CHAPTER 01

대규모 웹 서비스 개발 오리엔테이션

전체 그림 파악하기

필자 이토 나오야

대규모 웹 서비스로의 초대

대량의 데이터를 처리하는 세계

대규모 웹 서비스의 세계에 온 것을 환영한다. 이 책의 목적은 독자 여러분이 대규모 데이터를 보유한 웹 서비스를 개발할 때의 기본 방침과 전체 그림(지도와 컴퍼스)을 손에 얻기까지 인도하는 데 있다. 이를 위해 월간 고유 사용자수 1,500만 명을 넘는 ㈜하테나(http://www.hatena.ne.jp/)의 웹 서비스 개발을 소재로 대규모 웹 서비스를 운영하는 데 필요한 기초지식과 노하우를 설명한다.

대규모 웹 서비스란, 요컨대 거대한 데이터를 처리해야만 하는 웹 서비스를 말한다. 이 책에서는 대량의 데이터를 처리하기 위한 방법을 축으로 해서 이와 관련된 기술적인 화제를 거론한다. 실제로 가동하고 있는 블로그 서비스인 '하테나 다이어리(http://d.hatena.ne.jp/)'와 일본 내 최대의 소셜 북마크인 '하테나 북마크(http://b.hatena.ne.jp/)' 등의 구체적인 예를 바탕으로 설명해가도록 한다.

우선 제1장에서는 이 책의 테마인 '대규모'의 전체 그림을 떠올릴 수 있도록 하는 것을 목표로 한다. 이어지는 장에서는 대규모 서비스를 위한 개발에 관한 내용을 섞어가면서 대규모 서비스의 규모감, 대규모 데이터를 다루는 데 있어 어려운 점, 개발 모습 등을 설명해가도록 하겠다.

Memo

대규모 데이터를 보유한 웹 서비스 개발의 기본 방침과 전체 그림

- 이 책의 근간(➞ 강의 0)
- 대규모 서비스와 소규모 서비스(➞ 강의 1)
- 계속 성장하는 서비스와 대규모화의 벽(➞ 강의 2)
- 서비스 개발의 현장(➞ 강의 3)

강의 0

이 책의 근간 이 책에서 설명하는 것과 설명하지 않는 것

대규모 서비스 개발에 관련된 대학생 대상의 하테나 인턴십

이 책의 바탕이 되고 있는 것은 하테나에서 진행하고 있는 여름 인턴십(internship)의 강의 내용이다. 하테나에서는 매년 대학생들의 취업체험을 제공할 목적으로 인턴십을 개최하고 있다.

이 인턴십에서는 최종적으로 학생들이 실제 동작하고 있는 하테나의 대규모 서비스를 개발하는 데 참여하게 된다. 사용자가 이용하고 있는 대규모 서비스에 변경을 가할 때 규모를 고려하지 않고 어중간하게 구현해서 적용하다 보면, 어이없게도 아주 간단히 시스템 정지를 초래할 수 있다. 따라서 하테나에서는 경험이 풍부한 선배 사원이 학생들에게 2주 동안 강의를 실시해서 대규모 데이터 처리에 관한 핵심내용을 철저하게 가르친다. 그 강의 내용이 바로 이 책의 근간(根幹)이다. 실제 몇몇 장은 강의 내용 자체를 다듬어 작성되기도 하였다. 대화형식의 강의에서만 가능한 '실제로는 어떻죠?' 와 같이 이른바 "속 시원한 얘기"들을 많이 담아냈다.

강의의 경우에는 매일 강의가 끝날 때 학생들에게 과제를 내서 이해도를 테스트하는 것이 하테나의 방식이다. 이 책에서도 강의를 바탕으로 한 장에서는 실제로 출제했던 과제 또는 응답사례를 설명함으로써 이해를 높일 수 있도록 구성했다.

이 책에서 설명하는 것

테마는 대규모 서비스/대규모 데이터, 바탕은 인턴십 강의로 되어 있어, 그 구성이 약간 특이한 이 책은 주로 다음의 것들을 설명하고 있다.

- 대규모 웹 서비스 개발이란?
- 대규모 데이터를 다룰 때의 과제, 다루기 위한 기본적인 사고방식과 요령.
  예를 들어 OS의 캐시(cache) 기능이나 대규모 데이터를 전제로 한 DB 운용방법
- 알고리즘과 데이터 구조 선택의 중요성. 대규모 데이터를 예로 생각해본다.
- RDBMS(Relational DataBase Management System)로 모두 다룰 수 없는 규모의 데이터 처리방법. 그 예로 전문(全文) 검색 엔진 만드는 법을 살펴본다.
- 대규모 서비스가 될 것을 전제로 한 서버/인프라 시스템의 예와 개념

대규모 데이터 처리를 위해 꼭 설명해야 할 사항은 여러 분야에 걸치므로, 예를 들어 미들웨어 설정방법이나 프로그래밍 언어의 구문(syntax) 등 세세한 부분까지 설명해서는 하루가 금방 저물게 된다. 그래서 소프트웨어 사용법 등 매뉴얼 같은 부분은 설명하지 않고 대규모 서비스, 대규모 데이터를 다룰 경우를 대비한 기본적인 사고방식이나 개념, 개요에 한해서 설명하도록 한다. 대규모 데이터 처리뿐만 아니라 기술습득에서도 중요한 것은 How-To 습득이 아니라 밑바탕이 되는 전체 그림을 파악하는 것이다.

이 책에서는 제2장~제15장까지의 강의에 걸쳐 이런 내용을 정리해가고 있다. 주로 제2장~제5장, 제6장~제10장, 제11장~제15장으로 크게 세 파트로 되어 있다.

제2장~제5장은 데이터가 많을 때 어떻게 처리해야 하는지, 확장성에 문제가 있는 코드가 있어서 시스템이 멈춰버리는 상황이 발생하지 않도록 하기 위해 기본적으로 어떻게 생각하면 좋은지와 같은 내용이 주제를 이룬다. 주로 대규모 데이터를 다루는 애플리케이션 개발에 필요한 지식편으로서의 자리매김을 하고 있다. 구체적으로는 하테나의 서비스 설계로 보는 '대규모 데이터 처리방법'. 제2장은 '도대체 대규모 데이터란 어느 정도를 말하는가'에 대해 언급하고, 제3장에서는 대규모 데이터 처리의 밑바탕인 OS 캐시에 관해 얘기한다. 또한 하테나에서는

MySQL을 많이 사용하고 있는데, 제4장에서는 MySQL을 대규모 환경에서 운용할 때 어떤 점들에 주의를 해야 하는지에 대해 설명한다. 제5장에서는 대규모 데이터를 다루는 애플리케이션을 개발할 때 핵심이 되는 내용을 설명한다.

