

Come implementare una cache condivisa per i feed

Al termine del Capitolo 2, una possibile miglioria proposta prevedeva di mantenere separata la scansione dei feed dalla produzione della pagina di output, registrando i risultati delle scansioni eseguite nelle diverse sessioni. Davvero, l'autore non è proprio riuscito a trattenermi dall'offrirvi questo modulo-bonus, che fa esattamente questo...

In questa appendice viene presentato il codice di una cache condivisa con i dati di flusso catturati, che potete utilizzare come sostituto di `feedparser`. Dopo aver lavorato con `HTTPCache` in tutto il libro, era piuttosto evidente che dovesse esserci qualcosa di analogo, specifico per i feed: ecco, quindi, un tentativo in questo senso.

Questo modulo condivide tra i programmi l'attività di recupero dei feed, minimizzando l'effettivo download dei feed grazie al supporto minuzioso del GET condizionale HTTP e limitando l'esecuzione dei download a non più di uno ogni ora. Ogni volta che programmi diversi cercano di catturare lo stesso feed, o quando lo stesso programma cerca di recuperare lo stesso feed più volte, questa cache serve i dati registrati localmente, finché non ne viene determinata la "scadenza": soltanto in quel momento il programma catturerà il feed aggiornato.

Il codice presentato non è assolutamente vitale per il funzionamento di alcun programma illustrato in questo libro, tuttavia potrà contribuire a migliorarne l'efficacia, soprattutto se cominciate a lavorare con un gran numero di feed. In particolare, questo potrebbe rappresentare ben presto un problema se deciderete di filtrare grandi volumi di feed con un programma come il generatore di flussi con collegamenti popolari, analizzato nel Capitolo 15. L'auto-

re, per esempio, esegue questo programma con circa 700 feed, e questa cache risulta utilissima nei casi di esecuzioni ripetute. Inoltre, apprezzerete l'utilità di questa cache lavorando al vostro aggregatore di feed, nel corso delle varie prove che eseguirete per mettere a punto i modelli.

A questo punto non rimane che iniziare con il codice: il Listato A.1 presenta la parte iniziale di questo nuovo modulo, chiamato `feedcache.py`.

Listato A.1 `feedcache.py` (1 di 10)

```
#!/usr/bin/env python
"""
feedcache

Implementa una cache condivisa per i dati di feed ottenuti tramite feedparser.
"""
import sys, os, os.path, md5, gzip, feedparser, time
import cPickle as pickle

def main():
    """
    Visualizza un feed analizzato, oppure aggiorna tutti i feed.
    """
    feed_cache = FeedCache()
    if len(sys.argv) > 1:
        # Come dimostrazione, cattura e stampa un feed analizzato.
        from pprint import pprint
        pprint(feed_cache.parse(sys.argv[1]))
    else:
        # Apre la cache dei feed e li aggiorna tutti.
        feed_cache.refreshFeeds()
```

Come molti altri moduli riutilizzabili che abbiamo visto in questo libro, il Listato A.1 inizia con una funzione `main()` che accetta un argomento opzionale da riga di comando, l'URL di un feed: disponendo di questo argomento, il programma cattura il feed e visualizza i dati analizzati sulla console.

Tuttavia, se non è specificato alcun argomento, la funzione `main()` crea un'istanza di `FeedCache` e chiama il metodo `refreshFeeds()`. L'utilità di tutto questo è evidente quando il modulo viene eseguito in un crontab o come attività pianificata, per esempio con frequenza oraria. Il metodo `refreshFeeds()` analizza tutti i feed che sono stati recuperati e messi in cache in precedenza, assicurandosi così che tutti i feed in cache siano aggiornati prima che qualsiasi programma ne richieda l'uso. Poiché questo aggiornamento pianificato ha luogo in background, mentre state eseguendo altre attività, noterete una risposta più immediata da parte dei programmi che utilizzano la cache.

Continuando con il codice, il Listato A.2 definisce una classe chiamata `FeedCacheRecord`.

