

Hanbit eBook

Realtime 02



대용량 서버 구축을 위한

Memcached와 Redis

강대명 지음

대용량 서버 구축을 위한

Memcached와 Redis

지은이 강대명

빅데이터와 클라우드 등에 관심이 많은 개발자입니다. 파이널데이터에서 디지털 포렌식을, NHN에서 메일 서비스를 개발하였습니다. 현재는 모든 걸 정리하고 더 나은 미래를 만들기 위해서 필리핀에서 영어를 공부하고 있습니다.

- twitter: [@charsyam](https://twitter.com/charsyam)
- blog: charsyam.wordpress.com

대용량 서버 구축을 위한 Memcached와 Redis

초판발행 2012년 7월 16일

지은이 강대명 / 펴낸이 김태현

펴낸곳 한빛미디어(주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 6월 24일 제10-1779호

ISBN 978-89-7914-942-5 15560 / 정가 9,900원

책임편집 배용석 / 기획 김창수 / 편집 이순옥

디자인 표지 여동일, 내지 스튜디오 [림]

영업 김형진, 김진불, 조유미 / 마케팅 박상용, 박주훈

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.

한빛미디어 홈페이지 www.hanb.co.kr / 이메일 ask@hanb.co.kr

Published by HANBIT Media, Inc. Printed in Korea Copyright © 2012 강대명 & HANBIT Media, Inc.

이 책의 저작권은 강대명과 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 떠내고 싶은 아이디어나 원고를 메일(ebookwriter@hanb.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

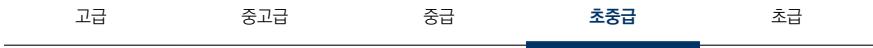
저자 서문

“아는 자는 좋아하는 자에게 미치지 못하고, 좋아하는 자는 즐기는 자에게 미치지 못한다”라는 공자님 말씀이 있습니다. 이 책을 집필하는 동안, 책을 읽는 독자가 이 기술들을 ‘아는 자’인지, ‘좋아하는 자’인지, ‘즐기는 자’인지 고민했습니다. 이제 최소한 클라우드, 빅데이터, NoSQL이라는 용어를 누구나 한 번씩은 들어보고, 또한 그 뜻을 모르면 안 될 것 같은 시기지만, 그 실체가 ‘분산 기술’이나 ‘DBMS’ 기술의 변형이라는 것을 잊어서는 안 됩니다.

분산 캐시라는 것도 마찬가지입니다. 디스크에 접근하는 것보다 메모리에 접근하는 것이 당연히 속도가 빠릅니다. 그러나 머신 하나의 메모리는 한정돼 있어서 ‘분산 기술’이 필요합니다. 우리가 아는 대부분의 NoSQL은 속도를 높이기 위해 메모리를 효과적으로 사용합니다. 결국 많은 기술이 서로서로 엮여 있는 것입니다.

저 역시, 이제 이런 기술들을 적용해보고 고민하는 사람으로서 마지막 당부의 말씀을 드린다면, 모든 문제를 해결해주는 ‘은총’은 없다는 것을 명심해야 합니다. 기술을 선택하는 사람이 “이것이 자기에게 적합한가?”, “실제로 우리 시스템에 맞는가?”를 테스트해보지 않고는 누구도 적합한 답을 찾아줄 수 없습니다. 이 책에 있는 내용도 항상 올바른 것이 아니라, 앞에 사람은 이런 고생을 해서 이렇게 됐구나 정도로 이해해주셨으면 합니다.

대상 독자



대용량 서비스를 설계하고 싶은 초보자나 이를 벗어나 실무에서 이를 어떻게 처리해야 하는지 고민하는 중급자까지가 이 책의 주요 독자입니다. 여러 내용은 알지만, 실제로 분산 캐시를 적용해보지 못한 분들에게 도움이 될 것입니다.

예제 테스트 환경

구분	내용
테스트 환경	우분투 11.10 64비트
사용 툴	파이썬 2.7.2
사용 언어	파이썬

예제 파일은 <https://github.com/charsyam/shorturl>에서 받을 수 있습니다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 염두를 못 내시는 선배, 전문가, 고수분에게는 보다 쉽게 접필하실 기회가 되리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 준비 중이며, 조만간 선보일 예정입니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정한 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나, 저자(역자)와 독자가 소통하면서 보완되고 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여, DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT기기에서 자유롭게 활용하실 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해, 독자 여러분이 언제 어디서 어떤 기기를 사용하시더라도 편리하게 전자책을 보실 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 계실 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전되지 않습니다.

차례

0 1	분산 캐시가 왜 필요할까?	1
1.1	대규모 트래픽 처리의 성능 이슈	3
1.2	서비스의 시작, 하나의 DB에서 모두 처리하기	15
1.3	Read가 많으면 Read와 Write를 분리하자	16
1.4	Write가 증가하면 파티셔닝하자	19
0 2	분산 캐시를 구현하는 핵심 기술: Consistent Hashing	23
2.1	Consistent Hashing	24
0 3	분산 캐시의 활용	37
3.1	Memcached	37
3.1.1	Memcached 설치하기	37
3.1.2	Memcached를 사용하기	38
3.1.3	Moxi Proxy를 이용해서 Memcached를 사용해보자	42
3.1.4	Memcached를 사용할 때는 항상 ExpireTime에 주의하자	45
3.1.5	Memcached를 어디에 사용할 수 있을까?	47
3.1.6	Memcached의 실행 옵션에 주의하자	49
3.1.7	Repcached를 이용한 Memcached 리플리케이션을 살펴보자	51
3.2	Redis	55
3.2.1	Redis 설치하기	56
3.2.2	Redis의 정보를 리플리케이션하자	56

0 3	Redis의 디자인 원칙과 성능 최적화	53
3.1	Redis의 디자인 원칙	53
3.2	Redis의 성능 최적화	57
3.2.1	Redis의 디스크 사용 방식	57
3.2.2	Redis의 디스크 사용 방식에 대한 주의사항	58
3.2.3	RDB와 AOF를 꼭 사용하자	57
3.2.4	Redis 서버를 데이터 유실 없이 이전해보자	60
3.2.5	비어 있는 마스터와 리플리케이션하지 않게 주의하자	61
0 4	분산 캐시 솔루션을 사용할 때 주의할 점	64
4.1	메모리 사용량에 주의하자	64
4.2	성능이 갑자기 떨어지면 메모리 스와핑을 의심하자	65
4.3	Memcached와 Redis의 flush는 어떻게 다를까?	66
0 5	Redis를 이용한 간단한 Short URL 서비스 구축하기	74
5.1	Read를 Redis로 대체하자	74
5.2	Short URL 생성 로직도 Redis로 대체하자	77
5.3	Redis의 Sorted Set을 이용한 인기 URL 찾아보기	78

1 | 분산 캐시가 왜 필요할까?

: Short URL 서비스로 알아보는 일반적인 성능 이슈들

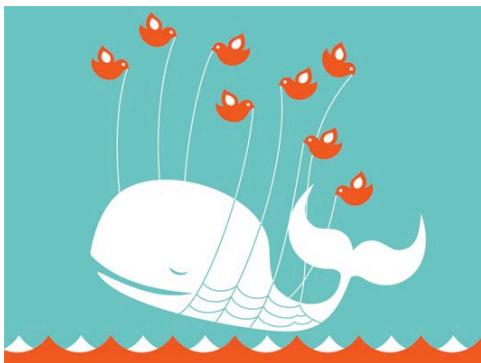
개발자는 인트라넷 같은 특정 사용자만을 위한 서비스도 개발하지만, 대부분은 구글Google, 트위터Twitter, 페이스북Facebook 같은 수억 명의 사용자가 이용하는 서비스를 꿈꾸며 개발합니다.

그림 1-1 대규모 서비스를 제공하는 구글, 트위터, 페이스북



하지만 대부분의 개발자는 대규모 서비스를 개발해본 적이 없기 때문에, 서비스가 성장함에 따라서 수많은 문제와 부딪치게 됩니다. 최근에는 발생 빈도가 좀 줄긴 했지만, 트위터만 하더라도 귀여운 고래를 자주 만나게 됩니다(트위터는 사이트 처리량을 넘어서면, 귀여운 고래 그림을 보여줍니다).

그림 1-2 트위터에서 사이트 처리량을 넘을 때 보여주는 고래



서비스를 만든다는 것은 핵심 기능을 사용자에게 전달하는 것입니다. 그래서 자신만의 핵심 기능이 중요합니다(구글이라면 검색, 트위터라면 트윗과 팔로잉 기능이 중요합니다).

그림 1-3 서비스의 핵심 기능



핵심 기능만큼 중요한 것이 안정성입니다. 구글 사이트가 허구한 날 ‘404 Not Found’ 에러⁰¹가 발생하고 검색하는 데 몇 시간씩 걸린다면 어떨까요? 트위터가 트윗 서비스는 되지 않고 고래만 며칠씩 보여준다면? 결국 사용자는 서비스를 떠나고 서비스는 사라질 것입니다.

처음부터 페이스북이나 트위터 같은 서비스를 안정적으로 만들 수는 없을까요? 페이스북은 하루에 5억 명의 사용자(또는 유저)가 로그인하여 사용합니다. 이런 서비스를 실패 없이 후다닥 만든다는 것은 전담을 타고 5분만에 전담의 OS가 성능이 좋지 않다고 새로 만들어낸 키라 야마토⁰² 정도가 아니고서야 불가능할 것입니다(2011년 12월 31일 기준으로 페이스북의 회원 수는 8억 4,500만 명입니다. 매일 27억 개의 ‘좋아요(like)’ 및 댓글, 2억 5,000만 개의 사진 업로드, 친구 관계를 맺는 건수가 1,000억 건입니다).

국내에서도 큰 규모의 포털(네이버, 다음 등)이나 몇몇 게임 업체, 최근에 급성장하는 카카오톡 정도가 아니고서는 대규모 트래픽을 경험한 곳은 거의 없을 것입니다.

01 요청한 페이지의 url을 가져올 수 없을 때 발생하는 에러

02 애니메이션 ‘기동전사 건담 SEED’의 주인공

니다(대규모라고 하더라도 페이스북이나 트위터의 트래픽보다는 많이 부족할 것입니다).

어떤 성능 이슈가 발생할지 미리 알 수 있다면 대규모 트래픽을 처리하는 서비스를 개발하는 데 큰 도움이 되지 않을까요? 물론 서비스를 하다 보면 수많은 이슈가 발생합니다. 이를 전부 해결할 수는 없으므로 꼭 알아둬야 할 성능 이슈를 소개하고, 왜 그런 일이 발생하며, 어떻게 해결할 수 있을지에 대해서 고민해 보겠습니다.

1.1 대규모 트래픽 처리의 성능 이슈

트위터, 페이스북 등 우리 주변의 대형 서비스는 대부분 데이터를 저장하고, 읽는 동작을 제공합니다. 이때 병목이 발생하는 곳이 대부분 데이터스토어입니다. 데이터스토어는 일반적인 관계형 데이터베이스 관리 시스템(RDBMS)이 될 수도 있고, 요즘 인기를 끌고 있는 NoSQL이 될 수도 있습니다.

여/기/서/잠/깐 NoSQL이 뭔가요?

NoSQL은 'Not Only SQL'의 약어로, 일반적인 DBMS로 처리하기에 비용이 많이 들거나 효과적이지 못한 부분을 해결할 목적으로 개발됐습니다. 대용량의 로그 처리나 Write가 많은 경우에 사용하지만, 각자 목적에 따라서 선택할 수 있는 제품이 다르고 DBMS 만큼 쉽게 사용할 수는 없습니다.

NoSQL은 크게 세 가지 범주인 Key-Value Store, Column Oriented Store, Document Oriented Store로 나뉩니다. 이 책에서 소개하는 Memcached, Redis 같은 캐시 솔루션이 Key-Value 범주에 들어가고, key로 value를 바로 찾을 수 있는 해시의 데이터 구조를 사용합니다. 일반적인 RDBMS가 하나의 Record 단위로 저장되는 Row Oriented Store라고 할 수 있으며, Column Oriented Store는 컬럼(Column)별 정보를 모아서 저장하는 형태로 생각하면 됩니다. HBase, Cassandra 같은 것이 대표적인 예입니다. 마지막으로 Document Oriented Store는 데이터를 JSON 형태로 저장하는 구조입니다. 그래서 스키마를 바꾸기 쉬우며 MongoDB, CouchDB, Riak 등이 대표적인 제품입니다.

여기서는 ‘bit.ly’ 라든지 ‘goo.gl’과 같은 Short URL 서비스를 간단하게 설계해 구현하고, 테스트를 통해서 대규모 트래픽 처리 시 어떤 문제가 발생할 수 있는지, 그리고 어떻게 해결할 수 있는지 알아보겠습니다.

Short URL 서비스는 이름 그대로 Real URL(실제 인터넷에서 사용하는 URL)을 받아서 Short URL로 바꿔주는 서비스입니다. 현재 [bit.ly](#), [goo.gl](#) 등 많은 곳에서 서비스를 하고 있습니다. Short URL 서비스를 하려면 크게 두 가지 동작이 필요합니다.

표 1-1 Short URL에서 제공하는 대표적인 기능 두 가지

동작	타입	내용
Short URL 생성	Write	Real URL을 받아서 Short URL을 생성합니다.
Real URL 전달	Read	Short URL을 받으면 연결된 Real URL로 만들어줍니다.

실제로 Short URL 서비스를 하려면 고려할 사항이 많습니다. 등록된 URL이 신뢰할 만한 것인지에 대한 판단, 클릭에 따른 통계정보 저장 등 다양한 사항을 고려해야 합니다. 관련 내용은 이 책의 범위를 벗어나므로 이 책에서는 다루지 않겠습니다.

bit.ly에서 링크가 클릭되는 수는 하루에 1억 5백만 건으로 대단히 많은 트래픽을 처리하고 있습니다. 일단 이런 규모는 잊어버리고 간단하게 서비스를 설계하고 구현해 보겠습니다. 서비스를 구현하는 가장 간단한 방법은 Real URL과 Short URL 정보를 매칭해 저장하는 것입니다. SQL로 표현한다면 최초의 DB 스키마 schema는 다음과 같습니다.

```
create table tbl_shorturl
(
    uid int primary key auto_increment,
```

```
real_url varchar(255) UNIQUE,  
short_url varchar(255) UNIQUE,  
count int default 0,  
ctime datetime  
)ENGINE=INNODB;
```

Short URL을 만드는 원리는 간단합니다. 들어온 순서대로 count를 증가시킨 값을 전달하면 됩니다. DB의 auto_increment 속성을 이용해서 count 값을 자동으로 증가시킵니다. 즉, “<http://charyam.wordpress.com/1>”이라는 주소가 들어왔을 때 count가 1이라면 ‘<http://localhost/1>’ 형태로 변환됩니다.

위와 같은 스키마에서는 최초의 값을 insert한 후에, 해당 id 값을 다시 업데이트해야 하는 단점이 있습니다(Short URL을 구성하는 방법은 여러 가지이며, 해당 방법도 그중 한 가지입니다).

다음은 간단하게 Short URL을 생성하는 파이썬python 예제입니다.

예제 1-1 Short URL 생성 예제

```
import MySQLdb  
import urllib  
import sys  
import os  
import redis  
import pdb  
from flask import Flask, request, session, g, redirect, url_for, \  
abort, render_template, flash  
  
# configuration  
DB_HOST='127.0.0.1'  
DB_USER='dbuser'
```

```
DB_PASSWORD='db_password'
DB_FILE='shorturl'

DIVISION_SEED=0
DEBUG = True
SECRET_KEY = 'development key'
USERNAME = 'admin'
PASSWORD = 'default'

app = Flask(__name__)
app.config.from_object(__name__)

global g_cursor
domain="sho.rt"

def ChangeHex(n):
    x = (n % 16)
    c = ""
    if (x < 10):
        c = x
    if (x == 10):
        c = "a"
    if (x == 11):
        c = "b"
    if (x == 12):
        c = "c"
    if (x == 13):
        c = "d"
    if (x == 14):
        c = "e"
    if (x == 15):
        c = "f"
```

```

if (n - x != 0):
    return ChangeHex(n / 16) + str(c)
else:
    return str(c)

@app.route('/')
def show_entries():
    g_cursor.execute('select short_url, real_url from tbl_shorturl')
    urls = [dict(short_url=row[0], real_url=row[1]) for row in g_cursor.fetchall()]
    return render_template('show_list.html', urls=urls, domain=domain)

@app.route('/<short_url>')
def redirect_real_url(short_url):
    try:
        value = get_realurl( short_url )
        return redirect(value)
    except Exception as inst:
        return str(inst)

def create_short_url_data(seed, count):
    url = "%02x%s"%(seed, ChangeHex(count))
    return url

def get_count(cursor, url):
    cursor.execute( "insert into tbl_shorturl values( 0, '%s', '%s', 0,
                                                       now() )"%(url, url) )
    cursor.execute( "select last_insert_id()" );
    value = int(cursor.fetchone()[0])
    return value

def save_short_url( cursor, short_url, real_url ):
    cursor.execute( "update tbl_shorturl set short_url='%s' where

```

```

real_url='%"%( short_url, real_url ) )

def create_short_url(cursor, url):
    real_url = urllib.quote(url.encode('utf8'), '/:')
    try:
        count = get_count(cursor, real_url)
    except:
        raise Exception("Error: Duplicate url: %s"%url)

    short_url = create_short_url_data( DEVISION_KEY, count )
    save_short_url( cursor, short_url, real_url )
    return short_url

@app.route('/add', methods=['POST'])
def add_entry():
    try:
        real_url = request.form['real_url'].strip();
        if( real_url == '' ):
            flash("No url")
        else:
            short_url = create_short_url( g_cursor, request.form['real_url'])
            flash('New entry was successfully posted')
    except Exception as inst:
        flash(str(inst))

    return redirect(url_for('show_entries'))

@app.route('/create/<path:url>')
def create_shorturl(url):
    try:
        real_url = url.strip();
        if( real_url == '' ):

```

```

abort(401)

short_url = create_short_url( g_cursor, url )
except Exception as inst:
    return str(inst)

return "http://%s/%s"%(domain, short_url)

def get_realurl(short_url):
    g_cursor.execute( "select real_url from tbl_shorturl where
                      short_url='%s'%"%(short_url) )
    if( g_cursor.arraysize > 0 ):
        value = g_cursor.fetchone()[0]
    else:
        raise Exception("Error: No url exists")

    return value

if __name__=='__main__':
    db=MySQLdb.connect( host=DB_HOST, user=DB_USER, passwd=DB_PASSWORD, db=DB_FILE )
    g_cursor=db.cursor()
    g_cursor.execute( "SET AUTOCOMMIT=1" );
    start_port = int( sys.argv[1] )
    app.run(host='0.0.0.0', port=start_port)

```

최초의 서비스는 DB 서버 하나에서 Read/Write를 모두 처리하는 모델입니다. 물론, 백업을 위한 리플리케이션 Replication 서버는 필수입니다. 기본적으로 DB는 다음과 같이 구성해야 합니다.