제6장~제10장은 좀더 구체적으로 프로그래밍, 다시 말해 구현단계와 관련된 파트다. 제6장에서는 먼저 데이터 압축기법에 관한 개요를 배우고 데이터를 컴팩트하게 유지하는 의미와 속도감각을 자세히 설명하고 넘어간다. 제7장 이후는 좀더 애플리케이션에 가까운 알고리즘에 관한 얘기를 한다. 제7장에서는 하테나의 각종 기능 중 알고리즘다운 요소가 필요한 것이 어떻게 적용되어 있는지에 대한 개론. 제8장은 하테나 키워드의 구현을 보면서 보다 구체적인 구현단계로 발을 내딛는다. 제9장과 제10장에서는 전문 검색 엔진을 실제로 개발함으로써 RDBMS에서 다룰 수 없는 규모의 데이터를 어떻게 요리할 것인가에 대해 살펴보도록 한다.

제11장~제15장은 하테나의 인프라 구성 등을 다루면서 오픈소스 중심의 대규모 환경으로 확장성을 갖게 하기 위해서 인프라는 어떻게 구성되어 있는지를 보도록 한다. 인프라 구성은 프로그램을 작성할 때에도 해당 시스템 구조가 애플리케이션 설계에 힌트가 되므로 확장성을 알기 위한 일환으로 배워두면 좋을 것이다.

이 책에서 설명하지 않는 것

아래와 같은 사항은 설명하지 않는다.

- 웹 애플리케이션 개발과 관련된 기본적인 How-To. MVC 프레임워크나 O/R 매퍼(mapper) 사용법 등
- 각종 소프트웨어 사용법. 아파치(Apache), MySQL의 설정방법이나 명령 등
- Perl이나 C++의 구문이나 기법 등 프로그래밍 언어에 대한 설명, 노하우
- 기본적인 알고리즘이나 데이터 구조(예를 들어 정렬, 탐색, 리스트, 해시 등)에 대한 세세한 설명

영광스럽게도 하테나는 소비자를 대상으로 웹 서비스를 운영하는 회사로서 어느 정도의 평가를 받고 있지만, '웹 애플리케이션의 기획방법'과 같이 기술 이외의 테마에 대해서는 이 책에서 언급하고 있지 않다.

위와 같은 사항에 대한 설명은 하고 있지 않으며, 너무 전문가적인 지식을 요구하지도 않는다. 애초에 기업에서의 웹 서비스 개발경험이 전무한 학생들을 대상으로 한 강의를 바탕으로 하고 있다. 취미로라도 웹 애플리케이션 개발경험이 다소 있다면 무리 없이 읽어나갈 수 있는 구성으로 되어 있음을 밝힌다.

앞으로 대규모 서비스를 마주하게 될 여러분에게

'자신이 관련된 웹 서비스의 규모가 미래에 성장해 있다면 어떻게 해야 좋을까?' 웹 서비스 개발자라면 누구나 이러한 불안감을 한 번쯤 가져봤을 것이다. 혹은 이미 거대해진 웹 서비스를 마주하고 고군분투하고 있는 분도 있을 것이다. 이 책이 이러한 분들 모두의 불안을 없애 자신감을 주고, 또한 대규모 서비스를 운용하는 데 도움이 되기를 바란다.

강의 1

대규모 서비스와 소규모 서비스

하테나의 서비스 규모

곧바로 실제적인 대규모 데이터 공략, 대규모 서비스 운용에 대해 살펴보기로 하자. 지금까지 '대규모, 대규모'라고 해왔는데 구체적으로 어느 정도가 '대규모' 인지를 머릿속에 이미지화할 수 있게 실제 하테나의 서비스 규모를 살펴보도록 하자.

이 책의 근원이 되고 있는 2009년 여름의 인턴십 강의 시점에서의 하테나 서비스의 규모는 대략 다음과 같은 수치로 나타났다[주1]. 참고로 그림 1.1이 실제 서버와 데이터가 놓인 데이터 센터의 모습이다.

- 등록 사용자는 100만 명 이상, 1,500만 UU(Unique User, 고유 사용자)/월
- 수십 억 액세스/월(이미지 등으로의 액세스는 제외)
- 피크(peak) 시 회선 트래픽 양은 430Mbps
- 하드웨어(서버)는 500대 이상

100만 명 이상의 사용자들이 블로그를 쓰거나 북마크를 등록하고 있고, 월간 1,500만 명 이상이 방문하고 있다. 이 방문자 수에 의해 월간 수십 억 액세스가 발

주1 여기서 소개한 것은 인턴 강의 시점(2009년 8월)의 수치다. 그 후 일부 서비스가 히트하면서 2010년 4월 원고 집필시점에는 규모가 좀 더 늘어났다(나중에 언급하겠지만, 예를 들어 하드웨어는 600대, 가상화해서 1,300대 정도로 성장했다.

생한다. 이 정도의 액세스가 되면 일일 액세스 로그는 기본적으로 기가바이트(gigabyte) 크기가 되며, DB 서버가 저장하는 데이터 규모도 대략 기가바이트 수준, 많을 때는 테라바이트(terabyte) 정도가 된다.

● **그림 1.1** 데이터 센터의 모습

이런 액세스, 데이터를 처리하는 서버는 당연히 1대로는 불가능하고 500대 이상이 된다. 서버가 500대가 되면 서버를 놓는 랙은 20랙 이상을 사용한다. 20랙이면 서버 이동 시 카트가 필요하고 체력이 꽤나 필요한 작업이다.

500대라는 것은 어디까지나 하드웨어 대수로, 실제로는 이 책 후반부에 설명하는 가상화 기술에 의해 1대의 서버 내에 복수의 호스트가 가동되고 있다. 결과적으로 호스트 수는 1,000대를 넘는다. 1,000대 정도 되면 어떤 서버가 어떤 역할을 하는지, 하테나 다이어리에 몇 대, 하테나 북마크에 몇 대와 같은 호스트 정보를 파악하는 데는 기억력만으로는 곤란해서 호스트 정보를 관리하기 위한 툴 등이 필요해진다.

피크 시 회선 트래픽은 430Mbps. 하테나의 서비스는 텍스트를 주로 하고 있으므로 유튜브(YouTube)나 니코니코 동영상(ニコニコ動画)[역주1]처럼 동영상 전송을

역주1 일본 NIWANGO에 의해 운영되는 동영상 공유 사이트
URL http://www.nicovideo.jp/

메인으로 하는 서비스에 비해 트래픽은 상당히 적은 듯하지만, 반대로 텍스트 전송이 중심임에도 이 정도의 트래픽이 발생한다면 그 유통량은 상당한 것이다.

