☑ 増築しよう

6章 スケールアウト

誰もが、自分のWebサイトがより大きいものへと成長してほしいと願っているはずです。今の家が手狭になってきたら、引っ越すか増築する必要があります。スケーラビリティの実現のためにRailsが用意している機能を活用すれば、一軒家にとどまらずマンションや地域社会全体を運営してゆくことさえも不可能ではありません。この章ではスケールアウトすなわちサーバの追加について学びます。

6.1 背景

引越しのたとえから少し離れて、地域社会の計画立案者になったつもりで考えてみましょう。密集した住宅地にある家から都心へと通勤しているなら、何車線もある道路はきっとなじみが深いことでしょう。そこではチーターのように早く車が行き交うことも、あるいはチーターの銅像のように流れが滞ることもあります。このような状況でなすべきことは、次のいずれかです。

- 制限速度を上げる。
- 車線を増やす。

このような問題はすでに何度も聞いたことがあるかもしれませんが、単純化されすぎているきらいもあります。むやみに速度を上げても法律や自然法則の制約を受けるし、車線の数を無制限に増やすこともできません。このような制限によって､交通量の限界というものがおのずと明らかになってきます。

しかも、先に挙げたような対策が問題の解決につながらないことさえあります。人を効率的に動かす際に障害となるのは、制限速度と車線数だけとは限りません。例えば、道路が交差あるいは合流する際に車の流れは妨げられます。高速道の出入り口では減速や車線変更が必要になり、事故や工事のために車線数が制限されることもあります。しかし最大のボトルネックであり最も問題になるのは、実は目的地にあります。1つの道路に大量の車が走っているのも問題ですが、大量の車がみな1か所に向かっていることの方がより大きな問題です。これは市街地そのものに問題があると言えます。このような状況では、駐車場などのスペースが十分に用意されていることを願うしかありません。

コンピュータ環境にも同様の問題が存在します。車線数はアプリケーションが受け入れ可能な同時接続数にたとえることができ、市街地はリソースの集合体に相当します。道路の交差点はネットワーク機器を表し、車はユーザが市街地で何か処理を行うために送信したリクエストを表します。このような状況下で、車線を追加したり制限速度を上げたりすることを意味するのがスケールアップです。CPU やメモリ、ディスクやネットワークの帯域などをグレードアップすることによってスケールアップは行われます。これはうまく行くこともありますが、限界もあります。1 台のマシンに CPU やメモリ、ディスクやネットワークインタフェースを無限に詰め込めるわけではありません。無理やり詰め込んだとしても、世界中から殺到するリクエストをすべて処理することは不可能でしょう。

スケールアップの問題点はこれだけではありません。サーバが大規模化するのにつれて、障害から回復するためのコストも増大します。障害時の対応やハードウェアのアップグレードに備えて、システムに冗長性を持たせていつでもバックアップのマシンが待機しているようにする必要があります。つまり、スケールアップは困難であり、しかもすぐに性能向上の限界に達してしまうということが言えます。

成功を収めている Web ビジネスのほとんどでは、スケールアップは必要に迫られて行うのではなく自主的に行っています。ただし、スケールアップを完全に否定するつもりはありません。データベースについてはスケールアップに大きなメリットがあるため、後ほどもう一度触れたいと思います。

6.2　クラスタリングによるスケールアウト

スケールアウトとは、サーバの台数を増加させることを意味します。もちろん、それぞれのサーバには個別に CPU やディスク、メモリ、ネットワークインタフェースなどが装備されています。道路のたとえに戻ると、本当の問題は誰もがみな都心に向かっていることでした。ここでスケールアウトを行うともう 1 つ都心が現れ、半分の車が今までとは別の道を通って新しい都心へと向かうことになります。つまり、交通量を半減できるという利点があるのですが、コストや複雑さ、メンテナンスの手間についても考えてみましょう。

サーバを数百台にもスケールアウトするのは、確かに金銭的にも複雑さの面からも高いコストが伴います。しかしスケールアウトをすべて一度に行う必要はありません。好都合なことに、少しずつスケールアウトすれば複雑さも少しずつしか増加しません。初期段階ではスケールアウトのコストは低く、準備を重ねてゆくにつれて大規模なスケールアウトも可能になってきます。つまり、今準備をしておけば後でより高い処理性能が要求されても容易に対応できるようになるのです。そのためにはインフラストラクチャ上でいくつかの作業を行っておく必要があります。後々の面倒を避けるために、今からスケーラビリティを高めておきましょう。

もちろん、アプリケーション自体の処理性能も適正なものでなければなりません。ハードウェアに投資することによって性能の向上を試みる場合でも、アプリケーションの改善には十分な時間を割くべきです。これについては「8 章　パフォーマンス」で解説します。