Listato A.2 feedcache.py (2 di 10)

```
class FeedCacheRecord:
    """
    Registra i record nella cache del feed.
    """
    def __init__(self, last_poll=0.0, etag='', modified=None,
                 data=None):
        """Inizializza il record di cache."""
        self.last_poll = last_poll
        self.etag      = etag
        self.modified  = modified
        self.data      = data
```

La definizione della classe `FeedCacheRecord` è molto essenziale: include alcune proprietà che descrivono un feed in cache, tra le quali i dati del feed e un timestamp che indica l'ultima data/ora di aggiornamento del feed corrente, nonché la data di modifica HTTP e gli header ETag dell'ultima cattura.

Il Listato A.3 mostra l'inizio della classe `FeedCache`.

Listato A.3 feedcache.py (3 di 10)

```
class FeedCache:
    """
    Implementa una cache di dati di feed aggiornati.
    """
    CACHE_DIR      = ".feed_cache"
    REFRESH_PERIOD = 60 * 60

    def __init__(self, cache_dir=CACHE_DIR,
                 refresh_period=REFRESH_PERIOD):
        """
        Inizializza e apre la cache.
        """
        self.refresh_period = refresh_period
        self.cache_dir      = cache_dir

        # Crea la directory di cache, se non esiste.
        if not os.path.exists(cache_dir):
            os.makedirs(cache_dir)
```

Il codice della classe `FeedCache` inizia con due costanti di classe:

- `CACHE_DIR` rappresenta il nome della directory che contiene i dati di feed in cache;
- `REFRESH_PERIOD`, espresso in secondi, è il tempo minimo che deve trascorrere fra gli aggiornamenti che, in questo caso, non avverranno prima che sia passata un'ora dall'ultimo aggiornamento.

Di seguito viene definito il metodo `__init__()`, che utilizza le costanti come inizzializzatori predefiniti per le proprietà dell'oggetto. Nel caso la directory di cache non esista, verrà creata. Il Listato A.4 definisce alcuni altri metodi.

Listato A.4 `feedcache.py` (4 di 10)

```
def parse(self, feed_uri, **kw):
    """
    Emula parzialmente le API feedparser, accettando soltanto un URI.
    """
    self.refreshFeed(feed_uri)
    return self.getFeedRecord(feed_uri).data

def getFeedRecord(self, feed_uri):
    """
    Cattura un record di cache del feed, in base all'URI.
    """
    return self._loadRecord(feed_uri, None)
```

Il metodo `parse()` di questo listato viene fornito per compatibilità con il modulo `feedparser`, poiché accetta lo stesso parametro (URI del feed) e restituisce gli stessi dati di flusso di `feedparser`. Nel fare questo, il metodo `parse()` attiva un aggiornamento del flusso tramite `refreshFeed()` e restituisce i dati di feed analizzati.

Vedrete tra breve i dettagli circa il funzionamento di questo metodo; per il momento dovete sapere che, sebbene sia chiamato il metodo `refreshFeed()`, questo obbedisce alle regole definite da `REFRESH_PERIOD` e dal `GET` condizionale `HTTP`. Di conseguenza non dovete preoccuparvi di eventuali chiamate ripetute a `parse()`, che scatenino un'attività di recupero dei feed eccessiva: questo è semplicemente un mezzo per attivare automaticamente gli aggiornamenti al momento opportuno, durante l'utilizzo dei dati di feed. Il codice continua con il metodo `getFeedRecord()`, che recupera il `FeedCacheRecord` di un determinato URI di feed, tramite il metodo `_loadRecord()`. All'apparenza questo meccanismo può sembrare ridondante, ma lo vedrete meglio nel seguito della trattazione. Il Listato A.5 continua la definizione di `FeedCache`.