여기까지가 하테나의 규모다. 대략이나마 이미지를 떠올릴 수 있게 되었는가?

하테나는 대규모, 구글 및 페이스북은 초대규모

주위의 다른 사이트와 비교해서 살펴보도록 하자.

하테나는 각종 인터넷 트래픽 조사에서 대략 일본 내에서 상위 20위 이상에는 항상 랭크인하는 사이트다. 웹 사이트의 통계정보를 공개하고 있는 Alexa의 일본 내 Top 사이트 랭킹[주2]에서 1위는 Yahoo! JAPAN, 그 뒤로 Google, 블로그 서비스를 하고 있는 FC2 등이 뒤를 잇는다. 20위 내에 드는 곳은 유튜브, 라쿠텐(楽天天), 믹시(mixi), 트위터(Twitter), 니코니코 동영상, 2채널(2Channel) 등의 메이저 사이트들이다.

하테나와 비슷한 정도이거나 하테나보다 몇 배 정도의 트래픽이 발생하는 사이트의 규모감은 아마 그다지 하테나와 다르지 않을 것이다. 서비스의 규모는 서버 대수 등으로 개략적으로 파악되는 경우가 많은데, 이런 관점에서 볼 때 백 대에서 수천 대 정도가 대규모 서비스라고 할 수 있을 것이다.

Google 및 해외에서 인기 있는 SNS로 최근 구글의 트래픽을 앞질렀다고 뉴스가 된 페이스북과 같은 세계의 Top 클래스 사이트는 서버 대수가 수백만 대 규모이고, 처리하는 데이터는 테라바이트~페타바이트(petabyte)급의 초대규모 서비스다. 페이스북의 데이터 센터 내에서는 직원이 작업을 위해 자전거나 스쿠터를 타고 데이터 센터 내를 이동할 정도다[주3]. 이동수단은 농담과도 같은 진실이지만, 기

주2 URL http://www.alexa.com/topsites/countries/JP

주3 'Facebook on bandwidth' URL http://link.brightcove.com/services/player/bcpid1701276884?bclid=1622640422&bctid=40363249001, '3억 명의 사용자를 지닌 페이스북의 데이터 센터. 이동은 자전거, 희망은 100Gb 이더넷' URL http://www.publickey1.jp/blog/09/3facebook100gb.html

술면으로도 운용면에서도 초대규모가 되면 또 다른 어려움이 있으리라는 것은 쉽게 상상할 수 있다. 구글의 규모감에 대해 보다 상세히 알고자 하는 분은 『구글을 지탱하는 기술』(니시다 케이스케 저/김성훈 역, 멘토르, 2008) 등도 참고하기 바란다.

한편, 일본 내에서는 모바일 사이트에도 대규모/초대규모 사이트가 군데군데 보이기 시작한다는 점도 알아두면 좋을 것이다. 대표적으로는 최근 수년 만에 급격하게 성장한 소셜 게임 사이트인 GREE나 모바게 타운 등이다. 이 사이트들도 서버 대수가 수천 대 이상 되는 거대한 서비스다.

소규모 서비스와 대규모 서비스의 차이

서버 몇 대 정도의 소규모 서비스에는 없는, 대규모 서비스에만 있는 문제나 어려움에는 어떤 점들이 있을까?

확장성 확보, 부하분산 필요

가장 먼저 떠오르는 것은 확장성과 부하분산일 것이다.

대량의 액세스가 있는 서비스에서는 서버 1대로 처리할 수 없는 부하를 어떻게 처리할 것인지가 가장 큰 문제다. 최근 10년 동안의 트렌드로는 이른바 '스케일아웃(scale-out)'이 이 문제에 대한 전략의 기초가 된다. 스케일아웃은 서버를 횡으로 전개, 즉 서버의 역할을 분담하거나 대수를 늘림으로써 시스템의 전체적인 처리능력을 높여서 부하를 분산하는 방법이다. 반면 스케일업(scale-up)은 하드웨어의 성능을 높여 처리능력을 끌어올리는 방법이다.

알다시피 하드웨어의 성능과 가격은 비례하지 않는다. 대량생산되고 있는 '일용품' 성격의 하드웨어일수록 저가에 구할 수 있다. 저가의 하드웨어를 횡으로 나열해서 확장성을 확보하는 것이 스케일아웃 전략이다. 스케일아웃 전략을 채용한 경우는 비용이 절감되는 반면에 다양한 문제가 발생한다. 서버가 1대였을 때에는 전혀 생각지 않아도 될 문제가 나타난다. 몇 가지 예를 살펴보자.

예를 들면 사용자로부터의 요청을 어떻게 분배할 것인가? 해답으로는 로드밸런서를 사용한다는 것인데, 서버 1대일 때에는 애초에 로드밸런서를 도입하는 것 자체를 생각할 필요도 없을 것이다.

데이터 동기화는 어떻게 할 것인가? DB를 분산시켰을 때 한쪽에 저장된 갱신 내용을 다른 한쪽 DB가 알지 못한다면 애플리케이션에 비정상 사태가 발생한다.

네트워크 통신의 지연시간(latency)을 어떻게 생각해볼 수 있을까? 작은 데이터라도 이더넷(Ethernet)을 경유해서 통신한 경우는 밀리초(ms) 단위의 지연시간이 있다. 밀리초라고 하면 사람이 체감하기로는 그다지 긴 시간이 아니더라도 마이크로초(μs)나 나노초(ns)에 작동하는 컴퓨터에 있어서는 매우 긴 시간이다. 통신의 오버헤드를 최소한으로 줄여가면서 애플리케이션을 구성해갈 필요가 있다. 그 밖에 스케일아웃에 동반하는 문제는 다방면에 걸쳐 있다.

다중성 확보

시스템은 다중성을 지닌 구성, 즉 특정 서버가 고장 나거나 성능이 저하되더라도 서비스를 계속할 수 있는 구성으로 할 필요가 있다.

스케일아웃을 해서 서버 대수가 늘어나면 서버의 고장률도 필연적으로 올라가게 된다. 그러므로 어딘가 잘못되면 서비스가 전부 정지해버리는 설계는 24시간 365일 계속 가동되어야 하는 웹 서비스에서는 도저히 용납할 수 없다. 서버가 고장 나더라도 혹은 급격하게 부하가 올라갈 경우에도 견딜 수 있는 시스템을 구성할 필요가 있다. 서비스가 대규모화되면 될수록 시스템 정지의 사회적 충격도 늘어나므로 더욱 더 다중성 확보가 중요해진다.