그림 1-4 일반적인 서비스의 기본 구성



이때, 사용자가 늘어나면 웹 서버나 DB에서 병목현상이 발생합니다. CPU Load, Memory 사용량, I/O wait, log 수 등을 모니터링 지표로 이용해서 병목 지점을 찾아야 합니다.

여/기/서/잠/깐 예제를 실행하는 데 필요한 파일 모듈

이 책의 예제는 파이썬 2.7.x를 사용하여 만들었습니다. 테스트에는 MySQLDB라는 mysql 접속 모듈과 Flask라는 간단한 웹 서버 프레임워크를 사용했습니다. 설치해야 할 프로그램은 다음과 같습니다.

- ▶ setuptools : <http://pypi.python.org/pypi/setuptools>
- ▶ MySQLDB(Mysql-Python) : <http://sourceforge.net/projects/mysql-python/>
- ▶ Flask : <http://flask.pocoo.org/>

파이썬 모듈은 쉽게 설치할 수 있습니다. 다음 두 가지 명령으로 설치가 가능합니다.

```
python setup.py build  
sudo python setup.py install
```

MySQLDB는 ‘mysql include’와 ‘lib’에 대한 설정이 필요하고, 이를 위해 실행 가능한 path에 ‘mysql_config’를 링크해야 합니다. ‘virtualenv’를 이용해서 환경설정하는 것을 추천합니다.

이제 실제로 해당 서비스에 부하를 주어, 어느 정도의 성능이 나오는지 확인해 보겠습니다. 테스트하기 전에 장비 사양부터 소개합니다.

표 1-2 테스트 장비 사양

웹 서버		DB 서버(Mysql 5.5.x)
CPU	Intel Q9400 2.66G(Quad)	Intel i5 750 2.67G(Quad)
RAM	4G	16G(단, DB는 4G만 할당)
HDD	320G(7200RPM)	320G(7200RPM)

테스트 장비는 웹 서버와 DB 서버로 분리되어 있으며, 웹 서버는 파이썬 Flask를 사용했습니다. 기본적으로 insert process가 다섯 개며, read process를 2에서 64까지 증가시키면서 처리 리퀘스트 수를 비교합니다. 각 프로세스는 파이썬으로 간단하게 작성했으며, 소스코드는 다음과 같습니다.

예제 1-2 Short URL 테스트 : get_url.py

```
import MySQLdb
import urllib
import sys
import random

port=5000

def get_page(page):
    url = "http://10.64.81.187:%s/%s"%(port, page)
    web = urllib.urlopen(url)

if __name__=='__main__':
    port = int(sys.argv[3])
    start_page = int(sys.argv[1] )
    end_page = int(sys.argv[2] )
```

```
while(True):
    get_page( random.randrange( start_page, end_page ) )
```

예제 1-3 Short URL 테스트 : insert_url.py

```
import MySQLdb
import urllib
import sys

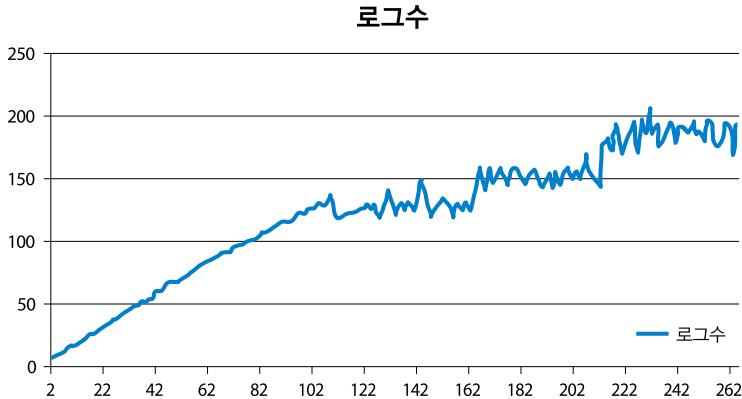
port=5000

def insert_list(page):
    url = "http://charyam.wordpress.com/%s"%page
    insert_page = "http://10.64.81.187:%s/create/%s"%(port, url)
    web = urllib.urlopen(insert_page)

if __name__=='__main__':
    port = int(sys.argv[3])
    for i in range(int(sys.argv[1]), int(sys.argv[2])):
        insert_list( i )
```

동작 결과는 그림 1-5와 같습니다. Read 프로세스가 늘어날수록 선형으로 처리 개수가 늘어나다가 어느 시점부터 Read 프로세스를 늘리면 증가가 멈추는 것을 볼 수 있습니다.

그림 1-5 테스트 결과 그래프(로그수)



벤치마크 benchmark 할 때 위와 같이 그래프상에서 한계에 다다른 것처럼 보이면, 성능상의 병목 원인이 테스트 클라이언트일 수도 있다는 사실에 주의해야 합니다. 즉 해당 시점에서 클라이언트에 다른 장비를 추가해 성능이 향상되는지 확인하는 것이 좋습니다.

여/기/서/잠/깐 벤치마크 어디까지 신뢰해야 할까?

인터넷을 검색해 보면 수많은 벤치마크 결과를 볼 수 있습니다. 이는 특정 제품이 좋다거나 또는 최소 어느 정도의 성능이 나온다고 믿는 근거가 됩니다. 벤치마크 결과를 볼 때 주의할 점은 다른 사람이 한 결과를 함부로 믿지 말고, 단순히 특정 상황에서는 특정 솔루션이 괜찮구나 정도로 보는 것이 좋습니다.

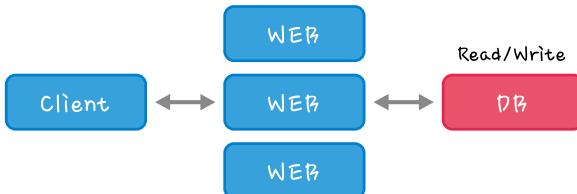
예를 들어, 단순히 “MySQL 벤치마크에서 초당 몇천 개를 처리하는데, 우리는 왜 몇 개밖에 처리하지 못하지?” 또는 “벤치마크 결과에서 A가 B보다 좋다고 했으니 A를 써야지?”라고 생각할 수 있는데, 자신이 사용하는 환경과 다를 수 있습니다. 벤치마크에서는 하나의 request가 select 하나로 끝나지만, 실제 서비스에서는 몇 번의 쿼리를 조합하고 업데이트 등 여러 조건이 다를 수 있으며, 데이터의 양이 다를 수 있습니다.

결국 벤치마크 결과는 어떤 시스템에 이런 특성이 있을 때, 이런 특성을 보여주더라 정도만 참고하는 것이 좋습니다(벤치마크는 직접 해봐야 정확한 결과를 얻을 수 있습니다).

프레임워크 Framework를 바꾸고 다른 웹 서버를 사용한 후 좀 더 최적화하면 전체적인 성능 차이가 크게 달라질 수 있지만, 이런 종류의 병목이 발생한다는 사실을 보여주는 간단한 테스트입니다.

웹 서버 쪽에 병목현상이 발생하면 어떻게 해야 할까요? 일반적으로 웹 서버는 정보를 요청만 하고, 자신이 따로 데이터를 저장하지는 않습니다. 즉, Stateless한 구조입니다. 이런 종류는 단순히 장비만 추가하면 해결할 수 있습니다.

그림 1-6 병목현상을 해결하기 위한 웹 서버의 확장



여기서 깐 Stateless한 구조는 왜 확장하기 편할까?

Stateful과 Stateless라는 두 용어가 있습니다. Stateful은 웹 서버에 클라이언트가 연결되면, 웹 서버에 모든 진행 정보가 들어 있습니다. 이 경우, 어떤 요청을 처리하는 중 웹 서버에 이상이 생기면 해당 클라이언트의 요청이 어디까지 처리됐는지 알 수 없습니다.

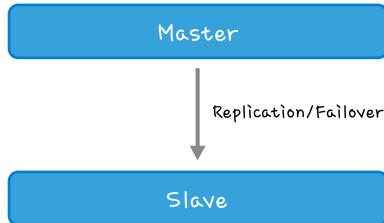
반대로 Stateless는 웹 서버에 어떤 정보도 저장하지 않고, DB 서버나 클라이언트에 해당 진행 정보가 모두 들어 있습니다. 이 경우, 웹 서버에 문제가 생겨서 다른 웹 서버에서 요청이 오더라도 현재까지의 진행 상황을 모두 알고 있으므로 문제가 되지 않습니다.

정말로 문제가 되는 것은 DB가 성능의 병목이 되는 시점입니다. 이제부터 DB의 병목현상을 대비해서 서비스를 어떻게 구축할 것인지 알아보겠습니다.

1.2 서비스의 시작, 하나의 DB에서 모두 처리하기

최초의 서비스는 하나의 DB 서버에서 Write/Read를 모두 처리하는 모델입니다 (물론, 백업을 위한 리플리케이션 서버는 장애 시 서비스를 빨리 복구하거나, 무정지 서비스를 구현하기 위해서 꼭 필요합니다). 즉 기본적으로 DB는 다음과 같이 구성해야 합니다.

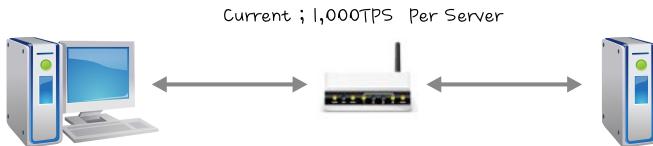
그림 1-7 일반적인 서비스의 DB 구성



성능 향상을 위해서 고민해야 할 것이 ‘Scale Up’과 ‘Scale Out’입니다. 예를 들어, 기존 서버가 1Ghz CPU와 메모리 2G로 서비스하고 있었다면, Scale Up은 성능을 세 배 높이기 위해서 3Ghz CPU와 메모리 6G를 장착한 장비를 이용해야 합니다. 전체 장비의 수는 같지만, 장비 하나의 성능을 세 배 높인 것입니다.

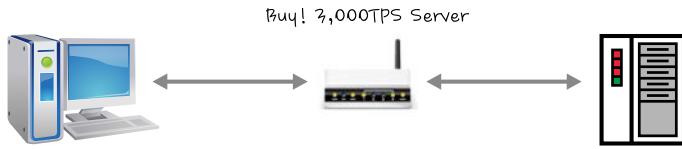
그림 1-8 Scale Up과 Scale Out

Want! 3,000TPS



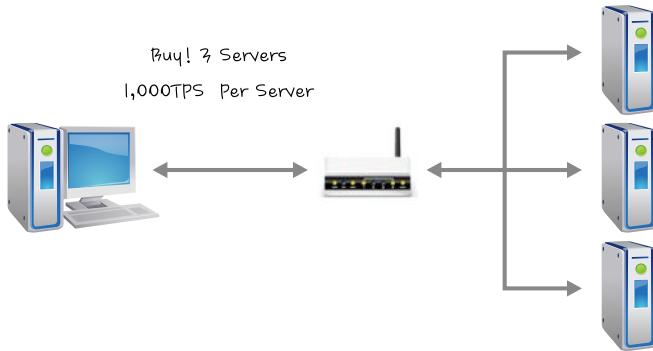
Scale Up은 기존의 아키텍처를 그대로 사용할 수 있어서 돈만 있다면 가장 적용하기 쉬운 모델입니다.

그림 1-9 Scale Up: 3,000TPS를 낼 수 있는 장비를 구입
Scale Up



Scale Out은 같은 장비를 세 대 사서 서비스에 투입하는 것입니다. 이때는 장비를 투입할 때마다 선형으로 성능이 늘어날 수 있는 아키텍처를 만드는 것이 중요합니다.

그림 1-10 Scale Out: 1,000TPS 장비를 세 대 구입
Scale Out

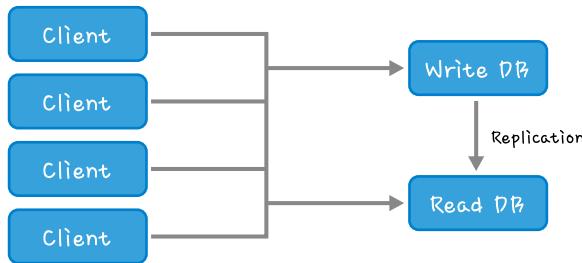


1.3 Read가 많으면 Read와 Write를 분리하자

DB에 병목현상이 발생할 때 특정 시점이 지나면 DB의 부하가 높아지는 것을 볼 수 있습니다. 여기서 서비스의 이용 패턴에 대한 분석이 필요합니다. 일반적인 서비스에서 Read/Write의 비율을 분석해보면 대략 7:3 또는 8:2 정도로 Read의 비율이 높습니다 (이것은 일반적인 경우로 서비스 특성에 따라서 비율은 달라질 수 있습니다).

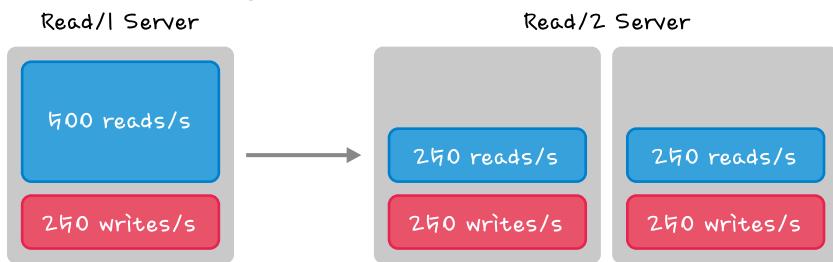
위 통계에 따르면 전체적인 DB의 부하를 떨어뜨리려고 Read를 분산시키면, DB 서버의 부하를 줄일 수 있다는 결론에 이르게 됩니다. 즉 One Master, Multi Slave 구성으로 Read와 Write를 다른 DB 서버로 분리하면 성능을 향상시킬 수 있습니다. Read 성능이 필요할 때마다 Read용 DB를 추가함으로써 DB 서버의 Read의 부하를 낮추고 마스터 DB의 전체 부하를 줄일 수 있습니다.

그림 1-11 One Master, Multi Slave 구조



Multi Slave로 Read를 분리하면 Slave DB 서버의 I/O 상황은 그림 1-12와 같습니다. Read 요청이 분리되면서 각 DB 서버에 어느 정도 여유가 생깁니다.

그림 1-12 Multi Slave를 이용하여 Read를 분리



여/기/서/잠/깐 슬레이브 DB 서버 수는 최소 몇 대가 되어야 할까요?

멀티 슬레이브(Multi slave)를 이용해서 Read를 분리하려면 DB 서버는 최소한 몇 대가 있어야 할까요? 답부터 공개하면 기본적으로 네 대가 필요합니다. 당연히 한 대는 Write를 처리할 마스터 DB고, 나머지는 Read를 분리하는 슬레이브 DB입니다. 왜 슬레이브가 세 대나 필요할까요?

장애 처리를 해야 하므로 세 대가 필요합니다. 슬레이브에 장애가 발생했을 때 리부팅으로 해당 장비의 장애를 해결할 수도 있지만, 최악의 경우에는 디스크가 깨지거나 장비를 변경해야 합니다. 이때 DB 서버가 세 대 미만이면 한 대에 장애가 발생하고 한 대만 서비스되는 상황이므로 새로운 장비를 셋팅할 때 데이터를 복구할 수 있는 서버가 없습니다(서비스 중인 DB에서 복사하면 부하가 기증됩니다). 그러므로 최소한 슬레이브가 세 대 있어야 한 대에 장애가 발생하면 다른 한 대로 서비스를 하고, 남은 한 대를 이용해서 복구할 수 있습니다.

여/기/서/잠/깐 Read를 분산시켰을 때의 문제점과 Eventual Consistency

Read를 분리하는 것은 성능을 향상시키는 데 도움이 되지만, 항상 적용할 수 있는 것은 아닙니다. 특히 데이터의 일관성(Consistency)이 중요한 경우에 문제가 발생할 수 있습니다. 예를 들어 계좌 A와 계좌 B에 10,000원이 있다고 가정해 보겠습니다. 계좌 A에서 계좌 B로 10,000원을 송금하고 계좌 A를 조회했을 때 어떨 때는 0원, 어떨 때는 10,000원이 보인다면 어떻게 될까요? 그리고 왜 이런 일이 발생하게 될까요?