Listato A.5 `feedcache.py` (5 di 10)

```
def refreshFeeds(self):
    """
    Aggiorna tutti i feed in cache.
    """
    # Carica l'elenco degli URI, ottiene i feed in cache
    # e inizia l'elaborazione.
    feed_uris = self._getCachedURIs()
    for feed_uri in feed_uris:
        try:
            # Aggiorna il feed dell'URI corrente.
```

```
        self.refreshFeed(feed_uri)
    except KeyboardInterrupt:
        # Permette l'interruzione del programma tramite tastiera.
        raise
    except Exception, e:
        # Continua anche in presenza di eventuali problemi.
        pass
```

Nel Listato A.5 viene definito il metodo `refreshFeeds()` utilizzato in `main()`. Qui viene chiamato il metodo `getCachedURIs()` per recuperare un elenco di tutti gli URI di feed presenti nei record registrati nella directory di cache: per ognuno di essi, il codice prova ad aggiornare il feed ricorrendo al metodo `refreshFeed()`. Qualsiasi eccezione diversa da `KeyboardInterrupt` è ignorata: questo approccio è utile nel contesto di un'attività pianificata, quando non siete al computer per notare eventuali errori; potreste tuttavia produrre un file di log per catturare eventuali problemi che potrebbero sorgere in fase di esecuzione del modulo. Il Listato A.6 si riferisce alla prima parte della definizione di `refreshFeed()`.

Listato A.6 `feedcache.py` (6 di 10)

```
def refreshFeed(self, feed_uri):
    """
    Aggiorna un determinato feed.
    """
    # Ottiene il record del feed corrente, creandone uno, se necessario.
    feed_rec = self._loadRecord(feed_uri, FeedCacheRecord())

    # Controlla se e' tempo di aggiornare il feed.
    # DA_FARE: gestire TTL, pianificare aggiornamenti, header controllo cache.
    if (time.time() - feed_rec.last_poll) < self.refresh_period:
        return
```

Per prima cosa, il metodo `refreshFeed()` cattura il record di cache corrispondente all'URI di feed fornito, utilizzando il metodo `_loadRecord()`. Osservate come questo metodo venga chiamato con un valore predefinito equivalente a un `FeedCacheRecord` vuoto, che viene restituito se in cache non esiste un record per l'URI di feed specificato. Di seguito viene eseguito un controllo sul timestamp dell'ultimo aggiornamento del record di cache. In caso di nuovo record, questo valore sarà 0. Se il tempo trascorso dall'ultimo aggiornamento è inferiore al valore specificato dalla costante `REFRESH_PERIOD`, il codice continua senza che il metodo restituisca alcunché.

La seconda parte del metodo `refreshFeed()` viene presentata nel Listato A.7.

Listato A.7 `feedcache.py` (7 di 10)

```
else:
    # Cattura il feed utilizzando le note ETag e Last-Modified.
    feed_data = feedparser.parse(feed_uri, \
```

```

        ➔ etag=feed_rec.etag, modified=feed_rec.modified)
feed_rec.last_poll = time.time()

bozo = feed_data.get('bozo_exception', None)
if bozo is not None:
    # Solleva un'eccezione per interruzioni da tastiera durante il parsing.
    if type(bozo) is KeyboardInterrupt: raise bozo

    # Non cercare di ignorare le eccezioni: non va bene!
    # (DA_FARE: salvare come testo per troubleshooting?)
    del feed_data['bozo_exception']

# Se lo stato HTTP del feed e' 304, non ci sono state modifiche.
if feed_data.get('status', -1) != 304:
    feed_rec.etag      = feed_data.get('etag', '')
    feed_rec.modified = feed_data.get('modified', None)
    feed_rec.data     = feed_data

# Aggiorna il record di cache del feed.
self._saveRecord(feed_uri, feed_rec)

```

Se l'esecuzione del metodo `refreshFeed()` arriva al codice del Listato A.7, significa che è trascorso abbastanza tempo dall'ultimo aggiornamento del feed: pertanto, per interrogare il feed viene chiamata la funzione `feedparser.parse()`, passando il valore ETag e la data di ultima modifica registrata dagli header di risposta ricevuti dopo la precedente interrogazione. Questa gestione dello stato permette alla cache di supportare il GET condizionale HTTP, rendendo così più efficace l'aggiornamento dei feed.