2001년 9월 미국의 동시다발적인 테러 발생 시에 상황을 알고자 하는 사람들이 일제히 야후(Yahoo!)에 액세스해서 야후 Top 페이지가 다운돼버리는 사태가 일어났다고 한다. 유사시에 즉시성 측면에서 가장 의지할 곳으로 여겨지는 인터넷 서비스를 사용할 수 없게 되었다. 사회적 충격의 크기를 말해주는 예인 것이다. 야후는 이때 CDN 서비스인 Akamai에 컨텐츠를 캐싱해서 트래픽을 우회시킴으로써

장애를 복구했다고 한다[주4].

웹 서비스는 언제 어떠한 경우라도 고장에 대해 견고해야 한다. 그렇다고는 해도 이는 상당히 어려운 태스크다. 시스템이 고장 나면 그걸로 끝이어도 괜찮은 시스템 구축과 고장 나더라도 다른 시스템이 자동적으로 처리를 인계받는 시스템 구축 간에는 기술적으로나 비용면에서 상당히 큰 차이가 있다.

효율적 운용 필요

서버가 1대라면 때때로 상태를 확인하는 정도로 서버가 정상적으로 동작하고 있는지를 간단하게 파악할 수 있을 것이다. 반면 서버 대수가 100대를 넘어서면 어떤 서버가 무슨 역할을 하고 있는지 기억해두는 것조차 곤란해진다. 또한 각 서버가 어떤 상황에 있는지 파악하는 것도 상당한 고생거리다. 부하는 괜찮은지, 고장 난 부분은 없는지, 디스크 용량은 아직 충분한지, 보안설정에 미비한 점은 없는지 등등… 이를 모든 서버에 대해 여기저기 잘 살펴야 하므로 큰일이다.

당연히 이쯤에서 감시용 소프트웨어를 사용하고 정보관리를 위한 툴을 사용하는 등 자동화를 하게 된다. 그러나 이 감시 소프트웨어를 설치하거나 정보를 보는 것은 결국 인간이다. 일손을 거치지 않고 대규모 시스템을 건강한 상태로 얼마나 계속 유지해갈 수 있을 것인가? 이를 위한 효율적 운용을 수행해야만 한다.

개발자 수, 개발방법의 변화

대규모 서비스가 되면 당연히 혼자서는 개발이나 운용이 어려워지므로 여러 기술자가 역할을 분담하게 된다. 사람 수가 늘어나면 역시나 고려해야 할 과제가 늘어난다. 예를 들면 개발 표준화는 어떻게 할 것인가? 애플리케이션을 각각의 기술자가 제멋대로 구현한 시스템의 전말은 생각도 하기 싫다. 프로그래밍 언어를 통일하고, 라이브러리나 프레임워크를 통일하고, 코딩 규약을 정해서 표준화하고,

주4 URL http://d.hatena.ne.jp/yamaz/20060911

소스코드 관리를 버전관리 시스템으로 제대로 하기. 이런 사항들이 올바르게 실행되기 시작해야 여러 사람이 작업할 때 좋은 효율이 나타난다. 여러 사람이 제각기 작업해서는 사람이 늘어나도 생산성은 오르지 않는다.

이러한 표준화는 툴을 정하는 것만으로는 좀처럼 잘 이루어지지 않으며, 누군가가 전체를 조정할 필요도 생긴다. 개발자 개개인과 팀에서 표준화 규칙이 지켜지고 있는지, 기술자 간 능력 차이에 따라 효율이 나쁜 부분은 생기지 않는지, 그렇다고 할 때 어떻게 교육을 할 것인지 등등… 팀 매니지먼트가 필요해지는 것이다.

대규모 데이터량에 대한 대처

이 책에서 가장 큰 테마가 바로 데이터량이다.

컴퓨터는 디스크(하드디스크, HDD)에서 데이터를 로드해서 메모리에 저장, 메모리에 저장된 데이터를 CPU가 패치(fetch)해서 특정 처리를 수행한다. 또한 메모리에서 패치된 명령은 보다 빠른 캐시(cache) 메모리에 캐싱된다. 이처럼 데이터는 디스크 → 메모리 → 캐시 메모리 → CPU와 같이 몇 단계를 경유해서 처리되어 간다.

각 단계 간에는 속도차가 매우 크게 나는 것이 현대 컴퓨터의 특징이다. 하드디스크에서 데이터를 읽어들이는 데에는 그 특성상 헤드 이동이나 디스크 원반의 회전이라는 물리적 동작이 수반된다. 따라서 전기적으로 읽어들이기만 하면 되는 메모리나 캐시 메모리와 비교하면 10^6 ~ 10^9배나 되는 속도차가 나게 된다.

이 속도차를 흡수하기 위해 OS는 이런저런 방법을 사용하게 되는데, 예를 들면 디스크로부터 읽어들인 데이터를 메모리에 캐싱해둠으로써 전반적으로 디바이스 간 속도차가 체감속도에 영향을 주지 않도록 하고 있다. DB를 비롯한 미들웨어도 기본적으로 이러한 속도차를 의식한 데이터 구조, 구현을 채용하고 있다.

하지만 OS나 미들웨어 등의 소프트웨어에서 이런 구조를 통해 분발한다고는

해도 당연히 한계는 있다. 데이터량이 많아지면 처음부터 캐시 미스(cache miss)가 많이 발생하게 되고, 그 결과로 저속의 디스크로의 I/O가 많이 발생하게 된다. 디스크 I/O 대기에 들어선 프로그램은 다른 리소스가 비어 있더라도 읽기가 완료되기까지는 다음 처리를 수행할 수가 없다. 이것이 시스템 전체의 속도저하를 초래한다.

대규모 웹 애플리케이션을 운용할 때 대부분의 어려움은 이러한 대규모 데이터 처리에 집중된다.

데이터가 적을 때에는 특별히 고민하지 않아도 모두 메모리에서 처리할 수 있으며, 복잡한 알고리즘을 사용하기보다 간단한 알고리즘을 사용하는 편이 오버헤드가 적기 때문에 더 빠른 경우도 종종 있으므로 I/O 부하는 일단 문제가 되지 않는다. 그러나 서비스가 어느 정도 이상의 규모가 되면 데이터는 증가한다. 이 데이터량이 분수령을 넘어서면 문제가 복잡해진다. 그리고 응급처리로는 쉽사리 풀리지 않는다. 이 점이 대규모 서비스의 어려운 점이다.