デプロイの観点からは、**図 6-1** のようなロードマップが導かれます。2 つまたは 3 つの仮想ホス

トが、同一または別々のマシン上で動作しています。そして1つ1つのMongrelクラスタが都心に相当します。サーバのうち1つは静的プロキシとロードバランサであり、Apacheかnginxが動作しています。これ以外のサーバが、Mongrelのクラスタを通じてRailsアプリケーションによるサービスを提供します。これらのサーバへのデプロイにはCapistranoが使われます。この章では、アプリケーションへの影響を最低限に抑えたうえで、単一のサーバから5台程度へと容易にスケールアウトできるようなアーキテクチャを紹介します。続いて次のようなトピックにも触れます。

- 複数の仮想マシンを実運用環境に追加する方法。
- DNSの`CNAME`フィールドを使い、クラスタ用にサブドメインを用意する方法。
- ApacheやnginxをWebサーバとして使い、負荷分散のためのプロキシとなるWebサーバをセットアップする方法。
- MySQLのクラスタで、マルチマスタまたはマスタ/スレーブの関係を構築する方法。

まずは非常にシンプルなデプロイ形式から始め、徐々にスケールアウトしてWeb 2.0ユーザの群衆にも対応できるようにしてゆきたいと思います。最終的にはWebサーバが別に用意され、Mongrelクラスタにリクエストを振り分けるロードバランサと静的プロキシの役割を果たすことになります。この状態を表したのが**図6-2**です。まず、ユーザはロードバランサを経由して入り口となるサーバにリクエストを送信します。これが静的なコンテンツに対するリクエストであれば、このサーバがコンテンツを返し、処理はそれで終了します。動的なコンテンツへのリクエストなら、Mongrelクラスタのいずれか1つにリクエストが転送されます。そしてクラスタは個々のMongrel