발생 원인은 DB의 리플리케이션 방식 때문입니다. 일반적인 리플리케이션은 동기/비동기, 두 가지로 생각 할 수 있습니다. 동기식의 경우 슬레이브가 데이터를 복제한 다음에 클라이언트에게 응답을 주는 방식입니다. 그래서 마스터와 슬레이브의 데이터는 항상 같지만, 속도가 많이 느려집니다. 비동기식은 응답이 클라이언트에 최대한 빨리 전달되므로 속도는 빠르지만, 슬레이브에서 처리되기 전에 응답이 나가므로 마스터와 슬레이브 간에 데이터 불일치가 발생할 수 있습니다. 또한, 슬레이브가 여러 대인 경우 한 대는 리플리케이션이 빨리 완료되고 나머지는 늦게 될 때 어떤 DB 서버에서 정보를 가져오는가에 따라 값이 달라집니다.

그런데 이렇게 리플리케이션이 늦어질 경우 시간이 지나면 모든 데이터는 올바르게 같아집니다(물론 업데이트가 계속 발생하므로 리플리케이션이 차이 나는 경우도 있습니다. 하지만 시간이 지나면 데이터는 같아집니다). 이런 식으로 시간이 지나면 데이터가 같아진다는 것이 'Eventual Consistency'입니다. 당장은 같지 않더라도 시간이 지나면 같아진다는 것입니다. 이런 형태의 Consistency를 제공하는 것이 카산드라(Cassandra)입니다.

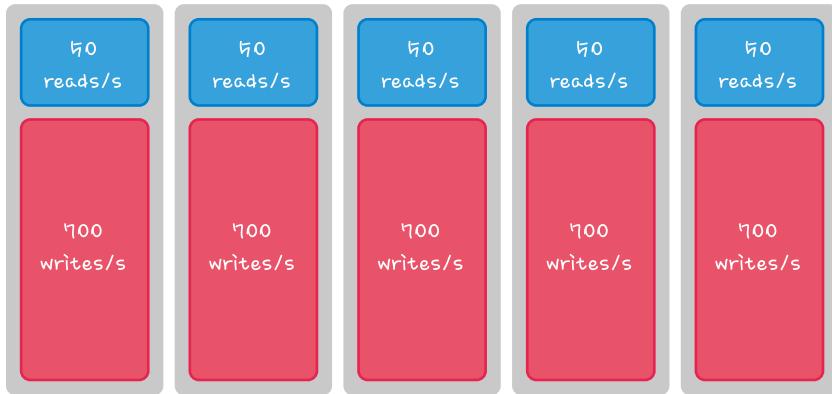
1.4 Write가 증가하면 파티셔닝하자

슬레이브 DB를 추가해 Read를 나눌수록 전체 성능이 올라가야 하는데, Request가 점점 많아지면 어느 시점부터 장비가 추가되더라도 성능이 높아지지 않게 됩니다. 왜 그럴까요? 장비 하나의 I/O 성능은 정해져 있습니다.

$$\text{I/O의 총 성능} = \text{Read 성능} + \text{Write 성능}$$

전체적인 Request가 증가해 Write가 늘어나게 되면, 슬레이브 장비들은 해당 Write를 리플리케이션 받기 위해서 일정 수 이상의 Write를 항상 처리해야 합니다. 그로 인해 Read를 분리하더라도 성능의 증가 폭은 점점 줄어듭니다. Write가 점점 더 늘어나면 그림 1-13과 같은 상황이 발생합니다.

그림 1-13 I/O의 대부분을 Write가 차지한 경우



한정된 I/O를 Write가 대부분 처리해서, 결국 Read의 성능도 줄어듭니다. 이럴 경우에는 전체적인 Write를 줄이는 파티셔닝 partitioning해야 합니다. 그렇다면 파티셔닝을 어떻게 해야 할까요?

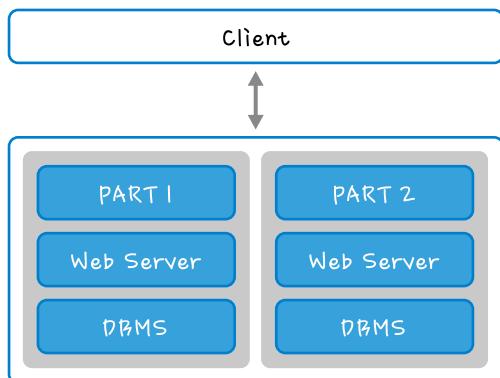
데이터베이스의 파티셔닝에는 크게 두 가지 개념, 수직 분할(Vertical Partitioning)과 수평 분할(Horizontal Partitioning)이 있습니다. Vertical과 Horizontal은 DB의 Column을 기준으로 설명하는 단어입니다. 즉 수직 분할이라는 것은 테이블의 Column에 A, B가 있다면 A는 A1 서버, B는 B1 서버 등으로 파티션하는 방법입니다. 수평 분할은 하나의 테이블에 있는 데이터를 특정 Row는 A1 서버 다른 Row는 B1 서버 등으로 나누는 것입니다. 흔히 말하는 Sharding이라는 것이 수평 분할을 의미합니다.

간단하게 정리하면 테이블 스키마가 같으면 수평 분할, 스키마가 서로 다르면 수직 분할이라고 할 수 있습니다.

파티셔닝은 데이터 모델에 따라서 여러 가지를 고민해야 합니다. 여기서는 insert 작업을 하는 서버를 기준으로 데이터를 파티셔닝하겠습니다. 예를 들어 웹 서버 1번과 2번으로 들어오는 요청은 DB 1번으로, 웹 서버 3번과 4번으로 들어오는 요청은 DB 2번으로 보냅니다. 이럴 경우 Read 시에 어떤 DB에서 읽어야 할지 알 수 없습니다. 따라서 Short URL을 생성할 때 첫 주소에 0이나 1등의 값을 부여해 해당 값으로 파티션을 찾습니다.

그림 1-14 데이터 파티셔닝

Scalable Partitioning



파티셔닝하면 ‘전체 request/N(파티션 수)’ 부하가 감소하는 효과가 발생합니다. 규모가 어느 정도 이상이 되면, 적당한 파티셔닝을 통해서 전체적인 Request를 제어합니다. 페이스북이나 트위터 등의 대규모 트래픽을 다루는 곳에서는 모두 이런 방식으로 파티셔닝하고 있습니다.

그럼 파티셔닝은 장점만 있는 것일까요? 그렇지 않습니다. 파티셔닝하면 성능은 향상되지만, 파티션이 늘어날수록 관리 포인트도 늘어납니다. 이에 따라 장비도 늘어나면서 비용도 증가하게 됩니다. 결국 파티셔닝의 규모나 시기는 서비스의 부하를 보고 적절히 결정할 수밖에 없습니다.

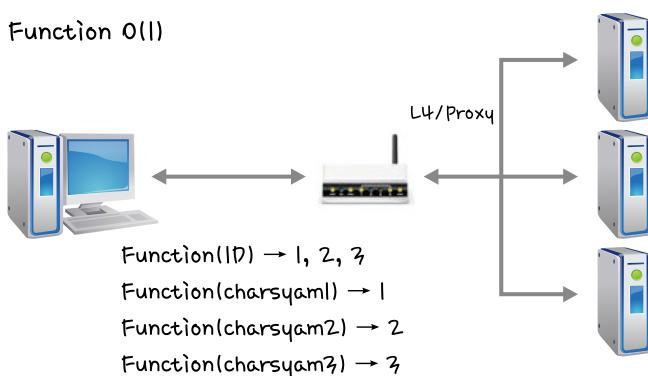
여/기/서/잠/깐 파티셔닝은 어떻게 해야 할까요?

파티셔닝하는 방법은 여러 가지가 있습니다. ID가 홀수인 사용자는 파티션1, 짝수인 사용자는 파티션2로 보내는 등의 연산을 이용하는 방법과 파티션 서버를 유지해서 특정 ID는 특정 파티션이라고 지정해 두는 방법이 있습니다.

각각의 방법에는 장단점이 있습니다. 특정 연산으로 파티션을 결정하는 방법은 파티션을 구하는 데 추가 정보가 필요 없다라는 것이 장점입니다. ID만 보면 누구나 쉽게 해당 파티션을 찾을 수 있습니다. 반대로, 특정 ID를 다른 파티션으로 옮기고 싶다면 파티션을 찾는 로직을 매번 바꿔야 하는 번거로움이 있습니다.

그림 1-15 특정 연산으로 파티션ning

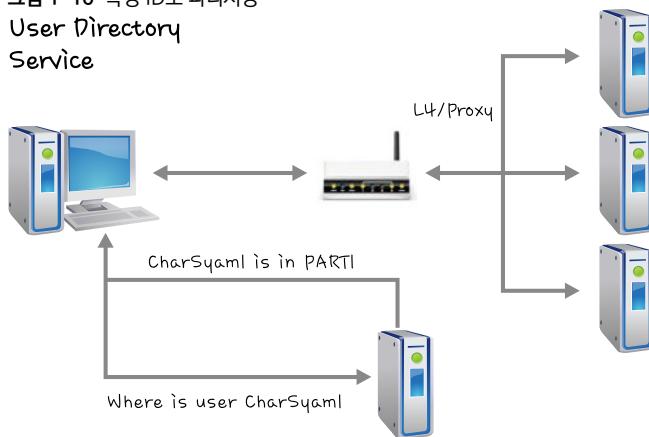
Function O(l)



특정 ID는 특정 파티션이라고 지정하는 방식은 원하는 형태로 파티션을 구성할 수 있지만, 특정 사용자의 파티션을 저장해두는 추가 작업이 필요합니다. 그리고 해당 서비스에 문제가 발생할 경우 모든 사용자가 서비스를 사용하지 못하게 되는 문제가 생길 수도 있습니다.

그림 1-16 특정 ID로 파티셔닝

User Directory Service



NoSQL 중에도 이와 비슷한 방식을 이용해서 데이터를 파티셔닝하고 있습니다. 카산드라는 Key를 해싱 한 후 해당 값을 이용해서 서버를 결정합니다. 즉, Key만 있으면 해당 데이터가 어느 서버에 있는지 알 수 있습니다. HBase는 Master 서버에서 데이터가 어디에 속해 있는지 알려줍니다.

페이스북에서 소개한 페이스북 메시지 아키텍처를 보면 사용자를 셀(Cell)로 나누고, User Directory 서비스를 통해서 해당 사용자가 몇 번째 셀에 있는지 알려줍니다(http://mvdirona.com/jrh/TalksAndPapers/KannanMuthukaruppan_StorageInfraBehindMessages.pdf).

2 | 분산 캐시를 구현하는 핵심 기술: Consistent Hashing

지금까지 서비스의 성능을 결정하는 일반적인 내용을 논의했습니다. 그런데 그 내용이 캐시와 어떤 연관이 있을까요? 앞에서 Read/Write의 성능 한계로 인해 파티셔닝하고, 이에 따라서 관리 비용이 늘어난다고 했습니다. 만약 파티셔닝 시점을 늦출 수 있다면 더욱 효과적으로 관리할 수 있을 것입니다.

캐시를 사용하면 각 단계의 이동 속도를 늦출 수 있습니다. 간단하게 말하면 더 적은 장비나 비용으로 더 효과적으로 성능을 낼 수 있습니다. 이 장에서는 캐시가 어떤 것인지 알아보고, 실제로 사용할 수 있는 분산 캐시를 알아보겠습니다.

캐시는 ‘이미 요청됐거나, 나중에 요청될 결과를 미리 저장해 두었다가 이를 빠르게 서비스해 주는 것’을 의미합니다. 우리는 알게 모르게 캐시를 사용하고 있습니다. 흔히 사용하는 웹 브라우저도 이전에 방문했던 페이지를 저장해 두었다가 변경되지 않았다면 그대로 보여주며, 대부분의 모바일 앱도 마지막 상태를 캐시하고 있다가 먼저 보여줍니다. 새로운 정보가 도착하면 그제서야 화면을 다시 보여줍니다. 이런 방법은 모두 속도를 위한 것입니다. 컴퓨터에서 사용하는 CPU에도 캐시가 있습니다. L1, L2, L3 심지어 하드디스크 안에도 캐시를 위한 메모리가 있습니다.

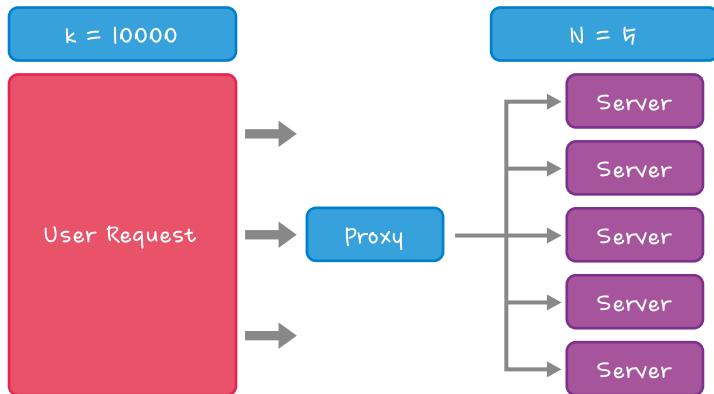
일반적으로 캐시는 디스크의 접근이 아닌 메모리의 접근을 의미합니다. 메모리에 접근하는 것이 디스크에 접근하는 것보다 훨씬 빠르기 때문입니다. 어떻게 보면 속도를 빠르게 한다는 것은 버블소트 bubble sort에서 퀵소트 quick sort 등으로 완전히 다른 알고리즘을 적용하거나, 디스크 접근을 최소한으로 줄이는 것이라 할 수 있습니다.

그런데 서버에 장착할 수 있는 메모리의 크기에 한계가 있습니다. 일반적인 서버는 16GB~48GB 정도의 메모리를 가지고 있습니다. 최근의 서버는 144GB 정도의 메모리 사용이 가능합니다. 그런데 실제로 데이터의 양은 이보다 훨씬 많습니다. 폼페이스북은 이미 페타^{petabytes} 이상의 데이터를 가지고 있습니다. 이 경우 16GB~48GB 정도의 메모리로 커버가 가능할까요? 결국 캐시 서버가 여러 대 필요해집니다.

2.1 Consistent Hashing

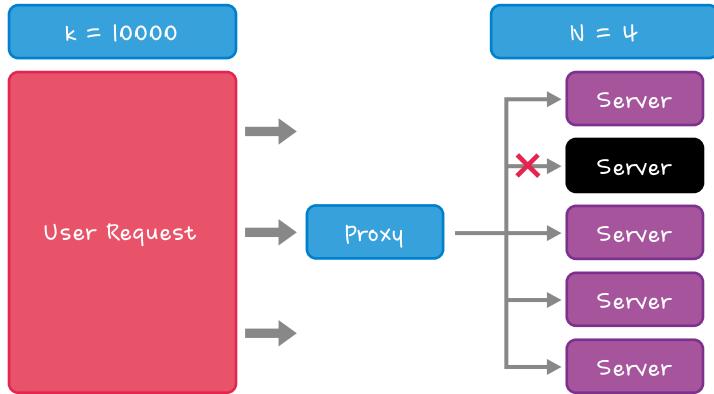
Consistent Hashing은 MIT의 David Karger라는 사람이 웹 서버 수가 변화하는 상황에서 분산 리퀘스트 Request를 처리하려고 고안했습니다. 이게 무슨 의미일까요? 간단하게 설명하면 그림 2-1과 같은 상황입니다. K는 요청하는 사용자 수, N은 웹 서버의 수입니다.

그림 2-1 User Request를 Proxy가 각 서버로 분배하는 상황



그런데 그림 2-2처럼 서버 몇 대에 장애가 발생하고 수시로 추가된다고 합시다.

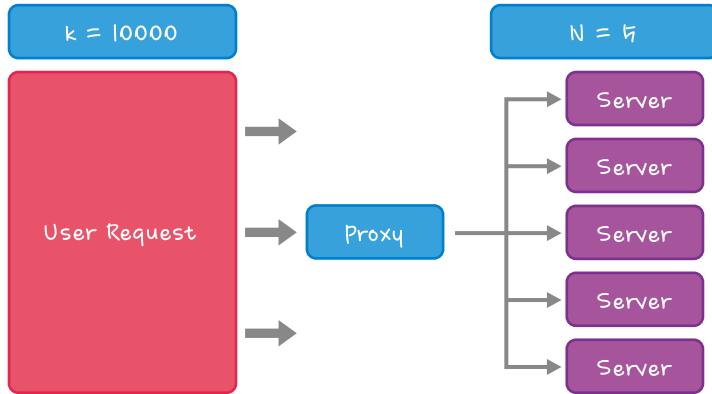
그림 2-2 장애로 인해서 서버 한 대가 서비스에서 제외된 상황



이렇게 되면 들어오는 리퀘스트는 나머지 네 대로 분배됩니다. 이걸 보면 이게 무슨 문제가 있느냐고 생각할 것입니다. 사실 별문제 없습니다. 각 리퀘스트가 전혀 연관 관계가 없다면 말입니다. 하지만 이미 세션이 연결돼 있어서 사용자가 기존 서버로 연결되기를 원한다면 문제가 복잡해질 수 있습니다(뭐, 해당 사용자가 전에 어느 서버로 갔었는지 기억하고 있다가 매번 그곳으로 보내주는 방식도 있지만, 이건 사용자별로 추가 작업이 필요합니다). 즉, 전체 사용자가 서버 수에 맞춰서 다시 재분배된다는 것입니다. 데이터가 추가될 때도 마찬가지입니다.