어떻게 하면 데이터를 적게 가져갈 수 있을까, 여러 서버로 분산시킬 수 있을까, 필요한 데이터를 최소한의 횟수로 읽어들일 수 있을까 등등… 이것이 본질적인 과제가 된다.

강의 2

계속 성장하는 서비스와 대규모화의 벽

웹 서비스의 어려움

대규모가 되면 어떤 문제가 발생하는지에 대한 개략적인 내용은 강의 1에서 살펴봤다. 웹 서비스가 다른 애플리케이션에 비해 어려운 또 하나의 원인은 서비스가 계속해서 성장해간다는 데 있다. 처음에는 소규모였던 서비스가 성장함에 따라 그 규모가 확대해가는 것이다. 성장해감에 따라 시스템 구성을 변화시켜 갈 필요가 있는 것이다.

서비스를 1년, 2년 계속 운용해가다 보면 거기에 보유하게 되는 데이터량도 성장해간다. 예를 들면 블로그 서비스 등을 떠올려 보기 바란다. 그런대로 사용되고 있는 서비스라면 매일 사용자가 새로운 일상사, 사진 등을 올릴 것이다. 데이터가 늘어났다고 해서 예전 데이터를 블로그 사업자가 임의로 지워버릴 수는 없으므로 모든 데이터는 계속해서 잘 보존하고 필요한 경우 추출해낼 필요가 있다.

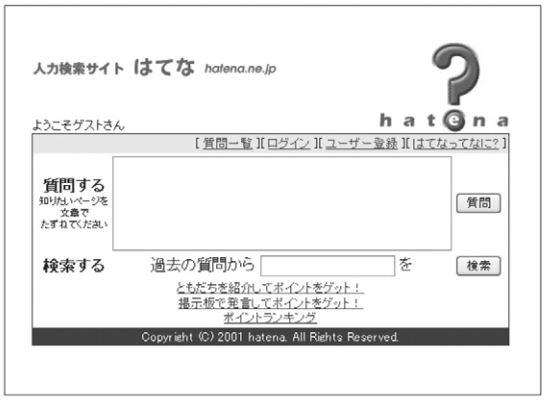
블로그 서비스와 같이 불특정다수의 사용자를 대상으로 개방하는 서비스가 상업적으로도 성공한 경우는 데이터뿐만 아니라 트래픽도 늘어간다. 그에 따라 데이터 조회 수나 작성횟수도 늘어갈 것이다. 이렇게 해서 서비스가 성장함에 따라 시스템 확장이 필요해지는 것이다.

다음에는 하테나의 성장을 뒤돌아보면서 어떤 어려움과 노고가 있었는지 살펴보도록 하자.

하테나가 성장하기까지

하테나가 2001년 당시, Q&A 사이트인 인력검색 하테나[주5]를 시작했을 때는 종업원 3명인 작은 벤처기업이었다(그림 1.2 참조). 따라서 시스템에 투자할 자본도 거의 없었기 때문에 초기 시스템은 펜티엄 III PC 1대뿐이었다. 회선은 당시 가까스로 최종사용자가 사용할 수 있게 된 변변치 않은 ADSL 회선이었다. 지금이야 대규모 서비스를 운영할 수 있게 되었지만, 당시에는 우리도 경험이 없었기 때문에 아무 생각도 없는 상당히 불완전하게 시작된 시스템 구성이었다. 그렇지만 초반에는 서비스가 좀처럼 히트되지 않았으므로 1대의 PC가 비명을 지르게 되는 일은 거의 없었다.

● **그림 1.2** 인력검색 하테나(2001년 당시)※



※ 2008년 7월, 하테나 7주년을 기념해서 재차 선보인 2001년 당시의 「인력검색 하테나」 Top 페이지
URL http://d.hatena.ne.jp/reikon/20080715/p1

주5 URL http://q.hatena.ne.jp/

얼마 후 하테나 안테나[주6]와 하테나 다이어리 등 새로운 서비스를 릴리즈한 무렵부터 상황이 변하기 시작했다. 서비스가 사람들 사이에 입으로 전해지면서 서서히 사용자가 늘어갔다. 역시나 사용자가 늘어나다 보니 서버 1대로는 다 처리할 수 없어 서비스가 정지해버리는 일이 빈번하게 발생하게 되었다. 이래서는 안 되겠다는 생각으로 서서히 다중화와 부하분산을 적용해나가게 되었다. 그러나 당시에 상용 로드밸런서 등은 매우 고가였으므로 엄두조차 낼 수 없는 장비였다.

시행착오를 거듭한 시스템 규모확장

그래서 오픈소스 소프트웨어를 활용하게 되었다. 하지만 웹 서비스의 부하분산에 관한 모범사례 등의 정보도 떠돌지 않던 시기였으므로 시행착오의 연속이었다. 라우터는 Linux 박스로 저가에 구축, HTTP 요청 분산은 아파치의 mod_rewrite로 대용, DB 분산은 아직까지 불안정했던 MySQL의 레플리케이션 기능을 조심조심 사용하던 상황이었다.

이렇게 해서 서서히 시스템 규모를 확장해가고 있었는데, 2004~2005년쯤에는 블로그 붐이 일어나서 트래픽 증가에 비해 시스템 확장이 따라가지 못하게 되었다. 애초에 대규모 시스템을 운용하기 위한 체제가 갖춰져 있지 않았던 점도 있지만, 불거져 나오는 문제가 너무 많아서 대처속도가 따라가질 못했는데, 심야가 되면 꼭 사이트가 무거워지거나 액세스가 불가능해져서 하테나의 안정성은 빈말로라도 칭찬할 수 없는 상태였다.

당시는 분산뿐 아니라 다중화, 효율적 운용면에서도 낙제점이었다. 서비스가 정지하면 감시 소프트웨어에서 휴대전화로 메시지가 전달되었는데, 다중화된 시스템이 자동으로 가동되는 것이 아니라 근처에 살고 있는 엔지니어가 알아차리고 자전거로 달려가서 Hang-Up된 서버를 재기동하는 등의 응급처치로 견뎌냈다. 심야 유지보수는 체력적으로나 정신적으로 큰 소모였다. 경우에 따라서는 심야에

주6 URL http://a.hatena.ne.jp/

시스템이 여러 번 다운되는 경우도 있어서 겨우 조치하고 나서 거의 귀가했다 싶으면 다시 달려가야 하는 상황도 종종 있었다.