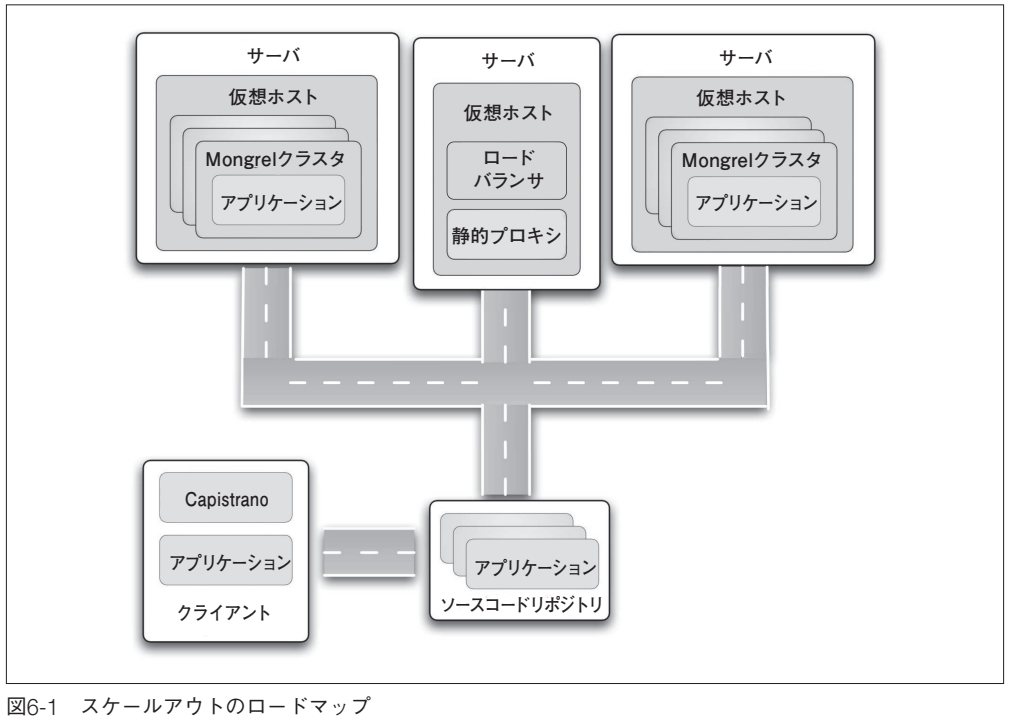

図6-1 スケールアウトのロードマップ

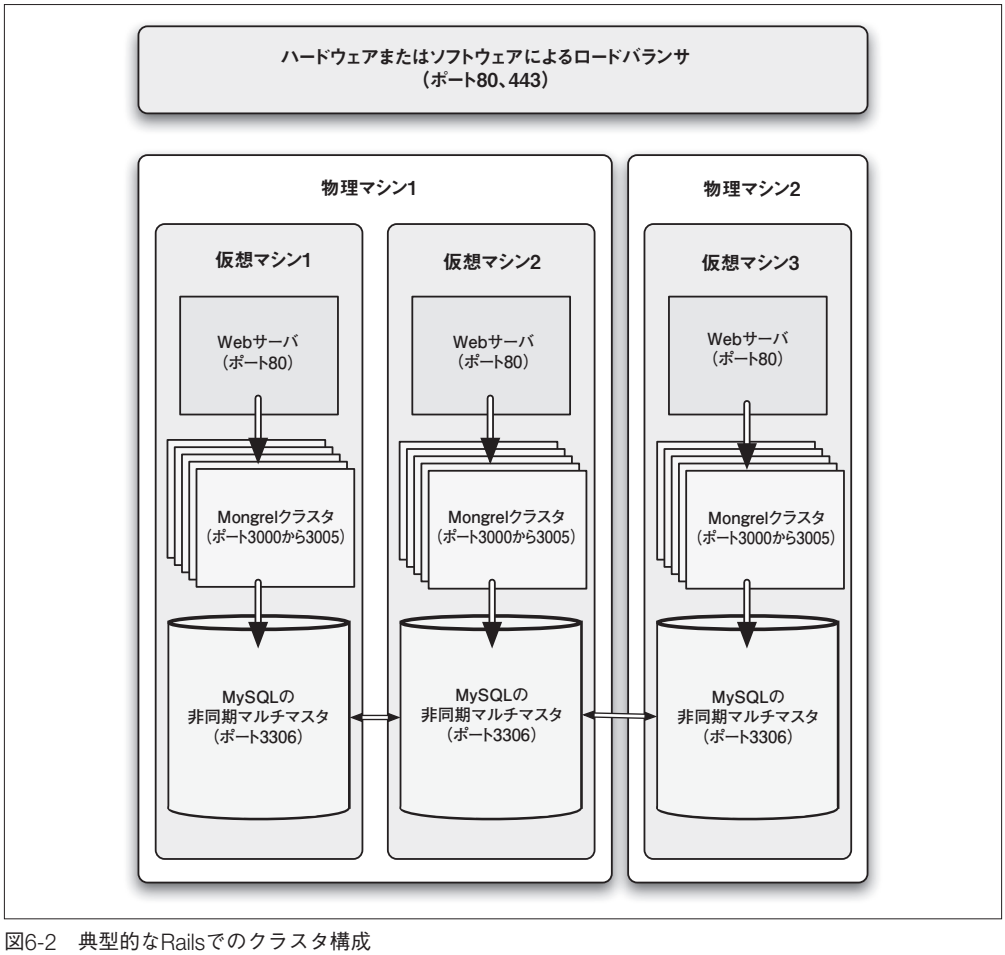

図6-2　典型的なRailsでのクラスタ構成

サーバのうち1つに対してさらにリクエストを転送します。Mongrelのバックエンドではデータベースサーバが実行されていることでしょう。

6.2.1　作業の準備

VPS（Virtual Private Server：仮想的な専用サーバ）のホスティングサービスを受けられないなら、自分のマシン上に仮想マシンをいくつか用意します。Windows上ではVMwareやMicrosoft Virtual PCが非商用目的に限り無料で利用できます。LinuxではVMware、Xen、OpenVZなど、商用のものも無料のものも用意されています。Mac OS XではParallelsやVMwareがありますが、いずれも有料です。どれを購入したらよいか分からない場合は、無料体験版が用意されていないかどうかチェックしてみましょう。筆者のお勧めはVMwareですが、どの仮想化ソフトでもかまいません。

ホスティングによるVPSも、自分で用意した仮想マシンも同じように機能します。あたかもコ

ンピュータ上に別のコンピュータが存在するかのように、仮想マシンに対してオペレーティングシステムをインストールできます。例えばホスティングによるVPSでどの程度のメモリやディスクが必要か分からない場合は、自分で用意した仮想マシン上でさまざまな環境を再現してみることができます。256メガバイトのメモリが確保されたXenによるVPS環境を年間契約してしまう前に、自分のマシン上でVMwareを使って256メガバイトの仮想マシン環境でアプリケーションの動作を確認してみましょう。

VPS上ではどんなLinuxディストリビューションも利用できますが、筆者はCentOSやFedoraなどのRed Hat系のものが最も使いやすいと考えます。これらはVPSプロバイダの多くで利用でき、特にRHELとCentOSは広く使われています。なお、Linux上でRailsアプリケーションを動作させるためには、依存関係にある多くのパッケージのインストールが必要になります。最低限でもCやC++のコンパイラ、Ruby、アプリケーションが必要とするGem、Subversion、Apache、MySQLサーバ、テキストエディタ（viやEmacsなど好みのもの）が必要です。場合によっては、オペレーティングシステムのインストール時に「Webサーバ」あるいは「開発」というオプションを指定するだけで必要なものがすべてインストールされることもあります。しかし、「3章　仮想ホストと専用ホスト」を読めば自分でインストールするのもさほど難しくはないはずです。

Joeの疑問

なぜスケールアウトに仮想化が必要なのか？

仮想化はスケールアウトに多くの利点をもたらします。まず、アプリケーションがすぐにスケールアウトできるような設定が可能になります。専用のサーバ1つ分の値段で、少なくとも2つ以上のVPSを利用できます。冗長性と高い処理性能の恩恵をすぐに受けることができ、後で専用のサーバに移行する場合でも設定への影響は少なく、その際にサーバを停止しなければならない時間を短縮できます。筆者は専用サーバ上でも仮想マシンを用意してアプリケーションを実行しています。ハードウェアとソフトウェアの進歩によって仮想マシンの性能は向上しており、仮想化による処理性能の低下はメリットと比べれば微々たるものです。