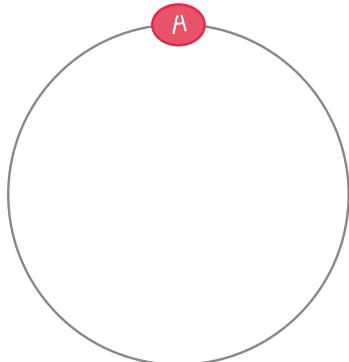
그럼 Consistent Hashing이 어떤 기능을 하는지 유추가 되나요? Consistent Hashing은 서버의 수, 즉 슬롯의 개수가 변하더라도 전체 데이터를 재분배할 필요가 없고 K/N 개의 아이템만 재분배합니다. 앞에서 서버가 다섯 대였으니 한 대당 2,000명을 처리합니다. 만약 서버 한 대에 장애가 발생하면, 장애가 발생한 서버의 2,000명만 서버를 재분배하고 나머지 사용자는 기존 서버에서 그대로 처리됩니다.

그림 2-3 장비가 다시 복구됐을 때의 상황



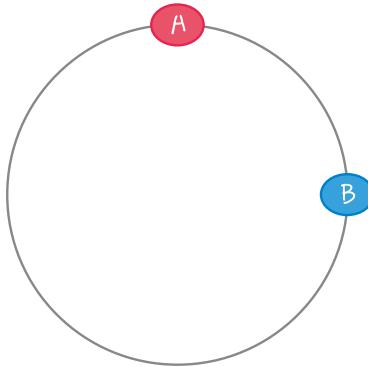
대단히 재미있는 기술이지 않습니까? 그럼 어떤 방식으로 처리되는 것일까요? 여기에는 HashRing이라는 개념이 들어갑니다. ‘Ring’은 둉근 것, 즉 꼬리에 꼬리는 무는 해시라고 해야 될까요. A, B, C, D 총 네 대의 서버가 있다고 가정합시다. 그림 2-4와 같이 처음에는 서버가 한 대만 있다면 모든 리퀘스트는 해당 서버로 가게 될 것입니다.

그림 2-4 HashRing에 서버 A가 등록된 상황



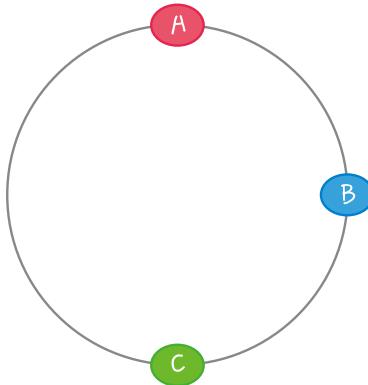
이 상황에서 서버 B가 추가되면 그림 2-5처럼 HashRing에 서버 2대가 등록됩니다.

그림 2-5 HashRing에 A, B 서버가 등록된 상황



여기서 서버 C가 추가되면 그림 2-6처럼 서버 3대가 HashRing에 등록됩니다.

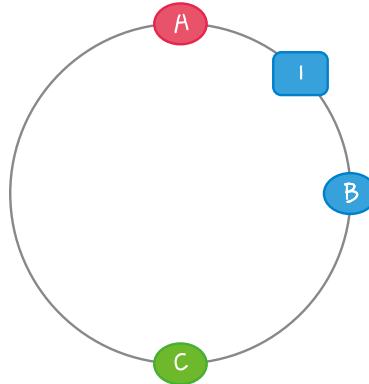
그림 2-6 HashRing에 A, B, C 서버가 등록된 상황



이제 해당 서버에 들어오는 Key의 해시값이 ' $A < \text{Key} \leq B$ '라고 한다면, 해당 Key는 자기보다 크고 가장 가까운 서버로 할당됩니다(그냥 규칙을 정하는 것입니다).

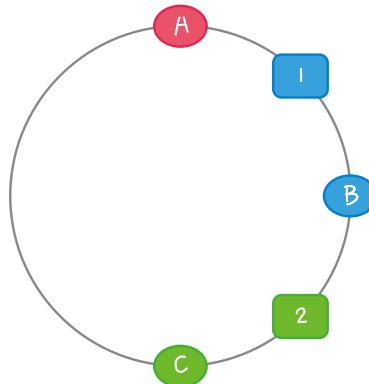
꼭 오른쪽으로 보내지 않고, 왼쪽으로 보내도 됩니다. 다만 규칙을 일관되게 적용해야 합니다). 즉, 여기에 Key 1이 들어와 ‘ $A < 1 \leq B$ ’라면 Key 1은 서버 B에 저장됩니다. 결과는 그림 2-7과 같습니다.

그림 2-7 HashRing에 ‘ $A < 1 \leq B$ ’인 Key 1이 들어온 상황



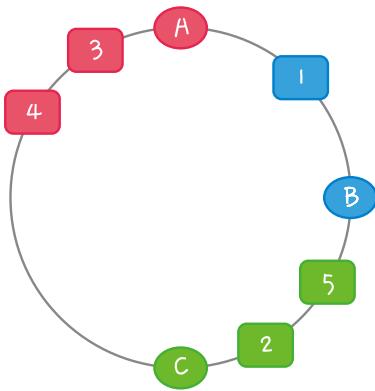
이제 Key의 해시값이 ‘ $B < 2 \leq C$ ’인 Key 2가 들어왔다고 합시다. 예상대로 Key 2는 그림 2-8과 같이 서버 C에 삽입됩니다.

그림 2-8 HashRing에 ‘ $B < 2 \leq C$ ’인 Key 2가 들어온 상황



이제 데이터가 엄청나게 많이 들어와서 그림 2-9와 같은 모습이 됐다고 합시다.

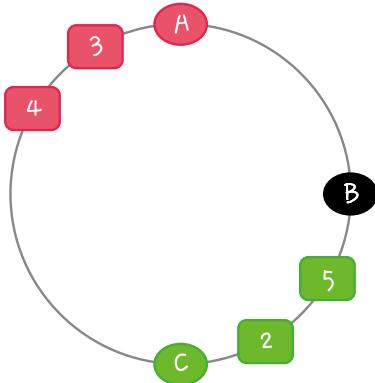
그림 2-9 데이터가 최종적으로 들어 온 모습



여기서 서버 B에 장애가 발생해 서비스하지 못하는 상태가 되었다고 가정합시다.

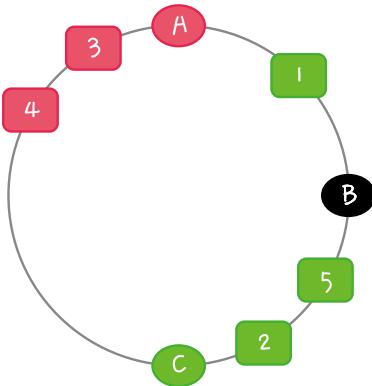
서버 B에 장애가 발생했으므로 Key 1은 유실됩니다. 그림 2-10의 상황입니다.

그림 2-10 서버 B의 장애로 인해서 HashRing에서 Key 1이 유실된 상황



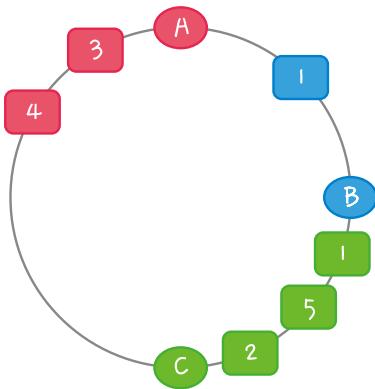
Consistent Hashing이 아니라면 그림 2-11의 장애 상황에서는 전체 Key가 재조정될 수도 있습니다. 하지만 Consistent Hashing 방식을 이용하므로 Key 5를 요구하면 해시값이 ‘ $A < 5 \leq C$ ’가 되므로 해당 서버가 그대로 서비스하게 됩니다. 이때 원래 B로 할당돼야 했던 Key 1이 다시 들어오면 어떻게 될까요? 예상한 바와 같이 $A < 1 \leq C$ 가 되므로 C에 할당됩니다(그림 2-11).

그림 2-11 서버 B가 장애일 때 Key 1이 들어온 경우



B가 복구돼 다시 투입된다면 어떻게 될까요? 그리고 1이 다시 추가된다면? 그러면 다음과 같은 상황이 벌어집니다.

그림 2-12 서버 B가 복구되고 다시 Key 1이 들어온 경우

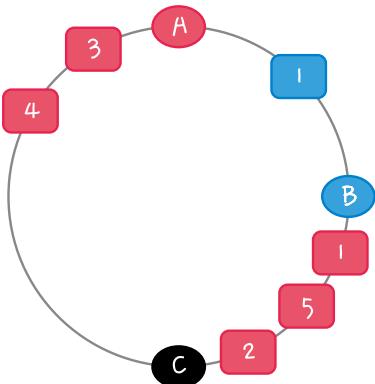


이런 경우 서버 B에도 데이터 1이 있고 서버 C에도 1이 있습니다. 그렇다면 데이터 1을 요청했을 때 어디서 데이터를 가져올까요? 원칙대로입니다. 우리의 원칙은 해시 값의 가장 가까운 오른쪽 서버로 가는 것입니다. Key 1의 해시값이 서버 A와 B 사이에 들어가는 값이므로 서버 B에서 1을 가져와서 처리합니다.

여기서 의문이 하나 생깁니다. 서버 C에 장애가 발생하고 Key 1, 2, 5가 추가되면 서버 A에 대부분의 데이터가 몰리게 됩니다. 즉 그림 2-13과 같은 결과가 나오지 않을까요?

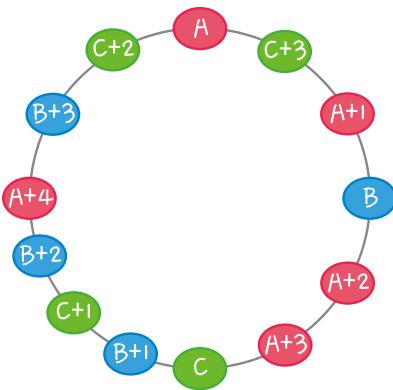
이렇게 되면 서버 A의 부하가 높아져서 전체적으로 장애가 일어나지 않을까 걱정이 될 것입니다. 한 번에 많은 데이터가 사라질 가능성도 높습니다. 그래서 실제로는 그림 2-14와 같이 하나의 서버에 값을 +1, +2, +3 식으로 해시값을 더 만듭니다.

그림 2-13 서버 C에 장애가 발생하고 Key 1, 2, 5가 추가됐을 때의 예상도



해시의 특성상 여러 값이 나오므로 전체적으로는 균등하게 퍼지는 효과가 있습니다. 이로 인해서 장애가 발생해도 전체적으로 다른 서버에게 일정량이 넘어가고, 한 번에 확 넘어가지 않습니다.

그림 2-14 Consistent Hashing을 이용해 실제 구현한 모습



이런 식으로 Consistent Hashing은 서버가 추가되거나 장애 발생 시 K/N개의 Key를 재분배해 문제를 해결하는 알고리즘입니다. Memcached는 이런 방식을 사용합니다. 정확하게 이야기하면 Memcached Client Library에서 이렇게 키를 저장할 서버를 선택합니다. 다음의 간단한 파이썬 코드를 보죠. 먼저 Localhost에 20001, 20002, 20003, 20004포트를 이용하는 memcached 서버 4대를 실행합니다.

여/기/서/점/깐 예제 결과 확인을 위한 프로그램 설치

파이썬 코드의 결과를 확인하려면 Libevent, Memcached, python-memcached 모듈이 설치되어 있어야 합니다. Libevent와 Memcached는 “3.1.1 Memcached 설치하기”를 참고하여 설치합니다. 그리고 python-memcached 모듈은 우부투Ubuntu 명령쉘에서 “`pip install python-memcached`”를 실행하면 해당 모듈이 설치합니다.

```
charsyam@charsyam-lv63:~/repo/memcached-1.4.13$ ps -ef | grep "memcached"
charsyam 11243      1  0 23:04 ?        00:00:00 ./memcached -p 20001 -m 64 -d
charsyam 11250      1  0 23:04 ?        00:00:00 ./memcached -p 20002 -m 64 -d
charsyam 11257      1  0 23:04 ?        00:00:00 ./memcached -p 20003 -m 64 -d
charsyam 11264      1  0 23:04 ?        00:00:00 ./memcached -p 20004 -m 64 -d
```

그리고 다음과 같은 set.py 프로그램을 만들어서 실행합니다.

예제 2-1 간단한 1에서 10까지 memcached에 저장하는 set.py 예제

```
import memcache
ServerList=[ '127.0.0.1:20001', '127.0.0.1:20002', '127.0.0.1:20003',
'127.0.0.1:20004' ]

if __name__=='__main__':
    mc = memcache.Client( ServerList );
    for idx in range(1,10):
```

```
mc.set( str(idx), str(idx) )
```

해당 코드를 실행한 후에 각 Memcached 서버에서 확인해보면, 1~10까지의 값들이 서버 4대에 분배된 것을 볼 수 있습니다. 값들이 서버에 어떻게 분배되어 있는지 확인할 수 있는 Memcached 명령어는 ‘get’입니다. get 명령어의 형식은 다음과 같습니다.

```
get [KEY]
```

get 명령어 다음에 Key를 string 형태로 입력합니다. 이에 대한 출력은 다음과 같습니다.

```
VALUE [KEY] [FLAGS] [LENGTH]
[VALUE]
END
```

[KEY]는 입력된 Key를 다시 한 번 보여줍니다. [FLAGS]는 flag 값으로 Key 저장 시에 설정합니다. [LENGTH]는 실제 VALUE의 길이에 대해서 알려줍니다. 즉, [VALUE]는 [LENGTH]의 크기만큼 저장되어 있습니다. END는 서버가 get 명령어에 대한 응답결과를 정상적으로 전달했음을 나타냅니다. 명령에 대한 자세한 설명은 "<https://github.com/memcached/memcached/blob/master/doc/protocol.txt>"에서 확인할 수 있습니다.

```
charsyam@ubuntu:~$ telnet 127.0.0.1 20001
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
get 1
```

```
VALUE 1 0 1
1
END

charsyam@ubuntu:~$ telnet 127.0.0.1 20002
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

get 1
END
get 2
VALUE 2 0 1
2
END
```

위의 결과로 서버 1에는 값 1이 저장되어 있음을 확인할 수 있습니다. 그리고 서버 2에는 값 2가 저장되어 있음을 확인할 수 있습니다. get 명령어로 서버에 값들이 어떻게 저장되어 있는지 확인해보기 바랍니다.

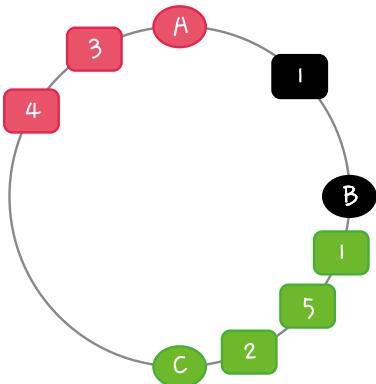
실제로 Memcached의 C 클라이언트 라이브러리인 libmemcached의 소스를 보면 memcached_send라는 함수가 있고, 이 안에 있는 memcached_generate_hash_with_redistribution라는 함수 안에서 Key의 해시값을 구하고 해당값이 포함되어야 하는 서버의 해시값을 알려줍니다.

```
uint32_t memcached_generate_hash_with_redistribution (memcached_st *ptr,
                                                       const char *key, size_t key_length)

{
    uint32_t hash= _generate_hash_wrapper(ptr, key, key_length);
    _regen_for_auto_eject(ptr);
    return dispatch_host(ptr, hash);
}
```

거의 발생하지 않지만 Consistent Hashing을 이용할 때, 이전에 등록된 오래된 데이터가 나와서 문제가 될 수 있습니다. 어떤 경우나 하면 장애가 발생한 경우입니다. 그림 2-11에서 실제로 서버 B와 서버 C에는 모두 1이라는 Key가 있습니다. 둘 다 정상일 때는 1의 Key 범위는 B에 속하므로 B에 추가된 최신의 데이터가 전달됩니다. 만약, 이 상황에서 B에 장애가 발생하면 어떻게 될까요? 그림 2-15처럼 1을 요청할 때 서버 B가 없으므로 다음 해시는 서버 C에 속하게 되고, 이전에 서버 C에 저장돼 있던 이전 Key 1의 데이터가 전달됩니다.

그림 2-15 Consistent Hashing에서 문제가 발생할 수 있는 경우



즉 장애가 빈번할 때 이런 문제가 발생할 확률이 높습니다. 기본적으로 Expire Time을 지정해서 특정 시간 이후에 해당 Key를 사라지게 하면, 이런 문제가 발생할 확률이 거의 없어집니다. 그리고 같은 서버가 죽었을 때만 위와 같은 문제가 발생합니다. 이런 장애 상황을 그려보는 것도 Consistent Hashing에 대한 이해를 높여줍니다.

3 | 분산 캐시의 활용

이 장에서는 Memcached와 Redis에 대해서 알아 보겠습니다.

3.1 Memcached

Memcached⁰³는 LiveJournal이라는 서비스를 개발한 Danga⁰⁴라는 회사에서 만든 메모리 분산 캐시 솔루션입니다. Danga라는 회사를 잠시 소개하면 Memcached뿐만 아니라 MogileFS, Gearman 등 멋진 솔루션을 개발한 회사입니다. MogileFS는 분산 파일시스템이며, Gearman은 일종의 Queue와 Job 서비스를 제공해주는 플랫폼입니다.

