义为常量 TEST_URL。如果针对某个网站有抓取需求，建议将 TEST_URL 设置为目标网站的地址，因为在抓取的过程中，代理本身可能是可用的，但是该代理的 IP 已经被目标网站封掉了。例如，某些代理可以正常访问百度等页面，但是对知乎来说可能就被封了，所以我们可以将 TEST_URL 设置为知乎的某个页面的链接，当请求失败、代理被封时，分数自然会减下来，失效的代理就不会被取到了。

如果想做一个通用的代理池，则不需要专门设置 TEST_URL，可以将其设置为一个不会封 IP 的网站，也可以设置为百度这类响应稳定的网站。

我们还定义了 TEST_VALID_STATUS 变量，这个变量是一个列表形式，包含了正常的状态码，如可以定义成 [200]。当然某些目标网站可能会出现其他的状态码，你可以自行配置。

程序在获取 Response 后需要判断响应的状态，如果状态码在 TEST_VALID_STATUS 列表里，则代表代理可用，可以调用 RedisClient 的 max 方法将代理分数设为 100，否则调用 decrease 方法将代理分数减 1，如果出现异常也同样将代理分数减 1。

另外，我们设置了批量测试的最大值为 TEST_BATCH，也就是一批测试最多 TEST_BATCH 个，这可以避免代理池过大时一次性测试全部代理导致内存开销过大的问题。当然也可以用信号量来实现并发控制。

随后，在 run 方法里面获取了所有的代理列表，使用 aiohttp 分配任务，启动运行。这样在不断的运行过程中，代理池中无效的代理的分数会一直被减 1，直至被清除，有效的代理则会一直保持 100 分，供随时取用。

这样，测试模块的逻辑就完成了。

接口模块

通过上述 3 个模块，我们已经可以做到代理的获取、检测和更新，数据库就会以有序集合的形式存储各个代理及其对应的分数，分数 100 代表可用，分数越小代表越不可用。

但是我们怎样方便地获取可用代理呢？可以用 RedisClient 类直接连接 Redis，然后调用 random 方法。这样做没问题，效率很高，但是会有几个弊端。

- 如果其他人使用这个代理池，他需要知道 Redis 连接的用户名和密码信息，这样很不安全。
- 如果代理池需要部署在远程服务器上运行，而远程服务器的 Redis 只允许本地连接，那么我们就不能远程直连 Redis 来获取代理。
- 如果爬虫所在的主机没有连接 Redis 模块，或者爬虫不是由 Python 语言编写的，那么我们就无法使用 RedisClient 来获取代理。
- 如果 RedisClient 类或者数据库结构有更新，那么爬虫端必须同步这些更新，这样非常麻烦。

综上考虑，为了使代理池可以作为一个独立服务运行，我们最好增加一个接口模块，并以 Web API 的形式暴露可用代理。

这样一来，获取代理只需要请求接口即可，以上的几个缺点弊端也可以避免。

我们使用一个比较轻量级的库 Flask 来实现这个接口模块，实现示例如下所示：

```
from flask import Flask, g
from proxypool.storages.redis import RedisClient
from proxypool.setting import API_HOST, API_PORT, API_THREADED
__all__ = ['app']
app = Flask(__name__)
def get_conn():
    """
    get redis client object
    :return:
    """
    if not hasattr(g, 'redis'):
        g.redis = RedisClient()
    return g.redis
@app.route('/')
def index():
    """
    get home page, you can define your own templates
    :return:
    """
    return '<h2>Welcome to Proxy Pool System</h2>'
@app.route('/random')
def get_proxy():
    """
    get a random proxy
    :return: get a random proxy
    """
    conn = get_conn()
    return conn.random().string()
@app.route('/count')
def get_count():
    """
    get the count of proxies
    :return: count, int
    """
    conn = get_conn()
    return str(conn.count())
if __name__ == '__main__':
    app.run(host=API_HOST, port=API_PORT, threaded=API_THREADED)
```