仮想マシンの環境は最低でも2つ必要になるでしょう。ただし、環境は1つ構築すれば後はそれをコピーし、IPアドレスやホスト名などの識別情報を変更するだけで済みます。

仮想マシンが稼動するようになったら、その内の1つに対してデータベースの作成とアプリケーションのデプロイを行います。デプロイにはCapistranoが使われます。まずは最もシンプルな構成でデプロイを行っており、Mongrelを起動してWebブラウザから直接Mongrelにアクセスするという形態になります。

ここではどんなアプリケーションを使ってもかまいませんが、（空のアプリケーションではなく）何らかの処理は行う方がよいでしょう。もしこのようなアプリケーションがないなら、“*Agile Web Development with Rails*”（`http://www.pragprog.com/titles/rails2/source_code`）で公開されているサンプルコード†を利用するという方法もあります。

† 訳注：日本語版『Railsによるアジャイル Webアプリケーション開発 第2版』のサンプルコードは次のURLから入手できます。
`http://www.ohmsha.co.jp/data/link/978-4-274-06696-2/`

Linux や Apache、MySQL の管理については専門書が多数出版されているので、Rails でのデプロイというトピックに集中するためにもここでの解説は省略したいと思います。セキュリティや処理性能の向上については、必要に応じて他の資料を読まれることをお勧めします。

6.3 仮想マシン環境のコピー

読者の環境には、さまざまなロール（役割）を持つサーバが多数存在するはずです。同じロールに属するサーバは、すべて同じように設定されているべきです。そうすればメンテナンスが容易になり、問題の発生は減少し、書かなければならないドキュメントも減ることでしょう。設定を一致させるのが面倒そうだと思った読者は、ぜひ仮想化によるメリットを体験してください。

仮想マシン環境はハードディスク上のファイルとして保存されており、移動やコピー、削除、バックアップの作成なども簡単に行えます。つまり仮想マシンのファイルをコピーするだけで、まったく同じように設定された複数の仮想マシン環境を作成できます。設定が一致した状態を保つ方法としては以降で示すようにいくつか考えられており、仮想マシンのサイズや変更が発生する時期に応じて使い分けられます。

6.3.1 手作業による環境全体のコピー

仮想マシン環境を初めて用意する場合には、まず完全な環境を１つ用意します。すべてのソフトウェアやパッチ、依存するライブラリなどをインストールして設定を行い、単体のサーバとして動作できるようにします。そして仮想マシンのファイルを必要な数だけコピーします。以上の操作をそれぞれのロール（アプリケーションサーバ、Web サーバ、データベースサーバ）について行います。

6.3.2 ツールによる設定作業の自動化

Capistrano を使うと、複数のサーバに対する設定の変更を自動化できることがよくあります。これまでの章でも見てきたように、Capistrano のスクリプトは Ruby の機能を使うことができ、サーバのロールごとに異なる設定を使い分けることも可能です。

6.3.3 仮想化ソフトウェアによるコピーの自動化

広範囲あるいは大幅な設定変更には、サーバの停止が必要になることもあります。しかし、仮想化ソフトウェアを使えばここでもコピーの概念を適用できます。仮想マシン群の中からどれか１つを取り出し、必要な変更を行ったうえで仮想マシン群に書き戻します。こうすることで、他のすべてのサーバに対して変更点を反映することができます。この手法は、増大した負荷に対処するため新たにサーバを追加する場合にも適用できます。注意点としては、設定は同一であってもそれぞれのサーバは識別できなければならないという点が挙げられます。同期が行われても、IP アドレスやホスト名などといった各サーバに固有の項目は適切に設定されているようにしなければなりません。各サーバに共通のスクリプトを使い、このような作業を自動化できるとよいでしょう。そしてドキュメントも忘れずに作成しましょう。

6.3.4　手作業による各サーバでの設定作業

自動化できない何らかの理由やサーバの停止が望ましくないなどの事情がある場合は、すべてのサーバを渡り歩いて手作業で設定作業を行うこともやむを得ません。もちろん、作業手順はまとめて記録しておくようにしましょう。これが後で大きなメリットをもたらすことでしょう。

6.3.5　仮想マシン環境をオフラインでも保持する

サーバとして直接インターネット上に公開されることのない、オフラインの仮想マシン環境を保持しておくことにも意義があります。初回のデプロイの際に、これを先ほどのように仮想マシンのファイルをコピーするためのコピー元として利用します。このようにして一度仮想マシンをオンラインにすれば、今後はこれを新しいコピー元として利用できます。この仮想マシンに対しては、いつでも設定変更を行えます。オフラインの仮想マシンは、セキュリティホールへの攻撃などによってサーバが被害を受けた場合に安全なバックアップとして利用できます。一度被害を受けたサーバは、もはや信頼することはできません。被害からの復旧に苦労するだけでなく、アプリケーションが保持しているデータへの脅威におびえ続けることになってしまいます。このような場合にオフラインの仮想マシンに対してパッチを適用しておけば、ネットワーク経由の攻撃を受けていない安全な環境を確保でき、必要に応じてオンラインの仮想マシン環境に対してデプロイを行えます。