3.1.1 Memcached 설치하기

Memcached는 현재 버전 1.4.13(2012년 2월 기준)까지 나와 있습니다. Memcached는 libevent에 대해서 의존성이 있으므로 함께 설치해야 합니다. 먼저 Memcached를 설치하는 방법부터 간단히 알아보겠습니다. 필요한 라이브러리는 다음과 같습니다.

표 3-1 Memcached에 필요한 라이브러리

라이브러리	홈페이지
memcached	www.memcached.org (1.4.13)
libevent	libevent.org (2.0.17)

libevent는 다음과 같이 설치합니다.

03 : www.memcached.org

04 : www.danga.com

```
root@charsyam ~/src # wget  
http://cloud.github.com/downloads/libevent/libevent/libevent-2.0.17-  
stable.tar.gz  
root@charsyam ~/src # tar zxvf libevent-2.0.17-stable.tar.gz  
root@charsyam ~/src # cd libevent-2.0.17-stable  
root@charsyam ~/src/libevent-2.0.17-stable # ./configure  
root@charsyam ~/src/libevent-2.0.17-stable # make  
root@charsyam ~/src/libevent-2.0.17-stable # make install
```

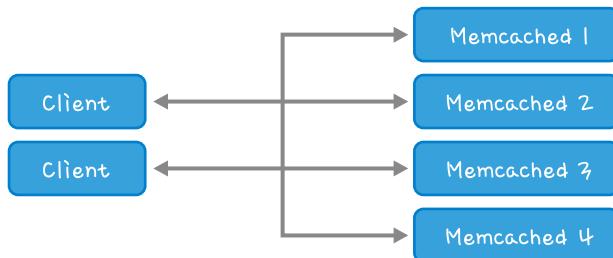
Memcached 역시 간단하게 설치할 수 있습니다.

```
root@charsyam ~/src # wget  
http://memcached.googlecode.com/files/memcached-1.4.13.tar.gz  
root@charsyam ~/src # tar zxvf memcached-1.4.13.tar.gz  
root@charsyam ~/src # cd memcached-1.4.13  
root@charsyam ~/src/memcached-1.4.13 # ./configure  
root@charsyam ~/src/memcached-1.4.13 # make  
root@charsyam ~/src/memcached-1.4.13 # make install
```

3.1.2 Memcached 사용하기

Memcached를 분산 캐시라고 부르지만, 사실 Memcached 자체에는 분산 기능이 없습니다. Memcached의 라이브러리인 Consistent Hashing을 통해서 데이터를 분산합니다. 일반적으로 Memcached는 다음과 같은 형태로 시스템을 구성합니다.

그림 3-1 일반적인 Memcached 서비스의 구조



클라이언트가 Memcached 서버의 주소를 모두 알고 있는 상태에서 서비스를 처리합니다. 위의 기본 구조는 Memcached 서버 한 대가 죽더라도 크게 문제가 발생하지 않습니다. 다음과 같이 간단하게 테스트해 보겠습니다.

예제 1-1의 set.py를 실행한 다음 get.py를 실행하겠습니다.

예제 3-1 간단한 1에서 10까지 memcached에서 읽어오는 get.py 예제

```
import memcache  
ServerList=[ '127.0.0.1:20001', '127.0.0.1:20002', '127.0.0.1:20003',  
'127.0.0.1:20004' ]  
  
if __name__=='__main__':  
    mc = memcache.Client( ServerList );  
    for idx in range(1,10):  
        print mc.get( str(idx) )
```

그러면 다음과 같은 결과가 나옵니다.

```
charsyam@charsyam-lv63:~/test/py-memcached$ python get.py
1
2
3
4
5
6
7
8
9
10
```

이 상태에서 포트 20002의 Memcached를 강제로 종료한 후에, 다시 get.py를 실행하면 다음과 같은 결과가 나옵니다.

```
charsyam@charsyam-lv63:~/test/py-memcached$ python get.py
1
None
3
4
None
6
7
8
9
10
```

2번과 5번에 데이터가 없는 것을 확인할 수 있습니다. set.py를 실행한 후 다시 get.py를 실행시켜 보겠습니다.

```
charsyam@charsyam-lv63:~/test/py-memcached$ python get.py  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

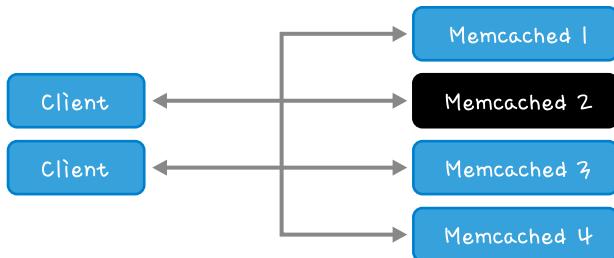
이번에는 데이터가 전부 나오는 것을 볼 수 있습니다. 이것은 어떤 의미일까요? 그림 3-2와 같이 Memcached 2에 장애가 발생하면 해당 서버의 데이터는 순간 사라집니다. 해당 서버는 자동으로 복구되지 않으므로 데이터 일부가 유실되는 것입니다. 그래서 Memcached 2에 있었던 Key를 요청하면, 데이터가 없다는 응답이 나옵니다.

캐시에 데이터가 없다면, 그 시점에 사용자가 데이터를 복구할 수 있습니다. 예를 들어, 실제 서비스에서는 DBMS에서 데이터를 읽은 후 캐시한 다음 복구합니다. 모든 Memcached 서버가 동시에 장애가 발생한 상황이 아니라면, 사용자가 DBMS에서 장애 시점을 읽어서 다시 캐시하는 로직을 미리 만들어 두면 Memcached 서버에 장애가 발생해도 별다른 문제 없이 서비스를 제공할 수 있습니다(실제 서비스에서는 장애를 모니터링하는 별도의 시스템을 만들어 자동으로 복구합니다). 이것이 Consistent-Hashing의 장점입니다. Consistent-Hashing이

아니라면 장애 발생 시, 일반적으로 다른 프로그램에서는 남아있는 서버만큼만 트래픽을 보내기 위해서 서버 설정을 다시 배포해야 합니다. 설정을 바꿔서 배포하지 않으면 장애가 해결되지 않거나 다른 문제가 발생하여, 서버가 제대로 작동하지 않을 수도 있습니다.

다시 같은 Key를 저장하려고 하면 다른 서버에 제대로 저장이 된다는 것을 의미합니다. 이것은 장비의 장애가 서비스의 운영에 문제를 주지 않는다는 것입니다.

그림 3-2 일반적인 Memcached 서비스 구조에 장애가 발생했을 경우



3.1.3 Moxi Proxy를 이용해서 Memcached를 사용해보자

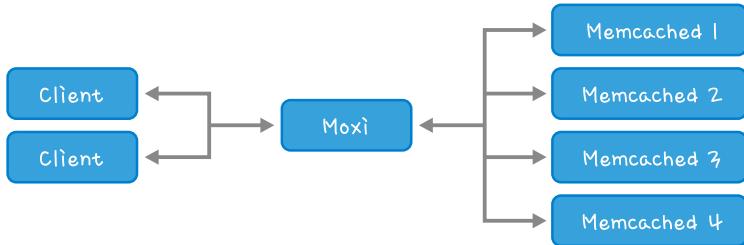
Memcached를 설치하면 장애 시에는 크게 문제가 발생하지 않습니다. 반대로 장비가 늘어나면 어떻게 될까요? 소스를 주섬주섬 열어서 새로운 장비의 목록을 코드에 추가해야 합니다. 정상적으로 돌아가는 소스는 아무리 간단한 변경이라도 하기 싫은 법입니다. 좀 더 성실한 개발자라면 해당 Memcached 서버의 목록을 설정으로 빼두었을지도 모릅니다. 하지만 이것은 당연히 해야 하는 일이고, 서버 재시작 등의 비용이 들어 갑니다. 그럼 어떻게 접근하는 것이 좋을까요? 바로 Memcached Proxy인 Moxi를 이용하면 됩니다.

Moxi는 현재 couchbase라는 프로젝트에 있는⁰⁵ 일종의 Memcached Proxy입니다.

05 Membase에서 이름이 변경됐습니다.

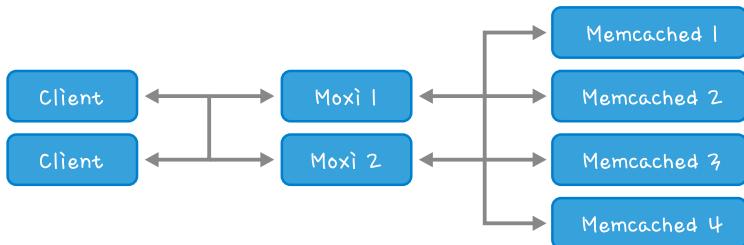
니다. Stateless 구조로 실제 라이브러리에서 해야 할 Consistent Hashing까지도 대신해줍니다. 즉 우리는 Moxi의 주소만 알고 있고, Moxi가 실제 Memcached 서버와 연결되어 있는 구조입니다.

그림 3-3 Moxi를 이용한 Memcached 구조



그런데 위와 같은 구조에서 Moxi에 장애가 발생하면, 서비스를 할 수 없는 문제가 발생합니다. 기존에는 Memcached 서버 중 한 대가 죽더라도 문제가 되지 않았는데, 이 구조에서는 Moxi 서버가 SPOF^{Single Point Of Failure}가 됩니다. 그런데 Moxi의 경우 Stateless하므로 몇 개를 사용하든 문제가 없습니다. 실제로 사용할 때는 Moxi의 수를 늘려서 장애에 대비할 수 있도록 사용합니다.

그림 3-4 Moxi에 대한 FailOver를 적용한 Memcached 구조

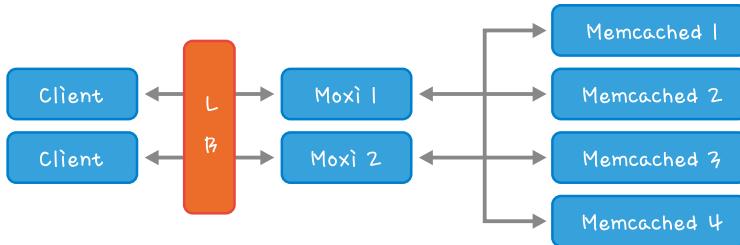


Memcached 장비가 추가될 때는 Moxi를 순차적으로 한 대씩 내리고 다시 설정한 후에 다시 실행하면 됩니다. 일반적으로 Moxi는 다음과 같은 명령으로 사용합니다.

```
moxi -z 11211=server1:20001,server2:20002,server:20003,server:20004
```

Moxi를 다시 추가해야 할 경우에는 같은 문제가 발생할 수 있습니다. 그래서 최종적으로는 Moxi 앞에 LB^{Load Balancer}를 사용하여 이런 문제를 해결합니다.

그림 3-5 Load Balancer를 이용한 Moxi의 확장

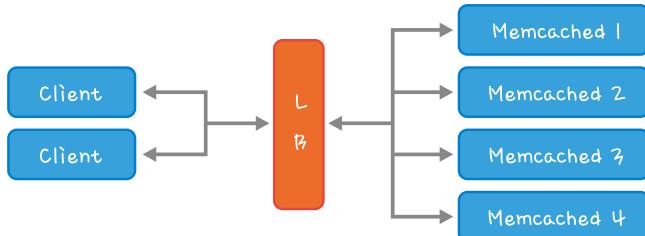


여기서 깐 처음부터 Memcached 앞에 Load Balancer를 사용하면 되지 않나요?

Moxi를 사용하는 구조에서 최종적으로 Load Balancer를 쓰는 것을 보고 “그럼 처음부터 Moxi를 안 쓰고 Load Balancer만 사용하면 되지 않나요?”라는 의문이 생길 수 있습니다. 그렇게 사용하면 어떤 일들이 벌어질까요? 당연히 위에서 설명 안하고 여기다가 추가로 설명하는 걸 보면 좋은 일은 벌어지지 않을 것 같다라는 굳은 희망이 생기실 겁니다.

Load Balancer와 Memcached를 바로 연결하면 다음과 같은 구조가 됩니다.

그림 3-6 Load Balancer와 Memcached의 다이렉트 연결



Memcached는 자체적으로 분산 능력이 없고, 라이브러리에서 Consistent Hashing을 해주는 것이라는 걸 기억하시길 바랍니다. 이 경우 클라이언트에는 서버 주소가 하나만 보이므로, Consistent Hashing도 한 곳으로만 가게 됩니다. 그런데 Load Balancer 때문에 실제로 어느 서버와 연결이 맺어지는지 모릅니다. 그리고 클라이언트가 접속할 때마다 어떤 Memcached 서버와 연결될지 모르므로, 한 곳에서 데이터를 추가해도 다른 쪽에서는 해당 데이터를 볼 수 없게 됩니다. 그렇기 때문에 Moxi를 사용하지 않고 Load Balancer와 바로 연결하면 데이터가 뒤틀리게 되어버립니다. 절대로 이런 구성을 하지 않으시길 바랍니다.

3.1.4 Memcached를 사용할 때는 항상 ExpireTime에 주의하자

Memcached를 사용할 때는 ExpireTime에 주의해야 합니다. Memcached의 모든 Key에는 생명주기가 있습니다. 이 생명주기는 초 단위로 지정합니다. ExpireTime을 설정한 Key는 지정한 시간이 지난 뒤에 메모리에서 사라집니다. 이때 주의할 점은 30일 이내의 경우는 초 단위로 지정하지만, 30일이 넘어가는 값에 대해서는 유닉스 타임스탬프로 생각하고 인식한다는 겁니다. 이를 기억하지 못할 경우에 데이터가 제대로 존재하지 않아서 고민하게 될 것입니다.

표 3-2 ExpireTime의 생명주기

ExpireTime	내용
0	LRU ⁰⁶ 로 지워질 때까지 지워지지 않는다.
30	생명주기는 30초다.
2592000	생명주기는 30일이다.
2592001	1970. 01. 31. (토) 09:00:01 KST로 지정된다. 즉 데이터를 set하자마자 사라진다.

06 LRU(Least Recently Used, 최저 사용 빈도)는 운영체계의 페이지 교체 알고리즘 중 하나입니다.

그리고 ‘flush_all’이라는 명령을 사용해서 모든 key를 지울 수 있습니다. ExpireTime을 지정하지 않을 경우에는 모든 Key를 삭제하고, ExpireTime이 설정된 경우에는 그 시간 내의 ExpireTime이 남은 모든 데이터가 삭제됩니다. 예를 들어, 현재 시간이 2012/02/28 02:23:10라고 한다면 ExpireTime을 10으로 주면 생명주기가 2012/02/28 02:23:20내에 있는 모든 Key가 삭제됩니다. 일반적으로 flush_all은 모든 데이터를 지울 때 사용합니다.

```
Flush_all [<expire time>]\r\n
```

여/기/서/잠/깐 Memcached는 생명주기가 지난 데이터를 어떻게 삭제할 수 있을까?

이쯤 되면 궁금증이 하나 생길 것입니다. Key마다 ExpireTime을 설정해 줄 수 있는데, 이 데이터는 어떻게 삭제하는 걸까요? Memcached에서는 ExpireTime이 지난 시점에 바로 데이터를 삭제하지 않습니다. LRU가 동작하지 않았다는 가정하에, 실제로 get을 통해 데이터를 읽으려고 할 때 ExpireTime을 체크하고, 그때 “Key가 존재하지 않는다”라고 응답을 줍니다.

그럼 왜 이런 식으로 동작하게 만들었을까요? ExpireTime마다 해당 데이터를 삭제하려면 너무 긴 시간이 필요하기 때문입니다. 그보다는 데이터를 읽을 때 ExpireTime을 확인하면 쉽게 데이터를 삭제할 수 있습니다.

이런 특성 때문에 Memcached는 ExpireTime을 지정했더라도 메모리 사용량이 줄어들지 않은 것처럼 보일 수 있습니다. 실제로 read 시점에서야 데이터가 사라지기 때문입니다.

그럼 delete/flush_all은 어떻게 동작하는 것일까요? delete/flush_all은 모든 데이터를 직접 지웁니다. 즉 ExpireTime의 설정에 의한 데이터 삭제와 delete/flush_all을 통한 데이터 삭제는 서로 다른 것임을 알아두어야 합니다.

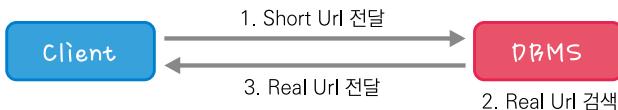
이미 메모리 사용량이 한계에 도달했다면, 데이터가 추가되는 시점에 LRU가 발생합니다. 그런데 재미난 것은 LRU가 발생하면 전체 데이터 중 빈 공간이 남았으면 주는 것이 아니라 해당 데이터가 추가되어야 하는 슬랙에서만 LRU를 확인한다는 것입니다. 즉, 단순히 먼저 데이터가 들어갔다고 해서 LRU 정책으로 인해서 사라지는 것이 아닙니다.

3.1.5 Memcached를 어디에 사용할 수 있을까?

지금까지 Memcached를 어떤 구조로 사용하면 좋을지에 대해서 이야기를 했지만, 실제로 어떤 식으로 이용할지 고민이 생길 것입니다. 1장에서 설명한 ‘Short URL 서비스’를 예로 든다면 Memcached를 어떻게 사용할 수 있을까요?

최초에 정의했던 Short URL 서비스의 동작은 크게 두 가지입니다. 첫 번째는 short_url을 읽는 동작이고, 두 번째는 short_url을 새로 생성하는 동작입니다. 먼저 읽기 동작부터 알아보겠습니다. 일반적인 ‘읽기’ 요청은 다음과 같은 흐름으로 진행됩니다.

그림 3-7 일반적인 Short URL 읽기 동작



그렇다면 읽기 속도의 대부분은 어디에서 결정될까요? 바로 URL을 저장하고 있는 DBMS에서 읽기 속도가 결정됩니다. 일반적으로 ‘80:20의 법칙’이 적용되는데, 사용자가 자주 접근하는 20%의 URL이 실제로 전체 서비스 읽기 요청의 80%라고 가정하면 읽기 속도만 향상시켜도 성능이 엄청나게 향상됩니다.

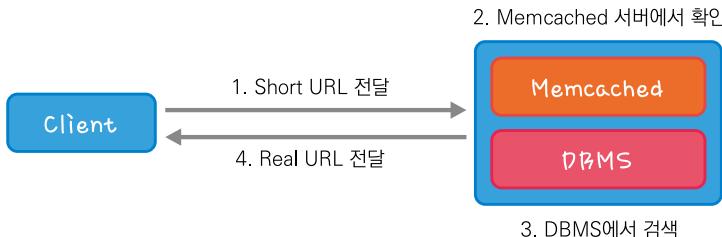
읽기 속도를 향상시키기 위해 Key를 Short URL로, value를 Real URL로 설정해서 Memcached에 저장합니다. Memcached 명령으로 본다면 다음과 같습니다.