그래도 현장의 노력으로 어떻게든 버텨내고 있었으나 2006년에는 다른 서비스가 연이어 히트하면서 이제는 회선 트래픽 양이 이용 중이던 광회선의 한계에 이르게 되었다. 그뿐만 아니라 당시 임대해서 쓰던 작은 서버 룸이 서버 증설에 따라 전력이 부족하게 되어 더 이상 서버를 추가할 수 없는, 추가하게 되면 전류 차단기가 내려가버리게 되는 상황에 처하게 되었다.

데이터 센터로의 이전, 시스템 쇄신

지금 되돌아보면 당시 하테나는 조직상으로도 아직 미숙해서 계속 증가해가는 트래픽이나 부하에 대해 계획적으로 대응할 수 없었던 점이 실패 원인이었던 것 같다. 어떻게도 할 수 없는 상황이 되어서야 조직체제를 재점검하고 시스템 운용 전담팀을 구성해서 그 이후의 대응에 임했다.

이때부터 1년에 걸쳐 작은 서버 룸에서 인터넷 데이터 센터로 서버를 이전하는 작업을 시작했다[주7]. 이전 작업을 하면서 네트워크 설계를 근본적으로 재수정하고 낡은 서버는 모두 교체하는 방침을 채택했다.

우선은 사전에 기존 시스템의 부하상황을 정리했다. 이 정보를 활용해서 각 서비스의 구성 중에 병목지점을 측정, 판정하고 I/O 부하가 높은 서버는 메모리를 중요시하고 CPU 부하가 높은 서버는 CPU를 중요시하는 형태로 서버 용도에 맞게 최적의 구성을 갖는 하드웨어를 준비해갔다.

다중화의 경우 로드밸런서 + 가동감시 기능을 하는 오픈소스 소프트웨어인 LVS + keepalived를 도입했다. 이에 따라 Linux 박스로 저가에 구축한 로드밸런서를 각 부문에 도입함으로써 차츰 개선해갔다.

서버 교체에 있어서는 서서히 OS 가상화도 진행해서 서버 가동률을 높임과 동

주7 이전 작업하는 모습은 필자의 블로그(http://d.hatena.ne.jp/naoya/)에 여러 번에 나눠서 공개하고 있다. 관심 있는 분은 위 블로그에서 "사쿠라 인터넷 이행기(さくらインターネット移行記)"로 검색해보기 바란다.

시에 유지보수성을 높여 갔다.

서버의 정보관리를 위해 독자적인 웹 기반 서버 정보관리 시스템도 개발했다. 이에 따라 서버의 용도나 부하상태와 같은 각종 정보에 액세스하기 쉬워져서 시스템 전체를 파악하기 용이해졌다.

서버/인프라 측면의 시스템 구성뿐만 아니라 애플리케이션의 각종 로직이나 DB 스키마 등도 재검토해서 비효율적인 부문을 서서히 배제해갔다. 필요에 따라 검색 엔진 등의 서브시스템을 독자 개발하는 경우도 있었다.

이런 종류의 작업을 계속 해나간 결과 서서히 시스템 안정성이 향상되기 시작하고, 그 결과로 장애에도 견딜 수 있는 시스템으로도 개선할 수 있었다.

그러나 시스템이 안정되었다고 해서 여기서 끝난 것은 아니다. 여기서 멈춰버린다면 결국 그 이후의 성장을 위해서는 다시 예전과 동일한 작업을 반복해야만 한다. 그러지 않기 위해 현재도 변함 없이 개발 담당, 인프라 담당이 하나가 되어 밤낮없이 시스템 품질개선을 수행하고 있다.

시스템의 성장전략 — 미니멈 스타트, 변화를 내다본 관리와 설계

서비스가 아직 소규모인 단계에서는 심플한 방법이 더 나은 경우가 많으므로 너무 이른 최적화가 좋은 방침이라고는 할 수 없다. 장래에 대규모가 될 것을 가정해서 처음부터 완벽한 부하분산 시스템을 구축하려고 하면 비용이 너무 많이 들게 된다(웹 서비스가 상업적으로 히트할 가능성이 매우 낮다는 사실을 잊어서는 안 된다).

한편 아무 생각 없이 불완전하게 시작하는 서비스도 생각해볼 수 있다. 이는 얄궂게도 하테나의 역사가 증명하고 있다. 대규모화의 벽은 갑작스레 눈앞에 나타난다. 예를 들면 데이터 규모가 증가함에 따른 I/O 부하 상승은 그 정도로 순조롭게 증가하는 것은 아니다. 캐시 미스가 발생하기 시작한 후 오래지 않아 갑자기 문제가 복잡해지므로, 알아차렸을 때에는 이미 시스템이 저속화하고 있는 경우가 자주 있다.

이러한 사태가 발생하지 않게 하기 위해 어느 정도의 수용능력 관리나 서비스 설계 시에 필요 이상으로 데이터를 증가시키지 않도록 하는 설계를 포함시키는 게 좋을 것이다.

지금은 하테나도 새로운 서비스를 시작할 때 요소요소에서 장래의 서비스 성장을 예측한 구성으로 해두면서, 필요 이상으로는 비용을 들이지 않고 미니멈 스타트(minimum start)할 수 있는 노하우를 보유하고 있다. 이 책을 통해 여러분이 그 노하우를 흡수할 수 있기를 바란다.

강의 3

서비스 개발의 현장

하테나의 기술팀 체제

그러면 제1장의 결말로서 현재 하테나가 어떠한 체제로 웹 서비스를 운영하고 있는지를 소개함으로써 대규모 서비스 운용의 뒷모습을 보다 구체적으로 이미지화할 수 있도록 하자. 주로 엔지니어링에 관계된 팀은 크게 두 부서로 나뉘어 있다.

- 서비스 개발부 : 하테나의 각종 서비스 구현을 담당하는 팀. 매일 애플리케이션 측면의 개선을 담당한다.
- 인프라부 : 서버/인프라 시스템의 운용을 담당하는 팀. 서버 준비, 데이터 센터 운용, 부하 분산 등을 담당한다.

하테나의 서버/인프라 시스템은 1,000대 규모의 호스트를 보유하는 큰 시스템이므로 전담팀이 그 운용을 담당하고 있다. 호스트가 1,000대나 있으면 며칠에 한 번은 문제가 발생한다. 고장, 과부하, 설정미비, 노후화로 인한 교체 등등. 이런 문제들을 전담해서 맡으면서 가상화나 클라우드 등 보다 세련된 새로운 모습으로 시스템을 확장해가는 것도 인프라부의 일이다. 현재 사원 4명과 아르바이트 몇 명이 소속되어 있다.