6.3.6　サードパーティのツール

ここまでに紹介したのは手作業を比較的多く含む手順が多かったのですが、これらをすべて自動化してくれるようなツールが仮想化ソフトウェアに含まれていたり、VPS のプロバイダが提供してくれることもあります。調べてみて、プロバイダに導入を依頼してみるのもよいでしょう。

6.4　ドメイン名とホスト

複数のサーバが稼動している場合、それらの名前を管理するのも重要です。「2 章　共有ホスト」や「3 章　仮想ホストと専用ホスト」で、Web ブラウザを使って自分のドメイン名を管理できるということを紹介しましたが、変更がインターネット上のすべてのホストに反映されるには数時間から数日もかかります。クラスタでのドメイン名の管理は単一のサーバの場合と比べて若干面倒ですが、次に示すことを知っておけば決して不可能ではありません。

A レコード

ほとんどの DNS では、例えば `brainspl.at` と `www.brainspl.at` の 2 つをデフォルトで管理しています。主となるドメインすなわちこの例では `brainspl.at` は、DNS 用語では A（Address の略）レコードと呼ばれます。このレコードによって、`brainspl.at` というドメイン名が IP アドレスへと最も低いレイヤで直接結び付けられます。

CNAME レコード

CNAME レコードは A レコードのエイリアス（別名）のようなものです。対象となる A レコードは自分のものでも他人のものでもかまいません。エイリアスによって、サーバが提供するサー

ビスの種類を表すことがよくあります。例えばwww.brainspl.atがbrainspl.atのエイリアスである場合、このホストはHTTPサーバであるということが示されます。そしてftp.brainspl.atはおそらくFTPサーバでしょう。名前の付け方には長い間守られ続けてきた慣習†があるので、可能な限り守るようにしましょう。

Joeの疑問

Aレコードと CNAMEレコードのどちらを使うべき？

AレコードとCNAMEの違いについて筆者も調べてみたのですが、CNAMEレコードの使用（あるいは乱用）について多くの意見が飛び交っており、さまざまな場合での長所と短所についても議論が交わされていました。言葉遊びに付き合うつもりはなかったので、筆者の保守的な性向に基づいて以下のようなルールに従うことにしました。

- 名前からIPアドレスの対応付けにはAレコードを使います。
- CNAMEレコードはAレコードの別名として使います。

6.4.1 名前の付け方

ドメイン名の仕組みについて学んだら、どのような名前が必要になるのか次に考えてみましょう。公開されているサーバについては、ロードバランサなども含めてすべてに名前を付けましょう。本書ではこれ以上詳しくは触れませんが、実運用環境では名前を付け公開する際に何らかのファイアウォールの準備も必要です。

ロードバランサ、2台のWebサーバ、そしてデータベースサーバからなる構成について考えてみます。ロードバランサはインターネットに対して直接公開されますが、データベースサーバは公開されません。したがってロードバランサについては名前をDNSに登録する必要がありますが、データベースサーバにはその必要がなく（それゆえにdb.brainspl.atといった名前は存在しません）、名前を持つとしてもローカルのイントラネット環境にだけあれば十分です。また、ロードバランサは2台のWebサーバに対してリクエストを転送します。デバッグや設定、テストなどのために個々のWebサーバに（直接またはロードバランサを経由して）アクセスすることがあるため、これらにも名前が必要です。合計すると、ロードバランサとWebサーバで3つのAレコードが必要になります。また、筆者のサイトにアクセスするユーザの便宜を図ってwwwというエイリアスも設定します。そして今後キャッシュサービスを利用することを想定し、もう1つCNAMEレコードを登録することにします。まとめると次のようになります。

Aレコード

- brainspl.at => 999.999.999.100 —— ロードバランサ
- www1.brainspl.at => 999.999.999.101 —— Webサーバ(1)
- www2.brainspl.at => 999.999.999.102 —— Webサーバ(2)

† http://ja.wikipedia.org/wiki/DNSに詳しい説明があります。

CNAME レコード

- www.brainspl.at => brainspl.at —— ロードバランサのエイリアス
- content.brainspl.at => content.contentcache.com —— キャッシュサービスのエイリアス

2つめの CNAME レコードは、他サイトの A レコードを参照しています。キャッシュサービスによって音楽や画像、動画などの大容量のコンテンツがキャッシュされることを想定しています。