```
set( short url, real url )
```

읽기 요청이 들어왔을 때 Memcached 서버에서 먼저 찾습니다. Memcached는 해시 hash 형태이므로 $O(1)$ ⁰⁷의 성능을냅니다. 그래서 검색 시간은 매우 짧습니다 (일반적으로 1ms 이내에 응답이 나옵니다).

그리고 Memcached에 없을 경우에는 DBMS에서 검색합니다. 이렇게 하는 이유는 일반적으로 메모리 캐시가 DBMS 용량보다 적기 때문입니다. 즉 다음과 같은 구조가 됩니다.

그림 3-8 ‘읽기’ 캐시를 적용한 구조



‘쓰기’도 마찬가지입니다. 그런데 쓰기는 무조건 DBMS에 저장해야 합니다. 그렇다면 항상 DBMS에 접근하게 되므로 속도의 이점이 없지 않을까요? 이 문제를 어떻게 처리해야 할까요? 방법은 여러 가지가 있을 수 있습니다만, 간단한 방법은 다음과 같습니다.

1. Short URL 생성 시에 로그를 먼저 씁니다.
2. Short URL을 생성한 다음 Memcached에 저장하고 응답을 리턴합니다.
3. 1에서 저장한 로그를 배치 작업으로 실제 DBMS에 저장합니다.

07 $O(1)$ 은 BigO라는 성능 표시 방법입니다. $O(1)$ 은 연산 한 번에 그 결과를 얻을 수 있다는 뜻입니다.

1장을 유심히 봤다면 “뭔가 이상하다”라는 생각이 들 것입니다. 왜냐하면, 1장에서 Short URL 생성 방법이 DB에 저장된 count를 증가시키는 방법이기 때문입니다. Memcached에서는 count를 Memcached에 저장해두고, 특정 주기 때마다 DB의 값을 count에 저장하는 형태로 처리합니다. 예를 들어서 다음과 같습니다.

1. DB에 count 값을 10000으로 설정합니다.
2. Memcached의 count 값을 0으로 설정합니다.
3. Memcached의 count 값만 증가시키다가 count가 10000의 배수가 되면 DB의 count 값을 기존 값 +10000으로 업데이트합니다.
4. 만약 중간에 장애로 인해서 Memcached의 정보가 사라진다면 기존 1부터 다시 설정합니다.

3.1.6 Memcached의 실행 옵션에 주의하자

일반적으로 Memcached에 대해서 잘 모르는 것 중 하나가 아이템의 크기입니다. 기본적으로 Memcached의 아이템은 Key와 Value를 합쳐서 48bytes입니다. 또한 각 아이템은 Chunk라는 큰 단위로 할당되는데, Chunk의 크기는 1MB입니다. 간단하게 말해서 1이라는 Key에 1이라는 Value를 할당한다면, 우리가 느끼기에는 2bytes밖에 사용하지 않은 것 같지만 실제로는 48bytes를 사용했다는 뜻입니다. 일반적인 환경에서는 아이템의 크기를 정확히 알 수 없으므로 여유를 가지는 것이 좋습니다. 하지만 서비스에서 사용하는 특성에 따라서 Key와 Value를 합쳐서 48bytes 이하라면, 이 값을 줄여서 메모리 사용량을 줄일 수 있습니다. 이 값을 시작 옵션에서 설정할 수 있습니다. 또한 Memcached의 경우 메모리 사용량이 넘치면 LRU로 이전 데이터를 지우게 되는데, 메모리를 모두 사용하면 에러가 발생하게 설정할 수도 있습니다. 다음 표는 Memcached를 실행할 때 추가할 수 있는 옵션입니다.

표 3-4 Memcached의 실행 옵션

옵션	내용
-M	지정된 메모리 이상의 데이터가 들어오면 LRU 정책으로 기존에 들어와 있던 데이터를 삭제하는데, -M 옵션을 사용하면 기존 내용을 지우지 않고 에러를 리턴합니다.
-c (-c 1024)	접속할 수 있는 최대 커넥션 수를 지정합니다. 기본값은 1,024입니다.
-f (-f 1.25)	증가할 Chunk 크기 비율을 지정합니다. Memcached에서는 데이터를 Slab 단위로 사용합니다. 즉 아이템 하나를 사용하는 최소 공간이 있는데, 이 크기보다 큰 값이 들어오면 지정된 factor 크기만큼 증가시키면서 데이터가 들어갈 공간을 만들게 됩니다. 기본값은 1.25입니다.
-n (-n 48)	아이템 하나가 사용할 최소 크기를 지정합니다. '키 + 값 + 속성 값'의 최소 크기가 됩니다. 기본적으로 48로 지정되어 있습니다.
-D (-D :)	Key에서 prefix를 구분할 수 있는 구분자를 지정합니다. 기본적으로 ':'로 지정되어 있으며, 이는 stats 명령으로 볼 수 있습니다.
-t (-t 4)	프로토콜 파싱에 사용할 스레드의 수를 지정합니다. 기본적으로 네 개로 지정되어 있습니다. 다만, 프로토콜 파싱에만 사용하고 데이터를 읽고, 저장하는 부분은 뮤텍스 ⁰⁸ 로 동기화되어서 하나만 접근되도록 동기화됩니다.
-l (-l 1mb)	Memcached에서 사용하는 기본 Slab은 1MB입니다. 이 크기를 바꿀 수 있는 옵션입니다. 최소 1KB, 최대 128MB까지 지정할 수 있습니다. 이 값에 따라서 전체적인 Memcached 성능이 결정되므로 신중하게 사용해야 합니다.

-I와 -f 옵션에도 주의를 기울여야 합니다. 흔히 Memcached는 1MB의 데이터가 최대로 알고 있습니다. Memcached는 메모리 관리를 내부적으로 Chunk라는 것을 만들고 여기에 아이템을 할당해서 처리합니다. 그런데 이 Chunk의 크기가 기본적으로 1MB로 지정되어 있습니다.

08 OS에서 제공하는 동기화 객체

그림 3-9 Memcached 내에서의 메모리 사용



-I 옵션은 이 Chunk의 크기를 변경할 수 있습니다. 그리고 -f 옵션은 최초 아이템이 48bytes로 할당되었을 때 Chunk 안의 크기를 어떻게 증가시킬지에 대한 팩터입니다. 즉 기본값이 1.25이니 처음에 48, 60, 75 순으로 크기를 점점 늘려가면서 할당하게 됩니다. 이 값 역시 잘 조정하면 성능이 향상됩니다.

3.1.7 Repcached를 이용한 Memcached 리플리케이션을 살펴보자

Memcached에 대해서 이야기하다 보면 거의 나오는 질문이 있습니다. 그 질문은 “Memcached를 리플리케이션하려면 어떻게 해야 하나요?”라는 것입니다. Memcached 같은 메모리 캐시 서버의 최대 단점은 전원이 내려가면 메모리의 내용이 모두 사라진다는 것입니다. 주목적이 캐시이므로 상관이 없어야 하는데, 캐시로 사용하다 보면 캐시가 데이터스토어이기를 바라게 됩니다. 항상 주변에 “Memcached는 캐시로 사용해야지 데이터스토어로 생각하지 마세요. 모든 데이터는 날아가더라도 다시 복구할 수 있게 해야 합니다. Memcached는 그런 용도가 아닙니다”라고 말하지만, 사실 리플리케이션되면 정말로 편합니다.

Repcached는 이런 목적으로 나온 것으로 Memcached를 Master/Master 리플리케이션되도록 만든 프로그램입니다. 다만, 절대로 데이터의 불일치가 일어나서는 안 되는, 은행의 이체나 계좌 정보를 저장하는 등의 중요한 곳에는 사용하지 마세요(해당 이유는 뒤에서 설명합니다). 그리고 Memcached는 libevent 2.0.x 버전도 사용이 가능하지만, Repcached는 무조건 libevent 1.4.x 버전을 사용해야 빌드됩니다.

표 3-5 Repcached의 장단점

구분	내용
장점	Master/Master 리플리케이션을 지원한다.
단점	Memcached 1.2.8만 지원한다(Memcached의 최신 버전은 1.4.13이다). Master/Master 리플리케이션이 가능하다. 한 번에 한 대만 리플리케이션된다.

그림 3-10과 같은 구조로 서버가 연결되어 있고, 각 클라이언트가 1과 2를 서로 다른 서버에 저장한다면, 그림 3-11처럼 자동으로 리플리케이션됩니다. 또한 서버 1대가 장애가 발생해서 제외된 후 복구되면, 다른 서버에서 모든 데이터를 가져와서 동일한 상태로 만듭니다.

Repcached를 실행하는 방법은 다음과 같습니다. -x(주소 지정)와 -X(포트 지정)을 이용하여 리플리케이션할 서버와 포트를 지정할 수 있습니다.

```
memcached -m 64 -x 192.168.0.101 -d  
memcached -m 64 -x 192.168.0.102 -d
```

간단하게 실제로 데이터가 복제되는지 확인해 보겠습니다.

```
charsyam@charsyam-1v63:~/repo/memcached-1.2.8-repcached-2.2.1$ telnet  
192.168.0.101 11211  
Trying 192.168.0.101...  
Connected to 192.168.0.101.  
Escape character is '^]'.  
set k 0 0 1  
1  
STORED  
get k  
VALUE k 0 1  
1  
END  
charsyam@charsyam-1v63:~/repo/memcached-1.2.8-repcached-2.2.1$ telnet  
192.168.0.102 11211  
Trying 192.168.0.102...  
Connected to 192.168.0.102.  
Escape character is '^]'.  
get k  
VALUE k 0 1  
1  
END
```

그림 3-10 Repcached 구조

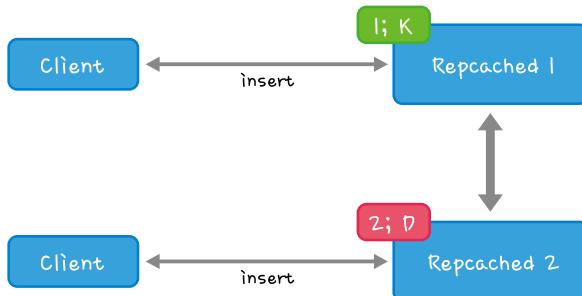
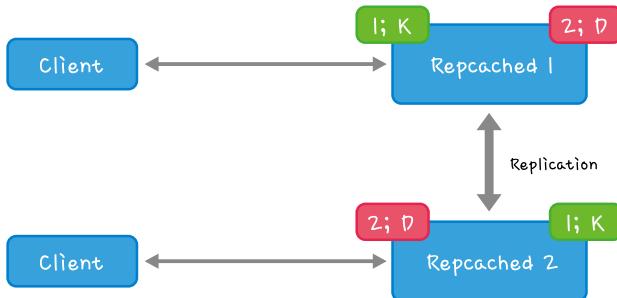


그림 3-11 자동으로 리플리케이션되는 Repcached



여기서 잠깐 Repcached는 어떻게 리플리케이션을 구현했을까?

Memcached의 소스를 보면 명령어를 파싱하는 부분과 데이터를 가져오는 부분으로 나눠져 있습니다. process_command에서 명령어를 파싱하고 set, add 등의 명령이면 process_update_command를 호출하고 다시 결과적으로 complete_nread를 호출하게 됩니다. 그리고 complete_nread가 다음과 같이 구현되어 있습니다.

```
ret = store_item(it, comm);
if (ret == 1) {
    #ifdef USE_REPLICATION
    {
        if( c != rep_conn ){
            replication_call_rep(ITEM_key(it), it->nkey);
        }
        out_string(c, "STORED");
    }
    #else
        out_string(c, "STORED");
    }
}
```

즉, 자신의 아이템을 store_item을 통해서 저장하고, 이것이 성공하면 리플리케이션용 서버에 연결되어 있으면 replication_call_rep을 호출합니다.

문제는 replication_call_rep에 대한 호출이 실패할 수도 있다는 것입니다. 이 경우, 해당 리턴값 등을 체크하지 않아서 리플리케이션 서버에 데이터 불일치가 발생할 수 있습니다. 따라서 절대로 틀리면 안 되는 데이터에는 repcached를 사용하면 안 됩니다.

3.2 Redis

Redis⁰⁹는 Memcached와 비슷한 분산 캐시의 일종입니다. Memcached의 발전은 지지부진한 반면에, Redis는 엄청난 속도로 발전하고 있으며 현재는 VMWare의 지원을 받고 있습니다.

Redis는 Memcached에서 제공하지 않은 여러 기능을 제공합니다. 다음은 Redis의 특징을 정리한 것입니다.

표 3-6 Redis의 특징

특징	내용
Data Types	List, Sorted Set, Hash 등의 자료구조를 제공합니다. Collection을 사용할 수 있어서 개발에 편의성을 제공합니다.
Replication	Master/Slave로 사용할 수 있는 리플리케이션 기능을 제공합니다.
Persistence	RDB라는 현재 메모리의 Data Set에 대해서 Snapshot을 만들 수 있는 기능을 제공합니다.
Pub/Sub	Redis를 Publisher/Subscribe 형태로 이용할 수 있는 기능을 제공합니다.

Redis는 Memcached에 비해서 많은 기능과 명령을 제공합니다. 명령에 대한 자세한 내용은 '<http://redis.io/commands>'에서 확인하시기 바랍니다.

09 <http://redis.io>

3.2.1 Redis 설치하기

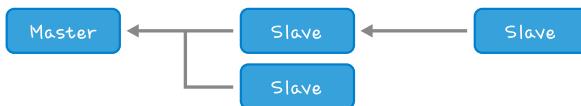
Memcached는 libevent에 대한 의존성이 있었지만, Redis는 그런 의존성이 없습니다. 현재 Redis는 2.4.9 버전까지 나와 있습니다(2012/03/28일 기준). 다음 순서로 Redis를 설치합니다.

```
root@charsyam ~/src # wget  
http://redis.googlecode.com/files/redis-2.4.9.tar.gz  
root@charsyam ~/src # tar zxvf redis-2.4.9.tar.gz  
root@charsyam ~/src # cd redis-2.4.9  
root@charsyam ~/src/redis-2.4.9 # ./configure  
root@charsyam ~/src/redis-2.4.9 # make  
root@charsyam ~/src/redis-2.4.9 # make install
```

3.2.2 Redis의 정보를 리플리케이션하자

Redis는 Memcached와 다르게 서버 기능으로 리플리케이션을 제공합니다. 슬레이브에 있는 Redis Conf에서 마스터 서버만 지정해주면 리플리케이션을 제공합니다. 계층적 리플리케이션 설정도 가능합니다.

그림 3-12 Redis의 계층적 리플리케이션 구조



Master 서버를 지정하는 방법은 redis.conf에 slaveof 구문을 이용하면 됩니다.

```
slaveof <masterip> <masterport>
```

이렇게 설정해두는 것만으로 Redis 서버가 동작하면서 설정된 IP 서버의 슬레이브로 동작합니다. 이때 슬레이브로 동작하는 서버도 데이터 변경이 가능한데, 이 데이터는 마스터 서버로 전달되지 않습니다. 그리고 Redis의 리플리케이션은 master/slave replication만 지원합니다.

여기서 깐 Redis는 master/master Replication 설정이 안 되나요?

결론부터 말하자면 master/master Replication은 동작하지 않습니다. 클라이언트는 개별 서버에 접속이 가능하지만, Redis 서버끼리 master/slave로 연결되지 않습니다. 다음과 같은 오류가 발생합니다.