한편 서비스 개발부는 서버/인프라 시스템이 지탱하는 기반 위에 서비스를 개발한다. 하테나 다이어리나 하테나 북마크에 신기능을 추가하는 것이 이 부서이다. 서비스 개발부 내에는 서비스별로 팀이 나뉘어 한 팀당 3~4명의 엔지니어가 소속되어 있다.

부하 상황은 서버/인프라팀이 감시하고 있고, 인프라 부분의 개선으로 대처할 수 있는 문제는 그들만으로 즉시 대응한다. 애플리케이션에 원인이 있는 경우 등 구현이 관련된 경우에는 서비스 개발부와 협력해서 대응한다.

한편 서비스 개발부에서도 담당하고 있는 서비스의 성능을 트래킹(tracking)하고 있으며, 주요한 페이지가 어느 정도의 응답시간에 응답하고 있는지를 정량화해서 매일 그것을 지표로 한계값(threshold)을 밑돌지 않도록 목표를 설정해서 개선하고 있다.

하테나에서의 커뮤니케이션 방법

하테나는 아직 종업원 40명 정도의 작은 기업이므로 기본적으로 업무지시는 구두로 한다. 단, 구두로는 효율이 나쁜 경우도 있고 이력도 남지 않으므로 그 점은 몇몇 툴을 조합해서 지시를 주고받는다.

• **하테나 그룹** URL http://g.hatena.ne.jp/

블로그와 위키(Wiki)를 조합한 그룹웨어. 매일의 업무 리포트 보고에 사용. 엔지니어는 다른 담당자에게 작업을 의뢰할 경우 블로그에 의뢰내용을 적어 담당자에게 트랙백을 날린다. 메일은 사용하지 않음. 또한 유지보수 등으로 시스템 작업이 발생한 경우는 절차를 작업로그로서 블로그에 남기거나 위키에 정리함으로써 나중에 참조할 수 있도록 하고 있다.

• IRC

도쿄/교토로 사무실이 분산되어 있기도 하므로 IRC 채팅을 사용한 커뮤니케이션도 이용. 유지보수 진행상황이나 긴급 문제가 있을 경우, 간략한 용건 전달 등은 IRC를 이용할 경우가 많다. 채널 중에는 시스템 로그가 표시되는 채널도 있다. 서버에 장애가 발생했을 때 장애 통지를 IRC로 출력시킴으로써 실시간 정보를 공유

할 목적으로 이용하기도 한다.

• **서버 관리툴**

강의 32에서 소개할 서버 관리툴도 어떤 면에서 엔지니어 간 커뮤니케이션 툴로 역할을 하고 있다. 현재 서버 상황을 한눈에 알 수 있는 툴이므로 개발 담당은 이 툴을 사용해서 유지보수 예정 유무를 확인하거나 시스템을 갱신해도 되는지 등을 판단한다. 또한 부하 상황 등도 이 툴로 참조할 수 있으므로 URL을 주고받으면서 '여기 부하는 OK, 여기는 안 되겠어'와 같은 정보를 IRC나 하테나 그룹으로 공유할 때도 많다.

실제 서비스 개발

시스템에 구현 측면의 변경을 가하기까지의 일련의 흐름은 다음과 같다.

- 매일 아침 각 팀별로 10분 정도 짧은 미팅을 한다. 미팅 중에 어제 진척상황, 오늘 할 일 등을 공유한다.
- 이 미팅에서 태스크 담당자가 정해진다. 각 담당은 미팅 후 바로 해당 태스크 구현에 들어간다.
- 구현에 있어서는 가능한 한 테스트 프로그램을 작성한다. '가능한 한'이라는 건 테스트 프로그램을 작성하는 것이 기본이지만, 레거시(legacy) 서비스 등 테스트 프로그램을 작성하기 다소 어려운 서비스도 있으므로 어느 정도 타협을 한다는 의미이다. 원리주의적으로 '테스트 프로그램을 모두 작성해야 한다'라는 방침을 취하지 않고 테스트 프로그램에 관해서는 유연하게 대응한다.
- 테스트 프로그램을 작성한 후 구현. 구현이 완료되면 버전관리 시스템에 커밋(commit)한다.
- 구현이 끝나면 팀 내 다른 엔지니어에게 부탁해서 코드 리뷰를 한다. 이 리뷰에 의해 버그가 지적되거나 사내 코딩규약에 따르지 않는 코드가 발견되거나 과부하가 유발될 만한 코드 등이 밝혀진 후 개선된다. 코드 리뷰는 매우 유용한 수단이다.
- 리뷰를 지나면 프로덕션 환경(실제로 동작하고 있는 시스템 환경)의 코드에 머지(merge)를 하고 스테이징 환경(동작 확인용 환경)에서 동작을 확인한 후 프로덕션 시스템에 반영한다.

여기까지가 기본적인 흐름이다. 변경 규모에 따라서는 리뷰를 건너뛰는 경우도 있다. 이것도 테스트 프로그램과 마찬가지로 효율과 품질을 저울질해서 유연하게 대응하도록 하고 있다.

구현자의 실력이나 구현 난이도에 따라서는 짝 프로그래밍(Pair Programming)을 할 경우도 있다. 혼자서 작업을 하기가 불안할 경우에 짝 프로그래밍을 할 때가 많은데, 짝 프로그래밍으로 보다 높은 품질을 보증할 수 있다. 단, 짝 프로그래밍은 매우 피로도가 높은 작업이므로 매번 하지는 못하고 사안에 따라 실시하도록 하고 있다.

전체적으로 애자일(agile) 개발 스타일이라고 할 수 있지만, 특정 교과서를 따라 이런 스타일이 됐다기보다는 창업 때부터 다양한 시행착오를 거쳐 지금 모습으로 자리잡았다는 표현이 맞을 것이다. 과거에는 리뷰나 짝 프로그래밍도 없었던 시기도 있었지만, 시스템 정지를 초래하는 등 치명적인 코드를 커밋해버리는 일이 가끔 발생해서 그 방지책으로 현재 방법으로 개발을 진행하게 되었다.

개발에 사용하는 툴

엔지니어가 개발에 사용하고 있는 툴도 지금 소개해두겠다.

프로그래밍 언어 — Perl, C/C++, JavaScript 등

서버-사이드(Server-Side)인 웹 애플리케이션은 창업 때부터 Perl로 개발하고 있기 때문에 Perl을 이용한다. 검색 엔진 등 메모리 요건이 엄격하거나 속도가 요구되는 곳에는 일부 C/C++ 등도 사용한다. 웹 애플리케이션의 사용자 인터페이스 개발은 별다른 선택의 여지없이 JavaScript를 사용한다.