A レコードや CNAME レコードを設定する際には、TTL（Time To Live：生存時間）パラメータの値を指定する必要があります。この値は、DNS に対する設定変更の有無を DNS サーバがチェックする間隔を表します。小さな値（例えば 30 分）が指定されていると、設定を変更した際すぐにその内容が DNS サーバに反映され、サイトへの影響を最小限にとどめることができます。一方大きな値（例えば 7 日）を指定すると、ブラウザなどのクライアントソフトウェアが DNS に問い合わせを行う回数を削減でき、DNS サーバの処理性能を向上できます。したがって、初めのうちは失敗してもいいように小さな値を指定し、満足のいく設定を行えたら 24 時間以上の大きな値へと変更するのがよいでしょう。

サーバに名前を付けたら、いよいよアプリケーションのデプロイです。

6.5 複数のホストへのデプロイ

Rails と Capistrano には、デプロイの際にさまざまな方法を選択できます。Capistrano は標準で3つのロール（役割）をサポートしています。

- web ロールは静的コンテンツを提供するサーバを指します。Apache や nginx などが該当します。
- app ロールは Rails アプリケーションを実行するサーバを指します。Mongrel などが該当します。
- db ロールはデータベースサーバを指します。MySQL などが該当します。

これらのサーバのそれぞれについて、Capistrano は指定されたファイルだけをデプロイします。そのためには、デフォルトの挙動を上書きする必要があります。サーバが 1 台だけの場合には、そのサーバがすべてのロールを受け持っていたと思われます。次に示す deploy.rb はサーバが 1 台の場合の設定です。後ほどこの設定を、複数台のサーバへと拡張してゆきます。

```
# カスタマイズされたdeploy.rb
set :application, "brainspl.at"
set :user, "ezra"
set :repository, "http://brainspl.at/svn/#{application}"
set :deploy_to, "/home/#{user}/#{application}"

role :web, "www1.brainspl.at"
role :app, "www1.brainspl.at"
role :db,  "www1.brainspl.at", :primary => true
```

6.5.1 負荷分散の選択肢

サーバが2台ある場合、複数のデプロイ方法が存在します。次の3つの方法から1つを選ぶのが妥当と考えられます。

データベースサーバを独立させる

この方法はシンプルで、アプリケーションにもよりますが個々のリクエストを処理する能力は最も高くなります。トランザクションを多用し、動的なデータへの依存度が高い場合にはこの方法を使うべきです。データベースを独立させたので、サーバの処理能力（高速なハードディスクが多く用意されていることでしょう）を独占できます。また、公開されたネットワークインタフェースからデータベースを隔離するためセキュリティも向上します。設定は次のようになります。

```
# deploy.rb(抜粋)
# ...
role :web, "www1.brainspl.at"
role :app, "www1.brainspl.at"
role :db,  "internal.brainspl.at", :primary => true
```

Web サーバを独立させる

この方法では Web サーバは多くのメモリとネットワークの帯域を活用でき、キャッシュや静的ページの提供に専念できるようになります。もしアプリケーションの中で静的コンテンツが大きな割合を占めており、ソーシャルブックマークなどを経由した大量のアクセスにも耐えたいと願うなら、専用の Web サーバを用意するというのは有力な選択肢です。もう1台のサーバはイントラネット内に隠され、アプリケーションサーバとデータベースサーバの役割を果たします。この場合でもデータベースと Web サーバを隔離できるというセキュリティ上のメリットは維持されます。しかし、Web サーバとアプリケーションサーバが異なるマシン上に置かれることによって、Rails が動的に生成したページをキャッシュし、それを直接クライアントに提供できるというメリットは失われてしまいます。この問題に対しては、クラスタ化されたファイルシステムや共有ファイルシステムを導入するか、アプリケーションサーバのマシンにもう1つ Web サーバを用意するといった解決策が考えられます。Web サーバを独立させた設定は次のようになります。

```
# deploy.rb(抜粋)
# ...
role :web, "www1.brainspl.at"
role :app, "internal.brainspl.at"
role :db,  "internal.brainspl.at", :primary => true
```

先ほどの設定との違いはわずかですが、もたらす結果の違いはきわめて重大です。実行しようとしているアプリケーションの種類や、利用可能なハードウェアなどを考慮して選択を行うようにしましょう。しかしこの2つの方法がもたらすメリットをしのぐほどの急激な成長が見込まれるなら、これから紹介する3つめの選択肢を検討するべきです。最終的にはこの3つめの方法を取らざるを

Joeの疑問

データベースに対してスケールアップを行うだけでよいのでは？

もちろん、スケールアップを行ってもかまいません。データベースはしばしば処理のボトルネックになり、最高速のディスクや多量のメモリを搭載することによって大幅な性能向上を期待できます。そのため、データベースサーバに対してスケールアップを行うというのは理にかなっています。また、RAID-1 や RAID-5、RAID-10 などを使うことによって、ソフトウェアによるクラスタリングと同等の冗長性を備えることができ、より高速に動作する可能性もあります。ただし、スケールアップのためのハードウェアのコストは無視できず、スケールアップの限界や影響についても知っておく必要があります。