```
[7567] 25 Mar 02:22:14 * Connecting to MASTER.  
[7567] 25 Mar 02:22:14 * MASTER <-> SLAVE sync started  
[7567] 25 Mar 02:22:14 * Non blocking connect for SYNC fired the event.  
[7567] 25 Mar 02:22:14 # MASTER aborted replication with an error: ERR Can't  
SYNC while not connected with my master  
[7567] 25 Mar 02:22:14 - Accepted 127.0.0.1:33609  
[7567] 25 Mar 02:22:14 - Client closed connection
```

마스터에 장애가 발생하면 클라이언트의 Redis 서버 설정을 슬레이브로 변경해주어야 합니다. 이때 슬레이브 노드에 접속해서 다음 명령을 수행해야 합니다.

SLAVEOF NO ONE

해당 명령을 수행하면 슬레이브로 동작하던 노드는 더 이상 마스터 서버와 싱크를 맞추지 않고 독자적으로 동작하게 됩니다.

3.2.3 RDB와 AOF를 꼭 사용하자

Redis에는 장애를 대비하기 위한 기능으로 RDB와 AOF를 제공합니다. RDB는 현재의 메모리 상태의 Snapshot을 만들어두는 것이고, AOF^{Append Only File}는 일종의

Jounaling 파일 역할을 합니다.

RDB는 현재의 메모리 정보를 가지고 있는 파일입니다. 일정 시기와 명령 개수가 쌓였을 때마다 남길 수 있습니다. Conf를 살펴보면 설정은 다음과 같습니다.

```
save 900 1
save 300 10
save 60 10000
```

'save <시간(초)> <변경된 Key 개수>'라는 의미입니다. 해석하면 Key가 하나 변경됐으면 900초, 즉 15분 뒤에 dump를 생성하라는 의미입니다. 밑에는 10개가 변경되면 300초, 10,000개가 변경됐으면 60초마다 dump를 생성하라는 뜻이 됩니다.

RDB를 압축할 것인지 여부와 RDB 파일의 이름도 지정할 수 있습니다.

```
rdbcompression yes
dbfilename dump.rdb
```

AOF 파일을 살펴보면 Key의 변경에 사용된 명령이 들어 있습니다. 즉 업데이트 관련 내용이 들어 있습니다. 다음은 AOF 파일을 열어본 것입니다.

```
*4
$4
ZADD
$6
myzset
$1
1
$5
```

```
"one"  
*4  
$4  
ZAdd  
$6  
myzset  
$1  
2  
$5  
"two"
```

Redis가 시작하면서 자동으로 해당 내용을 처리하므로 자세한 파일 구조를 알 필요는 없습니다.

AOF의 중요한 설정은 다음과 같습니다. AOF 사용 여부를 결정하는 것이 appendonly입니다. No로 설정하면 사용하지 않고, yes라고 설정하면 아래의 appendfilename에서 지정한 이름으로 생성됩니다. 지정하지 않으면 ‘appendonly.aof’로 지정되게 됩니다.

appendfsync는 얼마나 디스크에 자주 AOF 파일을 쓸지에 대한 설정입니다. 기본적으로 AOF 내용은 메모리에 저장하고 주기적으로 쓰입니다(everysec일 경우). always로 설정하면 변경될 때마다 디스크와 동기화를 하게 됩니다. 이 경우, 안정성은 높아지지만 속도가 느려지는 단점이 있습니다.

```
#appendonly no  
appendonly yes  
  
# The name of the append only file (default: "appendonly.aof")  
# appendfilename appendonly.aof
```

```
# appendfsync always  
appendfsync everysec
```

여/기/서/점/깐 AOF와 RDB를 이용하면 디스크를 사용하는데, Redis의 장점이 사라지지 않나요?

지금까지 디스크에 접근하면 속도가 느려지므로 Redis나 Memcached는 메모리만 사용한다고 말을 했습니다. 그런데 Redis의 AOF나 RDB는 write 주기를 조정할 수는 있지만, 결국은 파일에 데이터를 기록하게 됩니다. 이렇게 되면, 속도가 떨어지지 않을까라는 의문이 들것입니다.

결론부터 말하자면 RDB와 AOF는 성능을 저하시킵니다. 그래서 실제로 마스터는 서비스만 하고 슬레이브에서 AOF와 RDB를 이용해서 백업하는 형태로 사용합니다(현재 instagram에서 이런 식으로 사용하고 있습니다).

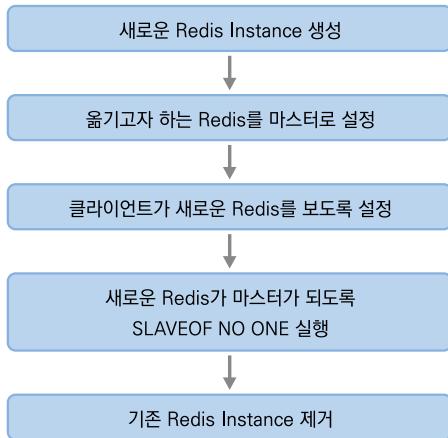
그림 3-13 Redis 슬레이브에서 백업하는 구조



3.2.4 Redis 서버를 데이터 유실 없이 이전해보자

서비스를 운영하다 보면 장비의 업그레이드나 장애 등으로 장비를 교체해야 할 경우가 생기기도 합니다. 이때 데이터 유실 없이 이전하는 방법에 대해서 알아보겠습니다. 순서는 리플리케이션을 받는 것과 비슷합니다.

그림 3-14 새로운 Redis 서버로 데이터 유실 없이 옮기는 방법



3.2.5 비어 있는 마스터와 리플리케이션하지 않게 주의하자

Redis를 사용하면서 의외로 잘 모르는 것 중에 하나가 초기에 마스터와 슬레이브 간의 동기화 과정입니다. 이 부분을 잘못 알고 있으면 큰 참사가 일어날 수 있습니다. 앞에서 마스터에서는 서비스를 하고 슬레이브에서 실제로 AOF, RDB를 통해서 Snapshot을 뜨는 방법에 대해서 얘기했습니다. 만약 마스터가 장비의 이상으로 장애가 발생하여 장비를 수리 후에 다시 서비스에 투입하면 어떤 일이 벌어질까요?

“데이터가 모두 사라질 수 있습니다.”

사실 위에서 말한 장애처리 과정을 제대로 거쳤다면 크게 문제될 것이 없지만, SLAVEOF NO ONE이라는 명령을 수행하지 않았다면 모든 데이터가 사라집니다. 이 문제는 Redis의 마스터와 슬레이브 간의 초기 동기화 과정 때문입니다.

Redis의 소스를 확인해보면 emptyDb라는 함수가 있습니다. 다음 코드는 Redis에서 모든 메모리의 데이터를 지웁니다.

```

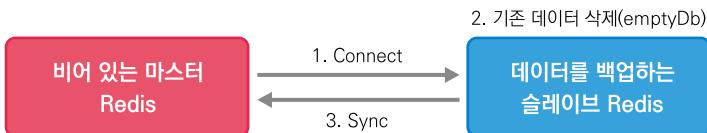
/* Empty the whole database */
long long emptyDb() {
    int j;
    long long removed = 0;

    for (j = 0; j < server.dbnum; j++) {
        removed += dictSize(server.db[j].dict);
        dictEmpty(server.db[j].dict);
        dictEmpty(server.db[j].expires);
    }
    return

```

그런데 이 코드가 슬레이브로 설정되어 있으면 마스터와의 연결을 감지하고 있다가 마스터가 재시작할 때마다 호출되며, 싱크Sync를 맞춥니다. 그런데 위에서 설명한 구조에서 마스터는 재시작하면 메모리에 아무런 데이터가 없는 상태입니다. 이렇게 되면 슬레이브도 먼저 자신의 데이터를 지우고, 리플리케이션할 데이터가 없으므로 마스터에도 데이터가 없어집니다. 데이터가 유실되는 상황이 벌어지는 것입니다.

그림 3-15 비어 있는 마스터와 리플리케이션이 되어서 데이터가 유실되는 경우



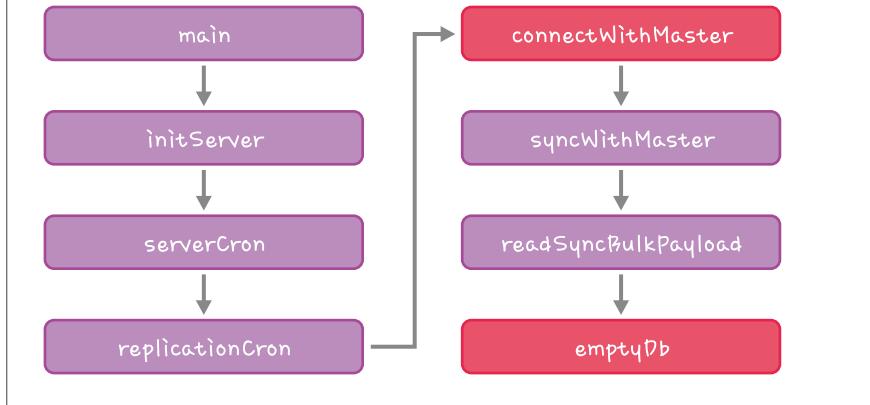
이를 방지해 주는 것이 SLAVEOF NO ONE 명령입니다. 일단 마스터/슬레이브 관계를 끊어주므로 더 이상 리플리케이션을 받지 않으며, 마스터가 재연결하더라도 데이터를 재동기화하는 과정이 없습니다. 주의할 점은, 이것은 버그가 아니라

데이터 동기화를 위한 방안이므로 사람이 주의하는 수밖에 없다는 것입니다. 그러면 실수할 수 있으므로 초기 동기화와 리플리케이션의 어떻게 일어나는지 등에 대해서 자세히 알아두어야 합니다.

여/기/서/잠/깐 emptyDb는 어떤 순서로 호출될까요?

오픈소스의 장점은 그 소스를 볼 수 있다는 것입니다(다만 분석이 어려울 뿐이지요). 그리고 오픈소스를 사용하는 개발자는 어느 정도 해당 소스를 이해해야만 합니다. 그래야만 장애나 문제에 대응이 가능합니다. 누군가 해결해 주기만 기다리면 발전은 없습니다. 간단하게 emptyDb가 어떤 순서로 호출되는지 그림을 통해서 살펴보겠습니다. Redis 소스에서 main이 있는 파일은 “redis.c”입니다. 흐름을 정리하면 다음과 같습니다.

그림 3-16 Redis에서 emptyDb가 호출되는 흐름



4 | 분산 캐시 솔루션을 사용할 때 주의할 점

Memcached와 Redis를 사용하다 보면 공통으로 주의해야 할 점들이 있습니다. 이 장에서는 이런 부분에 대해서 알아보겠습니다.

4.1 메모리 사용량에 주의하자

우리에게 메모리가 16GB인 장비가 있다면, 어느 정도의 메모리를 캐시 솔루션에 할당하는 것이 좋을까요? 16GB를 모두 할당하면 될까요? 여기서 고민해야 할 것 이 있습니다. 메모리를 캐시 솔루션에서만 사용하는 것일까요? 그리고 캐시 솔루션에 할당된 메모리만 사용할까요? 이런 것을 생각해보면 메모리 사용에 대해서 길이 보입니다.

일단 첫 번째 질문부터 생각해 보겠습니다. 해당 장비에는 운영체제가 실행되고 있 으며, 그 외에 다른 프로그램도 실행되고 있을 것입니다. 모니터링 프로그램이나 배치 작업 등 프로그램이 어느 정도의 메모리를 사용합니다.

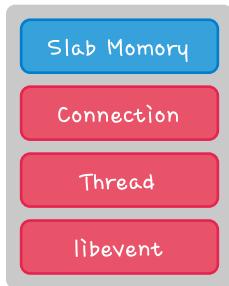
그림 4-1 일반적인 OS에서의 메모리 사용



이제 두 번째 질문을 다시 생각해 보겠습니다. Memcached에서 -m 옵션으로 4096으로 설정하면 일반적으로 메모리 4GB만 사용한다고 생각합니다. 여기서 주 의해야 할 것은 4GB는 데이터를 저장하기 위해서 할당되는 메모리 크기라는 것

입니다. Key, Value, Flags 등을 저장하는 Chunk를 할당하는 Slab 메모리에만 4GB를 사용하고, Memcached에서 추가로 Client의 접속을 관리하는 메모리, 스레드가 사용하는 메모리, libevent 내부에서 사용하는 메모리 등 실제로 사용하는 것은 ‘4GB+알파’의 메모리를 사용하게 됩니다. 경우에 따라서 이 알파가 클 수도 있습니다. Redis도 마찬가지입니다. 따라서 메모리를 빡빡하게 할당하면 스왑핑 swapping¹⁰이 발생해서 전체 성능이 확 떨어집니다.

그림 4-2 Memcached에서의 메모리 사용



그렇다면 메모리를 어느 정도 할당하는 게 좋을까요? 정답은 없지만, 일단 운영체제와 모니터링용 메모리로 기본적으로 3~4GB 이상 남겨두어야 합니다. 그리고 캐시 솔루션에서 사용하는 여분의 메모리가 필요합니다. 일반적으로 전체 메모리의 60~70% 정도를 사용하는 것이 안정적입니다. 실제로 Craigslist에서는 32GB 장비에서는 6GB 메모리를 사용하는 Redis를 네 개씩 띄워서 사용하고 있습니다. 즉 “6GB × 4 = 24GB”이고 전체적으로 8GB 가량 남겨두고 있습니다.

4.2 성능이 갑자기 떨어지면 메모리 스와핑을 의심하자

캐시 솔루션을 사용하다 보면 갑자기 응답속도가 떨어지는 시기가 발생합니다. 어느 정도 여유분의 메모리 용량을 설정해 두지만, 통계값을 얻기 위해서 무거운 프

10 메모리 내의 데이터 페이지나 세그먼트를 교체하는 것입니다.

로그램을 수행해야 할 수도 있고, 메모리가 필요해서 캐시 솔루션이 사용하는 메모리에서 스왑이 발생하는 경우도 생깁니다. 이 경우 두 가지 해결책이 있습니다.

첫 번째는 프로세서를 종료시킨 후에 재시작하는 것입니다. Memcached는 재시작할 때 데이터가 모두 사라지므로 사용할 수 없지만, Redis는 RDB를 이용해서 데이터 유실 없이 재시작할 수 있습니다. 두 번째 방법은 메모리를 실제로 더 이상 사용하지 않는다면 메모리 캐시를 전부 삭제해 스왑을 제거하는 것입니다. Root 사용자가 다음과 같이 실행하면 됩니다.

```
echo 3 > /proc/sys/vm/drop_caches
```

echo 1로 하면 pagecache만 삭제되며, echo 2로 하면 inode 캐시가 삭제됩니다. echo 3으로 하면 이 두 가지 캐시가 모두 삭제됩니다. 다만 현재의 캐시가 모두 사라지므로 일순간 부하가 늘어날 수 있습니다. 적절히 테스트하면서 서비스에 적합한 캐시 크기를 결정하는 게 중요합니다.

4.3 Memcached와 Redis의 flush는 어떻게 다를까?

캐시 솔루션을 사용하다 보면 모든 데이터를 지워야 하는 경우가 생깁니다. Memcached에서는 ‘flush_all’이라는 명령, Redis에서는 ‘FLUSHDB’라는 명령으로 데이터를 모두 지울 수 있습니다. 그런데 이 명령의 동작이 조금 다릅니다.

Memcached는 데이터를 읽거나 쓰는 도중에 flush_all 명령을 받으면 바로 수행이 되는 것처럼 보입니다. 그런데 Redis는 데이터 양이 많으면 다른 명령들이 모두 일순간 대기하게 됩니다. Memcached와 Redis, 둘 다 싱글 스레드처럼 동작하므로 인해 이런 대기는 당연한 것입니다. 그런데 왜 Memcached는 대기가 없는 것처럼 보이고 Redis는 대기할까요?

다음 예제를 실행한 후 Redis 서버에 접속해서 FLUSHDB를 해보면 대략 1.1초 정도 GET이나 SET 명령이 대기하는 것을 볼 수 있습니다.

```
import redis

r = redis.StrictRedis( host='127.0.0.1', port=1212 )
for i in xrange(0,4000000):
    r.set( str(i), str(i) )
    print i
```

왜 이런 차이가 발생할까요? 그것은 Memcached가 lazy delete라는 정책을 이용하기 때문입니다. Memcached는 flush_all 명령이 실행되었을 때 해당 시간을 기록한 다음 실제로 GET 명령이 들어온 시점에 삭제합니다. 이제 실제로 소스에서 어떻게 동작하는지를 살펴보겠습니다. 먼저 Redis부터 살펴보겠습니다 (Redis 2.4.13 버전을 기준으로 테스트했습니다). Redis의 db.c 파일의 198라인에 ‘flushdbCommand’라는 함수가 있습니다. 이 함수는 다음과 같습니다.

```
void flushdbCommand(redisClient *c) {
    server.dirty += dictSize(c->db->dict);
    signalFlushedDb(c->db->id);
    dictEmpty(c->db->dict);
    dictEmpty(c->db->expires);
    addReply(c,shared.ok);
}
```

dictEmpty 함수에서 실제로 삭제가 일어납니다. dictEmpty 함수는 다음과 같습니다(Dict.c의 584라인에 있습니다).

```
void dictEmpty(dict *d) {
    _dictClear(d,&d->ht[0]);
    _dictClear(d,&d->ht[1]);
    d->rehashidx = -1;
    d->iterators = 0;
}
```

다시 `_dictClear` 함수를 살펴보겠습니다(Dict.c의 358라인에 있습니다). 소스를 보면 루프를 돌면서 하나씩 제거하는 것을 볼 수 있습니다. 만약 데이터가 2백만 개가 있다면 2백만 번 삭제합니다.