在这里，我们声明了一个 Flask 对象，定义了 3 个接口，分别是首页、随机代理页、获取数量页。

运行之后，Flask 会启动一个 Web 服务，我们只需要访问对应的接口即可获取到可用代理。

调度模块

调度模块就是调用以上所定义的 3 个模块，将这 3 个模块通过多进程的形式运行起来，示例如下所示：

```
import time
import multiprocessing
from proxypool.processors.server import app
from proxypool.processors.getter import Getter
from proxypool.processors.tester import Tester
from proxypool.setting import CYCLE_GETTER, CYCLE_TESTER, API_HOST, API_THREADED, API_PORT, ENABLE_SERVER, \
    ENABLE_GETTER, ENABLE_TESTER, IS_WINDOWS
from loguru import logger
if IS_WINDOWS:
    multiprocessing.freeze_support()
tester_process, getter_process, server_process = None, None, None
class Scheduler():
    """
    scheduler
    """
```

```
    def run_tester(self, cycle=CYCLE_TESTER):
        """
        run tester
        """
        if not ENABLE_TESTER:
            logger.info('tester not enabled, exit')
            return
        tester = Tester()
        loop = 0
        while True:
            logger.debug(f'tester loop {loop} start...')
            tester.run()
            loop += 1
            time.sleep(cycle)

    def run_getter(self, cycle=CYCLE_GETTER):
        """
        run getter
        """
        if not ENABLE_GETTER:
            logger.info('getter not enabled, exit')
            return
        getter = Getter()
        loop = 0
        while True:
            logger.debug(f'getter loop {loop} start...')
            getter.run()
            loop += 1
            time.sleep(cycle)

    def run_server(self):
        """
        run server for api
        """
        if not ENABLE_SERVER:
            logger.info('server not enabled, exit')
            return
        app.run(host=API_HOST, port=API_PORT, threaded=API_THREADED)

    def run(self):
        global tester_process, getter_process, server_process
        try:
            logger.info('starting proxypool...')
            if ENABLE_TESTER:
                tester_process = multiprocessing.Process(target=self.run_tester)
                logger.info(f'starting tester, pid {tester_process.pid}...')
                tester_process.start()

            if ENABLE_GETTER:
                getter_process = multiprocessing.Process(target=self.run_getter)
                logger.info(f'starting getter, pid{getter_process.pid}...')
                getter_process.start()

            if ENABLE_SERVER:
                server_process = multiprocessing.Process(target=self.run_server)
                logger.info(f'starting server, pid{server_process.pid}...')
                server_process.start()

            tester_process.join()
            getter_process.join()
            server_process.join()
        except KeyboardInterrupt:
            logger.info('received keyboard interrupt signal')
            tester_process.terminate()
            getter_process.terminate()
            server_process.terminate()
        finally:
            # must call join method before calling is_alive
            tester_process.join()
            getter_process.join()
            server_process.join()
            logger.info(f'tester is {"alive" if tester_process.is_alive() else "dead"}')
            logger.info(f'getter is {"alive" if getter_process.is_alive() else "dead"}')
            logger.info(f'server is {"alive" if server_process.is_alive() else "dead"}')
            logger.info('proxy terminated')
if __name__ == '__main__':
    scheduler = Scheduler()
    scheduler.run()
```

3 个常量 ENABLE_TESTER、ENABLE_GETTER、ENABLE_SERVER 都是布尔类型，表示测试模块、获取模块、接口模块的开关，如果都为 True，则代表模块开启。

启动入口是 run 方法，这个方法分别判断 3 个模块的开关。如果开关开启，启动时程序就新建一个 Process 进程，设置好启动目标，然后调用

start 方法运行，这样 3 个进程就可以并行执行，互不干扰。

3 个调度方法结构也非常清晰。比如，run_tester 方法用来调度测试模块，首先声明一个 Tester 对象，然后进入死循环不断循环调用其 run 方法，执行完一轮之后就休眠一段时间，休眠结束之后重新再执行。在这里，休眠时间也定义为一个常量，如 20 秒，即每隔 20 秒进行一次代理检测。