得ないということも考えられます。

クラスタリング

ここでは 2 つのサーバがともにすべてのロール（Web サーバ、アプリケーションサーバ、データベースサーバ）を受け持ちます。この方法には、最大限の冗長性を得られるというメリットがあります。片方のサーバが完全に停止してしまっても Web サイトは機能し続けることができ、データの損失も発生しません。また、負荷の種類によっては他の場合よりも高速に処理できることもあります。同時にアクセスするユーザが最大 2 人であるような状況下では、それぞれのサーバに完全なアプリケーションスタックと占有可能なハードウェアが与えられているのと同じ状態であり、リクエストを高速に処理できます。しかしこの方法ではデータベースと Web サーバが同じ環境下に置かれており、データベースが Web 関連のさまざまなバグやセキュリティホールの影響を受けてしまうため、安全性は低下します。

この設定は扱いにくく、特にデータベースについては面倒です。マスタ / スレーブの設定を使って読み出しと書き込みを異なるデータベースに対して行っている場合、どちらがクラッシュしてもアプリケーションに対して影響が発生します。

冗長性を向上させるためには、どちらのデータベースにも読み出しと書き込みがともに行えるようなデータベースクラスタが必要になります。これについては「6.8　MySQL のクラスタ」で詳しく解説します。Capistrano での設定は次のようになります。

```
# deploy.rb(抜粋)
# ...
role :web, "www1.brainspl.at"
role :app, "www1.brainspl.at"
role :db,  "www1.brainspl.at", :primary => true

role :web, "www2.brainspl.at"
role :app, "www2.brainspl.at"
role :db,  "www2.brainspl.at"
```

6.5.2　3台以上のサーバでの設定

Capistrano と Rails の組み合わせはとても柔軟であり、サーバの設定に関して豊富な選択肢が用

意されています。サーバが3台以上になると、設定の組み合わせの数は大幅に増加します。ここではよく使われるものをいくつか紹介します。

6.5.2.1　データベースのクラスタ（サーバ4台）

ここではWebサーバとアプリケーションサーバをそれぞれ独立させ、データベースはクラスタ化しています。この設定ではトランザクションや動的なデータの処理に重点が置かれています。データベースの性能がボトルネックになっている場合や、データベースにフェイルオーバーの機能を持たせたい場合は次のように設定するとよいでしょう。

```
# deploy.rb(抜粋)
# ...
role :web, "www.brainspl.at"
role :app, "app.brainspl.at"
role :db,  "db1.brainspl.at", :primary => true
role :db,  "db2.brainspl.at"
```

6.5.2.2　Webサーバのクラスタ（サーバ5台）

クラスタ化されたWebサーバと、アプリケーションサーバそしてスケールアップが施されたデータベースサーバからなる構成です。静的なコンテンツの提供に重点が置かれていますが、データベースサーバも強力です。

```
# deploy.rb(抜粋)
# ...
role :web, "www1.brainspl.at"
role :web, "www2.brainspl.at"
role :web, "www3.brainspl.at"
role :app, "app.brainspl.at"
role :db,  "bigdb.brainspl.at", :primary => true
```

6.5.2.3　サーバ10台

図6-3は特定の重点を置かない完全なクラスタの例です。ここでは2台の高性能なデータベースサーバがアプリケーションにデータを提供しています。3台以上ではサーバの追加に見合うメリットを得にくいためです。より多くのデータベースサーバをクラスタ化したい場合は、「6.8.1.3　問題3：クラスタリングとSharding」で紹介するShardingを利用するとよいでしょう。

```
# deploy.rb(抜粋)
# ...
role :web, "www1.brainspl.at"
role :web, "www2.brainspl.at"
role :web, "www3.brainspl.at"
role :app, "app1.brainspl.at"
role :app, "app2.brainspl.at"
```