In seguito, nei dati restituiti da `feedparser` vengono rilevati eventuali errori, analizzando la proprietà `bozo_exception`. Nell'implementazione corrente, in realtà, queste informazioni non servono ad altro che a sollevare errori `KeyboardException` che consentono l'interruzione manuale del programma. Potreste pensare di inserire in questo punto un meccanismo di log che segnali gli errori trovati nei feed. Notate come alla fine di questo costrutto condizionale, i dati di eccezione vengano cancellati: ciò avviene perché gli oggetti complessi che spesso appaiono nelle eccezioni di parsing non sono adatti per essere registrati nella directory di cache, come invece avviene per gli oggetti `FeedCacheRecord`.

Dopo questo rapido passaggio nella gestione degli errori, il codice controlla lo stato HTTP dell'operazione di recupero corrente: se lo stato è 304, significa che il GET condizionale HTTP ha segnalato che il feed non è cambiato. In caso contrario, si può aggiornare `FeedCacheRecord` con i nuovi valori di header e i nuovi dati catturati. Il metodo `refreshFeed()` si chiude con una chiamata a `_saveRecord()`, che registra l'oggetto `FeedCacheRecord` aggiornato. Siamo quasi arrivati alla fine della classe `FeedCache`, che continua con il Listato A.8.

Listato A.8 `feedcache.py` (8 di 10)

```

def _recordFN(self, feed_uri):
    """

```

```

    Restituisce il nome di file per un determinato URI di feed.
    """
    hash = md5.md5(feed_uri).hexdigest()
    return os.path.join(self.cache_dir, '%s' % hash)

def _getCachedURIs(self):
    """
    Ottiene un elenco di URI di feed in cache.
    """
    uris = []
    for fn in os.listdir(self.cache_dir):
        rec_fn = os.path.join(self.cache_dir, fn)
        data = pickle.load(open(rec_fn, 'rb'))
        uri = data['data'].get('url', None)
        if uri: uris.append(uri)
    return uris

```

Il primo metodo definito in questo listato è `_recordFN()`, un metodo di comodo che trasforma un URI di feed in un nome di file della directory di cache, e che sarà utilizzato per gestire gli oggetti `FeedCacheRecord`.

Il metodo successivo è `_getCachedURIs()`, una tecnica molto rapida per ottenere tutti gli URI dei feed in cache, utilizzata da `refreshFeeds()`. Il metodo `_getCachedURIs()` itera tutti i file nella directory di cache, carica i dati recuperati e cerca di estrarne gli URI di feed. Questo elenco di URI viene restituito al termine del metodo.

Il Listato A.9 completa la classe `FeedCache`.

Listato A.9 feedcache.py (9 di 10)

```

def _loadRecord(self, feed_uri, default=None):
    """
    Carica un FeedCacheRecord dal disco.
    """
    try:
        rec_fn = self._recordFN(feed_uri)
        data = pickle.load(open(rec_fn, 'rb'))
        return FeedCacheRecord(**data)
    except IOError:
        return default

def _saveRecord(self, feed_uri, record):
    """
    Registra un FeedCacheRecord su disco.
    """
    rec_fn = self._recordFN(feed_uri)
    pickle.dump(record.__dict__, open(rec_fn, 'wb'))

```

I due metodi di questo listato, `_loadRecord()` e `_saveRecord()` servono, rispettivamente, per caricare e salvare gli oggetti `FeedCacheRecord`. In questa implementazione sono utilizzati i file nella directory di cache, e il modulo `pickle` viene impiegato per convertire le strutture dati Python nei feed binari salvati in questi file.

Notate come, anziché serializzare gli oggetti `FeedCacheRecord` direttamente, questi metodi manipolino il dizionario di proprietà degli oggetti. Questa non è la soluzione ideale, ma si è rivelata utile nella fase di sviluppo, quando le modifiche eseguite nella classe `FeedCacheRecord` hanno cominciato a creare problemi con `pickle` e questa tecnica si è dimostrata adatta per ottenere un comportamento corretto.