```
int _dictClear(dict *d, dictht *ht)
{
    unsigned long i;

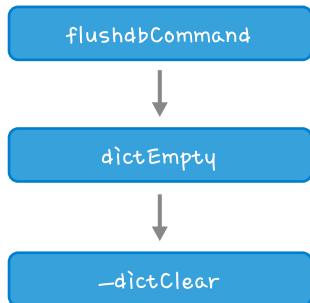
    /* Free all the elements */
    for (i = 0; i < ht->size && ht->used > 0; i++) {
        dictEntry *he, *nextHe;

        if ((he = ht->table[i]) == NULL) continue;
        while(he) {
            nextHe = he->next;
            dictFreeEntryKey(d, he);
            dictFreeEntryVal(d, he);
            zfree(he);
            ht->used--;
            he = nextHe;
        }
    }
    /* Free the table and the allocated cache structure */
}
```

```
    zfree(ht->table);
    /* Re-initialize the table */
    _dictReset(ht);
    return DICT_OK; /* never fails */
}
```

간단히 정리하면 다음과 같은 순서로 흘러갑니다.

그림 4-3 Redis의 삭제 흐름도



이제 Memcached의 소스를 살펴보겠습니다. flush_all이 되는 부분과 GET으로 아이템을 가져가는 부분을 살펴보겠습니다. Memcached는 1.4.13 버전을 기준으로 테스트했습니다.

Memcached.c의 3290라인을 살펴보겠습니다. 이 코드는 static void process_command(conn *c, char *command)라는 함수 안에 있습니다. 중요한 것은 settings.oldest_live를 셋팅하는 부분입니다. 여기서 flush_all 명령이 들어온 시간을 기록합니다.

```
time_t exptime = 0;

set_noreply_maybe(c, tokens, ntokens);
```

```

pthread_mutex_lock(&c->thread->stats.mutex);
c->thread->stats.flush_cmds++;
pthread_mutex_unlock(&c->thread->stats.mutex);

if(ntokens == (c->noreply ? 3 : 2)) {
    settings.oldest_live = current_time - 1;
    item_flush_expired();
    out_string(c, "OK");
    return;
}

exptime = strtol(tokens[1].value, NULL, 10);
if(errno == ERANGE) {
    out_string(c, "CLIENT_ERROR bad command line format");
    return;
}

/*
If exptime is zero realtime() would return zero too, and
realtime(exptime) - 1 would overflow to the max unsigned
value. So we process exptime == 0 the same way we do when
*/
if (exptime > 0)
    settings.oldest_live = realtime(exptime) - 1;
else /* exptime == 0 */
    settings.oldest_live = current_time - 1;

item_flush_expired();
out_string(c, "OK");
return;

```

이제 실제로 get할 때 어떻게 될 것인가를 보겠습니다. items.c의 483라인의 do_item_get이라는 함수가 호출됩니다.

```
if (settings.oldest_live != 0 && settings.oldest_live <= current_time &&
    it->time <= settings.oldest_live) {
    ...
    ...
}
```

settings.oldest_live가 current_time(현재시간)보다 적고, 0이 아니라는 것은 한 번 이상 flush_all이 실행됐다는 뜻입니다. 해당 초기값은 0입니다. 그리고 발견된 item의 시간과 oldest_live를 비교하면, 해당 Key가 flush_all 이전에 들어온 것인지 이후에 들어온 것인지 판단할 수 있습니다. 그래서 Memcached에서는 실제로 아이템이 지워지는 시간이 거의 없는 것처럼 느껴지는 것입니다.

그러나 뭐든지 장점과 단점이 있습니다. Memcached는 실제 키에 대해 GET 연산이 들어온 후에 삭제하므로, GET 실행이 기존보다 조금 느려집니다. Memcached는 처음부터 expire_time 등을 이용하는 cache 솔루션이 목적이었고, Redis는 메모리 스토어로 발전해서 그런 차이가 생긴 것 같습니다.

여/기/서/잠/깐 Redis는 싱글 스레드인데 어떻게 백그라운드 백업 명령어인 BGSAVE가 있나요?

Redis는 싱글 스레이드임에도 불구하고 BGSAVE라는 백그라운드로 RDB를 생성하는 명령이 있습니다. 어떻게 구현한 것일까요? Rdb.c의 499라인을 보면 자식 프로세스를 fork로 생성하고 여기서 실제로 동작하는 것을 볼 수 있습니다.

Memcached의 flush_all은 파라미터로 delay 시간을 줄 수 있습니다. 여기서 값을 주지 않으면 현재 시간을 기준으로 모두 삭제되고, delay 시간을 주면 delay 시간 이후에 삭제됩니다. 이 기능은 잘 알려지지 않았고, 거의 사용하지 않습니다. PHP와 파이썬의 라이브러리는 flush_all의 경우 파라미터를 받지 않게 구현되어 있습니다.

그럼 실제로 어떤 경우에 데이터가 살아날까요? flush_all과 flush_all [delay time]이 연달아 오는 경우입니다. 즉 다음과 같은 경우에 발생합니다.

```
charsyam@ubuntu:~$ telnet 0 11211
Trying 0.0.0.0...
Connected to 0.
Escape character is '^].
set h1 0 0 2
h1
STORED
set h2 0 0 2
h2
STORED
flush_all
OK
get h1
END
flush_all 10
OK
get h2
VALUE h2 0 2
h2
END
>> 10초 뒤
get h2
END
```

중간에 h2가 살아난 것을 볼 수 있습니다. 이것은 delay 시간으로 인해서 settings.oldest_live 값이 현재 current_time보다 일시적으로 커져서 그 시간 동안 데이터가 다시 존재하는 것처럼 보이고 get 연산 시에 flush되지 않아서 발생하는 문제입니다. 그래서 위의 예에서 10초가 지나면 h2가 사라집니다.

이 경우를 flush_all에만 사용한다면 문제가 없으므로 실제로 발생한 경우는 없을 듯합니다. 만약 delay 후에 flush_all을 꼭 하고 싶다면 cron 등으로 해당 시간에 flush_all을 실행시키도록 해주면 됩니다.

5 | Redis를 이용한 간단한 Short URL 서비스 구축하기

Redis를 이용해서 간단한 Short URL 서비스를 구축해 보겠습니다. 그리고 Redis의 Sorted Set 사용 예제를 간단히 살펴보겠습니다(Memcached는 사용 방법이 간단하므로 Short URL 서비스를 쉽게 구축할 수 있습니다).

5.1 Read를 Redis로 대체하자

1장에서 설명했듯이 Short URL 서비스에는 크게 두 가지 동작이 있습니다.

표 5-1 Short URL에서 제공해야 하는 두 가지 기능

동작	타입	내용
Short URL 생성	Write	Real URL을 받아서 Short URL을 생성합니다.
Real URL 전달	Read	Short URL을 받으면 연결된 Real URL로 만들어줍니다.

1장에서 설명한 예제의 Read 부분을 Redis를 이용해 수정하겠습니다. 이를 위해서 예제의 Short URL를 저장하는 부분과 real_url을 읽는 부분에 캐시 동작을 추가할 것입니다. 먼저 Short URL을 저장하는 부분에 Short URL을 DB 저장 후에 캐시에 저장하는 로직이 필요합니다. Real URL을 가져가는 부분에서는 캐시에서 읽어오는 로직이 필요합니다.

캐시에서 사용할 Key와 Value는 어떤 것이 좋을까요? Key는 사용자에게 요청받는 값, Value는 사용자에게 전달하는 값이 그대로 들어가는 게 좋습니다. Short URL 서비스는 Short URL을 요청받으면 Real URL을 그대로 전달하면 되니, 캐시를 사용하기에 상당히 좋은 서비스입니다.

그림 5-1 Read 캐시를 적용하는 Short URL 생성 로직



그림 5-2 Read 캐시를 적용하는 Real URL 전달 로직



Key → Short Url 주소 , Value → Real Url

1장의 예제에서 수정한 부분은 다음과 같습니다.

- ① Redis 관련 변수가 추가됩니다.

```
global g_redis

def connect_redis():
    return redis.StrictRedis(host=REDIS_ADDRESS, port=REDIS_PORT, db=0)

if __name__=='__main__':
    g_redis = connect_redis()
```

② Short URL 저장 로직에 Redis에 저장하는 코드를 추가합니다.

```
def create_short_url(cursor, url):
    real_url = urllib.quote(url.encode('utf8'), '/')
    try:
        count = get_count(cursor, real_url)
    except:
        raise Exception("Error: Duplicate url: %s"%url)

    short_url = create_short_url_data( DIVISION_SEED, count )
    save_short_url( cursor, short_url, real_url )
    g_redis.set( short_url, real_url )
    return short_url
```

③ 마지막으로 Real URL을 전달하는 부분에서 Redis를 먼저 읽는 코드를 추가합니다.

```
def get_realurl(short_url):
    url = g_redis.get(short_url)
    if url:
        return url
```

```
g_cursor.execute( "select real_url from tbl_shorturl where  
short_url=%s"%(short_url) )  
if( g_cursor.arraysize > 0 ):  
    value = g_cursor.fetchone()[0]  
    g_redis.set( short_url, value )  
else:  
    flash("Error: No url exists")  
  
return value
```

이제 대부분의 Read 요청을 Redis가 처리하므로 Read 부하가 크게 줄어들고, DB에 Read 요청이 아주 일정하게 들어오게 될 것입니다.

5.2 Short URL 생성 로직도 Redis로 대체하자

Read의 부하를 줄이더라도 서비스의 요청이 많아지면 어느 순간부터 DB의 부하는 다시 높아집니다. 이렇게 되면 DB의 부하를 줄이는 방법을 찾아야 하는데, 이때 많이 생각하는 방법이 하드디스크를 SSD로 바꾸거나, 파티셔닝하는 것입니다. 물론 한 계에 다다르면 파티셔닝을 해야겠지만, 쓰기에 캐시를 도입하면 부하를 많이 줄일 수 있습니다. 항상 “메모리는 디스크보다 빠르다”라는 사실을 명심해야 합니다.

Short URL은 DB의 AUTO_INCREMENT 값을 이용하고, 하나의 Short URL을 생성할 때마다 디스크에 저장하므로 고유한 값이 겹치지 않을까 걱정할 수도 있습니다. 다행히 Redis나 Memcached에는 고유한 값을 겹치지 않게 증가시켜주는 기능이 있습니다. Redis에서는 ‘INCR’이라는 명령을 이용하면 값이 계속 증가되는 수를 만들 수 있습니다. 이를 이용하면 간단하게 Short URL을 생성할 수 있습니다.

get_count와 save_short_url 함수만 다음과 같이 수정하면 간단하게 DB를 쓰지 않고 Redis의 기능만 사용하는 버전이 됩니다.

```
def get_count(cursor, url):
    value = g_redis.incr(REDIS_INCR_KEY)
    return value

def save_short_url( cursor, short_url, real_url ):
    pass
```

장애는 RDB와 리플리케이션 기능을 이용해서 쉽게 복구할 수 있습니다. 실제로 대용량 서비스를 해야 한다면 캐시에 데이터를 올리고 해당 작업의 로그를 남겨서 백그라운드 작업을 합니다. 그리고 해당 내용을 한 번에 배치 작업으로 추가하면 DB에 부하를 거의 주지 않고 서비스를 처리할 수 있습니다.

5.3 Redis의 Sorted Set을 이용한 인기 URL 찾아보기

Redis는 Memcached가 제공하지 않는 collection 기능을 제공합니다. List, Set, Sorted Set 등이 바로 그것입니다. 이 중에서 Sorted Set은 랭킹 서비스를 할 때 유용합니다. 네이버의 실시간 검색 순위 같은 것을 만들 때 Sorted Set을 유용하게 사용할 수 있습니다.

Short URL을 서비스할 때 관리자는 어떤 사이트가 인기 있는지 궁금할 것입니다. 그리고 상위 사이트의 내용을 조사하면 현재 무엇이 유행하는지도 쉽게 알 수 있습니다. 실제로 bit.ly 같은 곳은 현재의 트렌드 분석도 함께하고 있습니다.

여기서는 Sorted Set을 이용해서 상위 다섯 개의 인기 URL을 보는 기능을 추가해 보겠습니다. Sorted Set은 score를 기준으로 내부적으로 정렬합니다. 로직을 살펴보면 다음과 같습니다.

- ① Real URL 전달 요청이 왔을 때 해당 Real URL이 있으면, 이 값으로 score를 1 증가시킵니다. 이렇게 되면 해당 score에 따라서 내부적으로 정렬되는 순서가 바

립니다. 즉 어떤 웹사이트를 가장 많은 사용자가 클릭했는지 알 수 있게 됩니다. score 값을 증가시키기 위해서는 Redis의 zincrby 명령을 사용합니다. 만약 처음에 해당 URL이 없다면 zincrby가 처음 실행될 때 해당 아이템이 생성됩니다.

```
@app.route('/<short_url>')
def redirect_real_url(short_url):
    try:
        value = get_realurl( short_url )
        g_redis.zincrby( REDIS_RANK1, value, 1 )
        return value
    except Exception as inst:
        return str(inst)
```

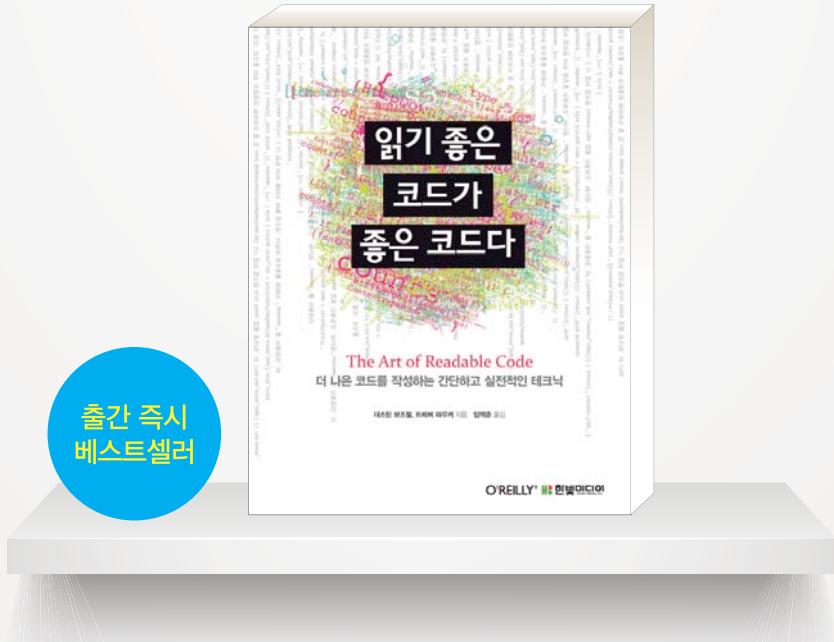
② 이제 실제로 랭킹을 가져오는 일은 더 간단합니다. 실제로 Sorted Set은 score가 작은 값부터 정렬이 되므로, 역순으로 가져오는 zrevrange를 사용하면 됩니다.

```
@app.route('/rank')
def show_info():
    ret = g_redis.zrevrange( REDIS_RANK1, 0, 10 )
    urls = [dict(url=row) for row in ret]
    return render_template('show_rank.html', urls=urls )
```

예제 파일은 “<https://github.com/charsyam/shorturl>”에서 모두 받으실 수 있습니다.

The Art of Readable Code

자신의 코드를 남에게 보여주기가 꺼려집니까?



읽기 좋은 코드가 좋은 코드다 : 더 나은 코드를 작성하는 간단하고 실전적인 테크닉

“이 책을 번역하기로 결정한 과정은 다소 우연이었지만, 번역 과정에서 내가 평소에 생각하던 부분들이 이렇게 책으로 정리되어 나왔다는 사실에 안도감과 고마움을 느꼈다. 내가 지금까지 경험한 바에 의하면, 이 책을 읽지 않아도 좋은, 원래부터 간결하고 효율적인 코드를 작성하는 능력을 가진 프로그래머는 열에 하나에 불과하다. 자신이 그 하나에 속한다는 확신이 없으면, 이 책을 꼭 읽어보기 바란다”

– 임백준, 소프트웨어 개발자 & IT 라이터

더스틴 보즈웰, 트레버 마우커 지음 | 임백준 옮김 | 18,000원

* 한빛미디어 도서구매 인증하세요!

-  **한빛이코인 포인트 적립!**
-  **부가서비스 E-mail 자동 전송!**
-  **구매도서 독자평가 실시!**

혜택1. 도서 구매 시 현금처럼 사용할 수 있는 한빛이코인을 적립해 드립니다.

혜택2. 구매 도서에 대한 정오표, 부록 등 관련자료를 E-mail로 자동 전송해 드립니다.

혜택3. 구매 도서에 대한 독자평기를 실시, 기획에 적극 반영합니다.



인증방법: 한빛미디어 홈페이지▶로그인▶도서 인증

www.hanb.co.kr

* 한빛리더스가 되세요!

한빛리더스(Readers & Leaders)란?

한빛미디어에서 출간된 도서들에 대한 리뷰 등 객관적인 도서 평가와 출판사의 기획&마케팅 활동을 돋기 위한 미션을 수행합니다. 궁극적으로는 더 좋은 도서를 출간하기 위해 독자들의 경험과 지식을 공유하는 커뮤니케이션 그룹으로 운영되고 있습니다.

 **신간도서를 읽고 리뷰 및 오픈자 등록**

 **미출간 도서 읽고 미션 수행**

 **급미션 수행**

 **기획의견 및 벤치마킹 아이디어**

 **설문조사 및 베타리더 참여**

 **전략도서 토론회**



한빛리더스 회원은 연 2회 한빛 웹사이트와 페이스북을 통해 모집합니다.

한빛미디어 커뮤니케이션 파트너: 한빛리더스 활동을 지금 확인하세요!

<http://facebook.com/hanbitreaders>