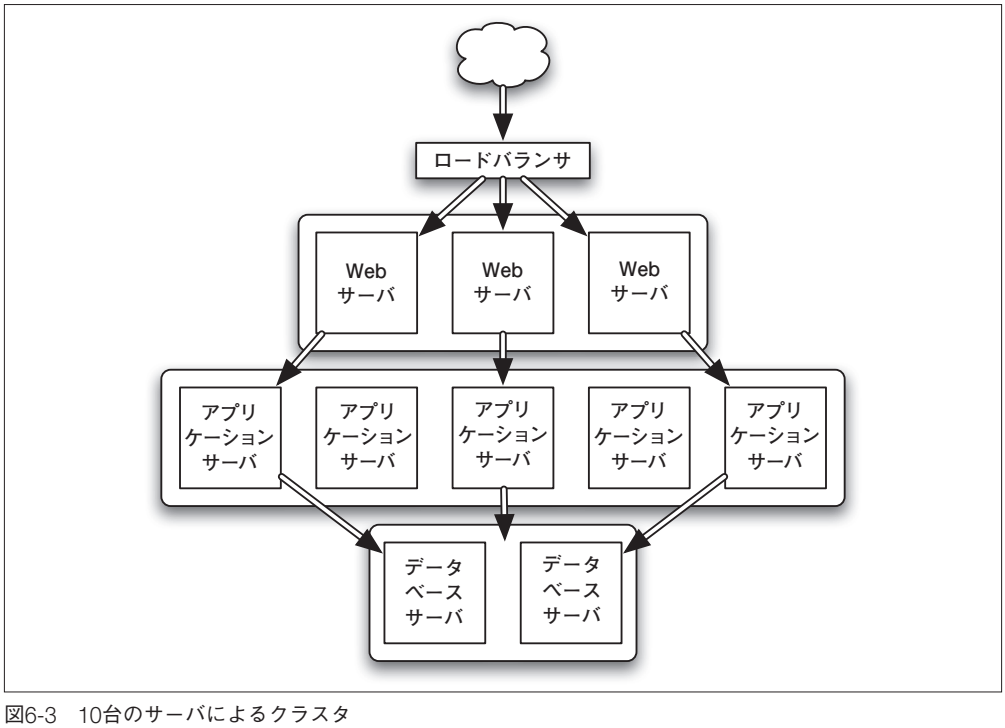

図6-3 10台のサーバによるクラスタ

```
role :app, "app3.brainspl.at"
role :app, "app4.brainspl.at"
role :app, "app5.brainspl.at"
role :db,  "bigdb1.brainspl.at", :primary => true
role :db,  "bigdb2.brainspl.at"
```

6.5.3 Webサーバとアプリケーションサーバの違い

ここまでの説明の中で、web そして app というロールを何度も目にしてきました。また、Mongrel は Web サーバとしては最善ではないということも示唆してきました。ここでは2つのロールの違いについてもう少し考察してみましょう。

6.5.3.1 Webサーバ

Web サーバはリクエストを高速に振り分けたり、静的なコンテンツの提供やキャッシュを得意としています。したがって、Web サーバはクライアント側から見てアプリケーションサーバの手前に位置するプロキシとして優れた役割を果たしてくれます。Rails コミュニティではしばらくの間 lighttpd を使ってみようという動きがありましたが、その後すぐに Apache と nginx という2つのサーバのいずれかに落ち着き、これらを web ロールでのサーバとして推奨するようになりました。Apache はスケーラブルかつ高機能であり、きわめて高い信頼性を備えています。しかし Apache にはあまりにも多くの機能が用意されているため、不必要なものについても何らかの設定を行わな

ければならないという難点もあります。これに対してロシア生まれの nginx も高い能力を持ち、Mongrel クラスタのフロントエンドとして好適です。しかも設定項目はシンプルであり、高速で、システムリソースの消費量も多くはありません。仮想ホストではメモリの量が限られており、ソフトウェアの効率化が重要な意味を持つため、nginx は非常に優れた選択肢です。

6.5.3.2 アプリケーションサーバ

アプリケーションサーバは、コードの安全な実行と実行環境の管理を主眼としています。Rails では単一スレッドのアーキテクチャが採用されているため、実行環境が複数必要になります†。1 つの Mongrel インスタンスは一度にリクエストを 1 つずつしか処理できません。また、Rails では shared-nothing（何も共有しない）という方針がとられており、リクエストが同時に到着してもそれぞれの実行環境は隔離されます。

Rails でのアプリケーションサーバには次の 2 つの問題があります。

- Mongrel での静的コンテンツを提供する処理は最適化されておらず、Rails のコントローラに不必要な負荷がかかってしまいます。
- 実運用向け環境では複数のアプリケーションサーバがそれぞれ異なるポートを使って実行されており、これらに対してリクエストを振り分けるルータのようなものが必要になります。

そのため、リクエストの振り分けや静的コンテンツの提供あるいはキャッシュに特化した Web サーバが必要になります。同時に、動的なコンテンツの提供に最適化されたアプリケーションサーバも望まれます。Mongrel は処理速度や安定性、セキュリティなどの点で優れており、Rails によるコンテンツの提供にはうってつけです。一方 FastCGI は Mongrel よりもわずかに高速ですが、安定性の問題や設定の面倒さのためにほとんど使われていません。

さて、ここまでの時点で Mongrel はクラスタとして構成され、サービスとして動作しているでしょうか。続いては Mongrel と対をなす Web サーバについての解説に移ります。

6.6 Apache による負荷分散

Apache はおそらく、オープンソース史上最も大きな成功を収めたプロジェクトと言えるのではないでしょうか。Apache は世界最大級の Web サイトでも数多く実行されており、巨大なコミュニティが築かれています。また、非営利団体 Apache Software Foundation が Apache を支えています。今後も長い間、Apache が Web サーバのデファクトスタンダードであり続けるであろうことは確実です。

しかしこれほどの大きな成功とは裏腹に、Apache のインストールや設定、管理は容易ではないとよく言われます。これは物の見方の問題でもあります。ある人は「Apache は複雑で、セットアップは困難だ」と言い、またある人は「Apache は信じられないほど柔軟で、最も使いやすいサーバの 1 つだ」と言います。どちらの意見が真実であっても、Apache を使うと決めたならセットアップを

† 訳注：Rails はバージョン 2.2.0 でスレッドセーフになりました。