L'altro particolare da notare a proposito di questi ultimi metodi è che offrono alcune occasioni di personalizzazione. Per esempio, supponete di voler implementare un database SQL in luogo della directory di cache. Non dovrete aver problemi a lasciare invariata la maggior parte della classe, modificando soltanto l'implementazione della gestione di `FeedCacheRecord` in una sottoclasse.

Finalmente, il Listato A.10 completa il modulo.

Listato A.10 `feedcache.py` (10 di 10)

```
def parse(feed_uri, cache=None, **kw):
    """
    Implementazione parziale delle API feedparser, che accetta soltanto un URI.
    """
    return (cache or FeedCache()).parse(feed_uri, **kw)

if __name__ == '__main__': main()
```

Questa ultima porzione di codice è una funzione a livello di modulo, l'ultimo tassello per la compatibilità di `feedparser`. Vi consente di sostituire con una chiamata a `feedcache.parse()` praticamente qualsiasi chiamata a `feedparser.parse()`, ottenendo risultati identici: l'unica differenza è che tutto risulterà “magicamente” più veloce ed efficiente. Tenete presente, tuttavia, che abbiamo scritto “praticamente qualsiasi” con riferimento all'emulazione delle API `feedparser`. In effetti, questa funzione accetta un URI, ma non un file né una stringa, come il vero modulo `feedparser`. Inoltre, `feedparser` accetta altri parametri, quali un *Etag*, una data di modifica, un *user agent* e un *referrer*. Per ragioni di compatibilità, la funzione `parse()` del Listato A.10 e il metodo `parse()` del Listato A.4 accettano entrambi tutti questi parametri, ma li ignorano in fase di esecuzione. Date un'occhiata al Listato A.11, che mostra una veloce modifica al programma `feed_reader.py` del Capitolo 2 (Listato 2.18).

Listato A.11 `cached_feed_reader.py`

```
#!/usr/bin/env python

import sys
import feedcache as feedparser

if __name__ == '__main__':
```

```

feed_uri = sys.argv[1]
feed_data = feedparser.parse(feed_uri)

print "======"
print "'%(title)r' all'URL %(link)r" % feed_data['feed']
print "======"
print

for entry in feed_data['entries']:
    print "-----"
    print "Data: %(modified)r" % entry
    print "Titolo: %(title)r" % entry
    print "Link: %(link)r" % entry

    if not entry.get('summary', '') == '':
        print
        print "%(summary)r" % entry

    print "-----"
    print

```

L'unica modifica apportata a questo programma, rispetto a quello del Capitolo 2, è la riga:

```
import feedcache as feedparser
```

Grazie a questa modifica, il modulo `feedcache` viene importato con il nome `feedparser`, consentendo a `feedcache` di “spacciarsi” per il modulo originale. Tuttavia, esattamente come il programma originale, l'esecuzione di una sessione con questo programma di test darebbe un risultato analogo al seguente:

```

# python cached_feed_reader.py http://www.boingboing.net/atom.xml

======'Boing Boing' all'URL http://www.boingboing.net/===='
===='
-----
Data: 2005-01-18T13:38:05-08:00
Titolo: Cory NPR interview audio
Link: http://www.boingboing.net/2005/01/18/cory_npr_interview_a.html
-----

-----
Data: 2005-01-18T13:12:28-08:00
Titolo: Explanation for region coded printer cartridges?
Link: http://www.boingboing.net/2005/01/18/explanation_for_regi.html
-----

```

Vi sembra familiare? Dovrebbe esserlo. A questo punto, se eseguite questo programma più volte di seguito, dovrete notare un certo miglioramento nei tempi di esecuzione, benché forse meno avvertibile di quanto dovrebbe, a causa del maggior tempo richiesto dall'interprete Python per avviare ed eseguire il programma. In effetti, i benefici della cache di flusso risulteranno molto più evidenti quando comincerete a gestire grandi volumi di flussi. Che cosa aspettate?