最后，只需要调用 Scheduler 的 run 方法即可启动整个代理池。

以上内容便是整个代理池的架构和相应实现逻辑。

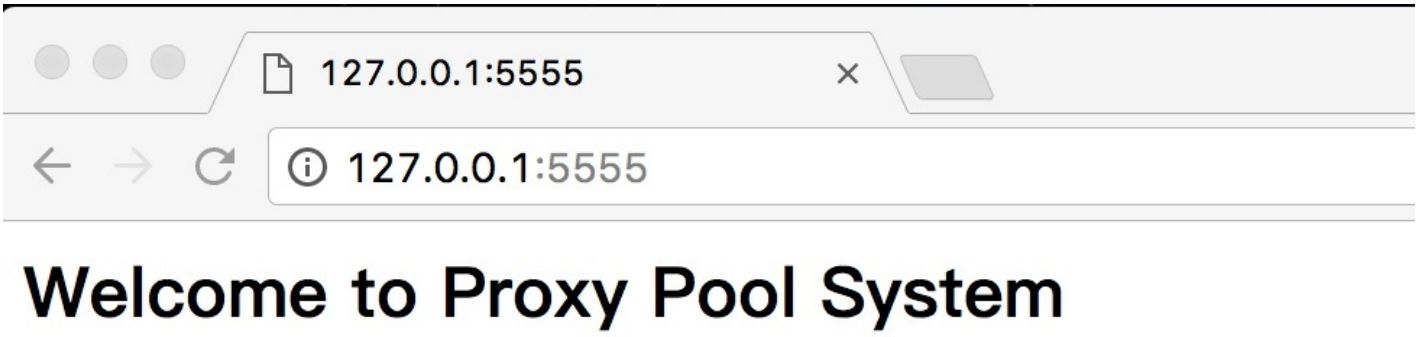
运行

接下来我们将代码整合一下，将代理运行起来，运行之后的输出结果如下所示：

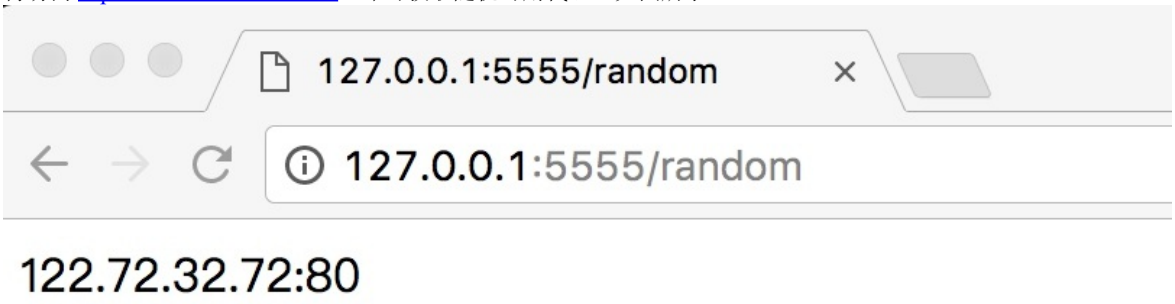
```
2020-04-13 02:52:06.510 | INFO     | proxypool.storages.redis:decrease:73 - 60.186.146.193:9000 current score 10.0, decrease 1
2020-04-13 02:52:06.517 | DEBUG    | proxypool.processors.tester:test:52 - proxy 60.186.146.193:9000 is invalid, decrease score
2020-04-13 02:52:06.524 | INFO     | proxypool.storages.redis:decrease:73 - 60.186.151.147:9000 current score 10.0, decrease 1
2020-04-13 02:52:06.532 | DEBUG    | proxypool.processors.tester:test:52 - proxy 60.186.151.147:9000 is invalid, decrease score
2020-04-13 02:52:07.159 | INFO     | proxypool.storages.redis:max:96 - 60.191.11.246:3128 is valid, set to 100
2020-04-13 02:52:07.167 | DEBUG    | proxypool.processors.tester:test:46 - proxy 60.191.11.246:3128 is valid, set max score
2020-04-13 02:52:17.271 | INFO     | proxypool.storages.redis:decrease:73 - 59.62.7.130:9000 current score 10.0, decrease 1
2020-04-13 02:52:17.280 | DEBUG    | proxypool.processors.tester:test:52 - proxy 59.62.7.130:9000 is invalid, decrease score
2020-04-13 02:52:17.288 | INFO     | proxypool.storages.redis:decrease:73 - 60.167.103.74:1133 current score 10.0, decrease 1
2020-04-13 02:52:17.295 | DEBUG    | proxypool.processors.tester:test:52 - proxy 60.167.103.74:1133 is invalid, decrease score
2020-04-13 02:52:17.302 | INFO     | proxypool.storages.redis:decrease:73 - 60.162.71.113:9000 current score 10.0, decrease 1
2020-04-13 02:52:17.309 | DEBUG    | proxypool.processors.tester:test:52 - proxy 60.162.71.113:9000 is invalid, decrease score
```

以上是代理池的控制台输出，可以看到可用代理设置为 100，不可用代理分数减 1。

接下来我们再打开浏览器，当前配置了运行在 5555 端口，所以打开：http://127.0.0.1:5555，即可看到其首页，如图所示。



再访问 http://127.0.0.1:5555/random，即可获取随机可用代理，如图所示。



我们只需要访问此接口即可获取一个随机可用代理，这非常方便。

获取代理的代码如下所示：

```
import requests

PROXY_POOL_URL = 'http://localhost:5555/random'

def get_proxy():
```

```
    try:
        response = requests.get(PROXY_POOL_URL)
        if response.status_code == 200:
            return response.text
    except ConnectionError:
        return None
```

这样便可以获取到一个随机代理了，它是字符串类型，此代理可以按照上一课时所示的方法设置，如 requests 的使用方法如下所示：

```
import requests

proxy = get_proxy()
proxies = {
    'http': 'http://' + proxy,
    'https': 'https://' + proxy,
}
try:
    response = requests.get('http://httpbin.org/get', proxies=proxies)
    print(response.text)
except requests.exceptions.ConnectionError as e:
    print('Error', e.args)
```

有了代理池之后，我们再取出代理即可有效防止 IP 被封禁的情况。

总结

本课时代码地址为：https://github.com/Python3WebSpider/ProxyPool，代码量相比之前的案例复杂了很多，逻辑也相对完善。另外代码库中还提供了 Docker 和 Kubernetes 的运行和部署操作，可以帮助我们更加快捷地运行代理池，如果你感兴趣可以了解下。