프로그래밍 언어 선택 정책은 '동일 레이어인 언어는 하나로 한정한다'는 것이다. 예를 들면 근래의 웹 애플리케이션 개발에는 PHP, Python, Ruby 등의 스크

립트 언어가 자주 사용된다. 개발자에 따라서는 이러한 언어가 기호에 맞는 사람도 있겠지만, 기본은 Perl을 사용하도록 부탁하고 있다. 이는 앞서 언급한 표준화의 관점에서 중요하다.

같은 언어를 사용하면 자사 내에서 노하우가 널리 통용되고 팀 간 이동도 원활하다. 다른 사람이 만든 시스템의 유지보수도 용이하다.

Perl, PHP, Python, Ruby에는 구문상 특징으로 언어 간 차이는 있지만, 기본적으로 담당영역은 유사하다. 어느 것을 사용하더라도 생산성이나 할 수 있는 일이 비슷한 정도일 것이다. 이런 경우는 어느 하나를 선택해야 하는데, 표준화 관점에서 관습을 중시한다. 한편, C/C++은 분명하게 스크립트 언어가 전문으로 하지 않는 영역에서도 힘을 발휘할 수 있는 언어이므로 Perl로 할 수 없는 일을 할 때 이용한다.

주요 미들웨어 — 미들웨어/프레임워크 통일

마찬가지로 표준화 관점에서 이용할 미들웨어와 프레임워크도 모든 팀에서 통일하고 있다. 어느 팀은 MySQL을 사용하고 다른 팀은 PostgreSQL을 사용하는 일이 일어나지 않도록 관리를 확실하게 하고 있다.

주요 미들웨어는 Linux, Apache, MySQL, memcached와 같은 최근 웹 개발의 기본에 해당하는 것들이다. '너무 최신인 것은 다루지마'와 같은 정책은 없지만, 개발자의 경험상 안정화된 것이 가장 사용하기 쉽다는 점 때문인지 기본적으로 쓰이는 것이 선택되는 경향이 있다.

웹 애플리케이션 프레임워크 — 자체 개발한 Ridge

지금은 애플리케이션 개발의 효율을 높이기 위해, 또한 표준화를 진행하려는 의미로 프레임워크를 이용하는 것이 당연시되었다. 하테나에서는 웹 애플리케이션 프레임워크로 자체 개발한 Ridge라는 Perl 프레임워크를 사용하고 있다.

Ridge는 이렇다 할 특징적인 기능이 있는 것은 아니지만, 그만큼 심플하고 직

관적으로 사용할 수 있는 MVC 프레임워크다(공개하고 있지는 않다). 일단 웹 서비스를 만들면 그 이후에 수년 동안이나 계속 개발할 경우도 많다. 따라서 그다지 유행에 좌우되지 않는 기본적인 프레임워크를 사용하는 것이 좋다고 생각한다.

O/R 매퍼에도 DBIx::MoCo라는 자사에서 자체 개발한 라이브러리를 사용하고 있다. 이것은 Perl의 라이브러리 집적소인 CPAN에 공개되어 있다[주8].

주위 머신의 OS 및 에디터 — 기본적으로는 자유

프레임워크나 미들웨어는 프로덕션에서 사용하는 것과 동일한 것을 사용하지 않으면 개발이 어려우므로 표준화하고 있지만, 그 이외의 부분, 예를 들면 에디터나 PC의 OS는 기본적으로 자유다. Emacs나 Vim, Windows나 Mac OS X 등 각자 원하는 것을 사용하고 있다.

그렇지만 Windows에서 Apache를 구동시키는 것은 아니고 VMware나 coLinux와 같은 가상 OS를 각 개발자가 도입해서 그 위에 프로덕션에서 사용하는 Linux를 구동해서 개발을 진행한다.

에디터는 자유이지만 코딩 규약으로 indent 폭이나 block 스타일 등은 큰 틀에서 정해져 있으므로 거의 모든 개발자는 Emacs나 Vim을 사용해서 하테나의 규약대로 정형화해주는 설정을 도입하고 있다.

버전관리는 git, BTS는 독자 개발한 '아시카'

소스코드에 대한 버전관리에는 git를 사용한다. git는 분산 버전관리 시스템으로 브랜치(branch)를 나누거나 머지(merge)하는 등의 작업을 간단히 수행할 수 있다는 점에서 애용하고 있다. 이전에는 CVS나 Subversion 등을 사용했으나 지금은 완전히 git로 이전 완료했다.

BTS(Bug Tracking System, 버그추적 시스템)은 자사의 git 리포지토리와 연동한

주8 URL http://search.cpan.org/dist/DBIx-MoCo/

'아시카' 라는 독자 개발한 웹 애플리케이션을 이용하고 있다. 태스크와 소스코드 변경을 대비해서 볼 수 있는 기능 등 호스팅 서비스 GitHub와 같은 시스템을 자사용으로 마련한 것이다.

개발툴에 관해서

지금까지 특별한 소프트웨어를 이용하고 있는 것이 아니라는 것을 알았으리라 생각한다. 자사에서 개발한 프레임워크 등도 색다르게 만들어졌다기보다 오히려 기능은 적은 편이다.

대규모 개발에 있어서는 고기능인 툴을 선택하기보다도 얼마나 효율성을 희생하지 않고 표준화할 수 있는가, 자사의 Workflow에 맞게 사용할 수 있는가와 같이 툴이 아닌 부분의 운용방법을 어떻게 할 것인지가 더 본질적인 과제라고 느끼고 있다.

정리

제1장에서는 대규모 웹 서비스의 개발 모습으로서 여기서 발생하는 문제를 살펴봤다. 또한 이 책에서 다루는 '대규모' 를 이미지화하기 위해 하테나의 실제 모습을 소개했다.

소규모 웹 서비스가 어느 정도 성장하면 갑자기 지금껏 없었던 난제가 나타난다. 그러면서 서비스는 매일 성장해간다. 이 성장에 수반되는 노고는 하테나의 발전 모습으로부터 슬쩍 엿볼 수 있었을 것이다. 또한 현재 규모의 하테나를 지탱하는 현장의 체제, 개발툴 등을 살펴봄으로써 실제 현장의 모습도 어느 정도 그려볼 수 있었으리라 생각한다.

그러면 개론은 이 정도로 하고 이어지는 강의에서 보다 구체적으로 대규모 데이터 공략방법과 대규모 시스템의 구성방법을 설명해가도록 하겠다.