19

物理ティアと配置

概要

アプリケーション アーキテクチャの設計は、モデル、ドキュメント、およびシナリオという形で存在しています。しかし、アプリケーションは物理環境に配置する必要があるので、アーキテクチャに関する決定の一部がインフラストラクチャの制限事項によって実現できなくなることがあります。そのため、アプリケーションの設計プロセスでは、提案された配置シナリオとインフラストラクチャを検討する必要があります。この章では、分散スタイルと非分散スタイルなど、さまざまな種類のアプリケーションの配置に使用できるオプション、アプリケーションの拡張方法、およびパフォーマンス、信頼性、セキュリティの各問題についてのガイダンスとパターンについて説明します。設計プロセスの一環としてアプリケーションで可能な配置シナリオを考慮することで、技術的なインフラストラクチャの制限事項によりアプリケーションを正常に配置できなかったり設計要件を達成できなかったりする事態を回避できます。

配置に関する方針を選択する際には設計のトレードオフが発生します。たとえば、プロトコルやポートの制約が存在する場合や、配置先の環境で特定の配置トポロジがサポートされていない場合があります。設計の早い段階で配置に関する制約を特定して、後で予想外の事態が発生しないようにします。また、ネットワーク チームとインフラストラクチャ チームのメンバーにも、設計プロセスに協力してもらいます。配置に関する方針を選択する際は、次の項目について理解します。

- 配置先の物理環境
- 配置環境に基づいたアーキテクチャや設計に関する制約
- 配置環境がセキュリティとパフォーマンスに及ぼす影響

分散配置と非分散配置

配置に関する方針を作成する際には、まず分散配置と非分散配置のどちらのモデルを使用するかを決定します。限られたユーザーがアクセスする、組織内で使用する単純なイントラネット アプリケーションを構築する場合は、非分散配置の使用を検討します。スケーラビリティと保守容易性を最大限に高める必要がある、より複雑なアプリケーションを構築する場合は、分散配置の使用を検討します。

非分散配置

非分散配置では、データ ストレージ機能を除くすべての機能とレイヤーが 1 台のサーバーに配置されます (図 1 参照)。

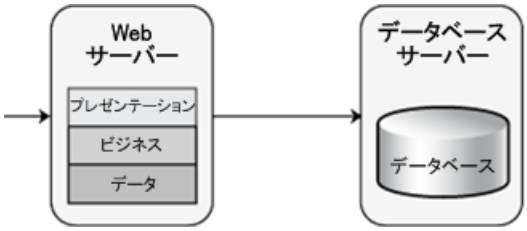

図 1

非分散配置

この手法のメリットは構造が単純なことで、必要な物理サーバー数を最小限に抑えられます。また、レイヤー間の通信がサーバーやサーバー クラスター間の物理的な境界を越える必要がある場合に発生するパフォーマンスへの影響を、最小限に抑えられます。1 台のサーバーを使用することで、通信に関するパフォーマンスのオーバーヘッドを最小限に抑えられますが、他の領域でパフォーマンスが低下することがある点に注意してください。すべてのレイヤーでリソースが共有されるので、あるレイヤーが集中的に使用されると、他のすべてのレイヤーがそのレイヤーの影響を受ける場合があります。また、非常に厳密な運用上の要件に従ってサーバーを汎用的に構築および設計する必要があり、システム リソースを最も多く使用するコンシューマーの最大使用量をサーバーでサポートする必要があります。単一のティアを使用すると、すべてのレイヤーで同じ物理ハードウェアが共有されるので、全体的なスケーラビリティと保守容易性が低下します。

分散配置

分散配置では、アプリケーションのレイヤーが別個の物理ティアに配置されます。ティア型の分散によって、システムのインフラストラクチャが一連の物理ティアに分類され、特定の運用上の要件とシステム リソース使用量に合わせて最適化されたサーバー環境が提供されます。これにより、アプリケーションのレイヤーを異なる物理ティアに分割できます (図 2 参照)。

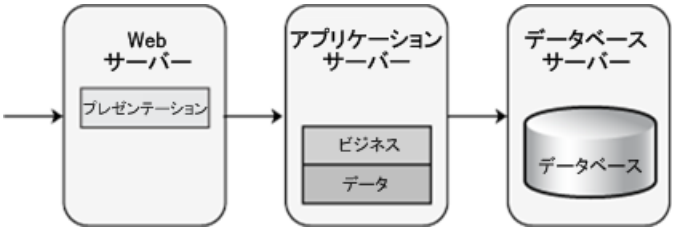

図 2

分散配置

分散アプローチを使用すると、さまざまなレイヤーをホストするアプリケーション サーバーを構成して、各レイヤーの要件を最適な状態で満たすことができます。ただし、コンポーネントの配置を最適化する最大の手段はコンポーネントのリソース使用量についての特性を適切なサーバーに合わせることなので、レイヤーとティアの直接的なマッピングは、多くの場合、配置に関する方針として最適ではありません。

複数のティアを使用すると複数の環境を実現できます。一連の運用上の要件とシステム リソース使用量に合わせて、各環境を最適化できます。このようにすると、リソースの要件に最も合うティアにコンポーネントを配置して、運用上のパフォーマンスと動作を最大限に高めることができます。使用するティアが増えるほど、各コンポーネントで選択できる配置オプションが増えます。分散配置によって柔軟な環境が提供され、この環境ではパフォーマンスの制約事項が発生した場合や処理の要求が増加したときに各物理ティアをさらに柔軟にスケールアウトまたはスケールアップできます。ただし、ティアを追加するほど、複雑さ、配置の手間、およびコストが増大する点に注意してください。

ティアを追加するもう 1 つの目的は、特定のセキュリティ ポリシーを適用することです。分散配置を採用すると、より厳密なセキュリティをアプリケーション サーバーに適用できます。このようなセキュリティを適用するには、たとえば、Web サーバーとアプリケーション サーバーの間にファイアウォールを追加したり、さまざまな認証と承認のオプションを使用したりします。

分散環境のパフォーマンスと設計に関する考慮事項

ティア間でコンポーネントを分散すると、物理的な境界を越えるリモート呼び出しのコストによってパフォーマンスが低下することがあります。ただし、コンポーネントを分散すると、スケーラビリティを拡大する機会が増加し、管理容易性を向上して、時間の経過に伴うコストを削減できます。物理的に分散したインフラストラクチャで実行されるアプリケーションを設計する際には、次のガイドラインを考慮します。

- パフォーマンスの低下を最小限に抑えながらコンポーネントが安全に通信できるように、ティア間の通信パスとプロトコルを選択します。非同期呼び出し、一方向の呼び出し、またはメッセージ キューを使用して、物理的な境界を越える呼び出しを行う際のブロックをできる限り防止します。
- 分散トランザクションのサポートや認証など、設計を簡略化して相互運用性を向上できるサービスやオペレーティング システムの機能の使用を検討します。

- コンポーネントのインターフェイスを簡略化します。タスクの実行に多数の呼び出しを必要とする、粒度の細かいインターフェイス (chatty なインターフェイス) は、同じ物理コンピューター上に配置されている場合に最適な状態で動作します。1 回の呼び出しで 1 つのタスクを実行するインターフェイス (chunky なインターフェイス) は、コンポーネントが別個の物理ティアに分散されている場合に最高のパフォーマンスを発揮します。ただし、インプロセス呼び出しだけでなく他の物理ティアからの呼び出しもサポートする必要がある場合は、インプロセス呼び出し用に粒度の細かいインターフェイスを実装し、他の物理ティアで使用するために、呼び出しをラップして chunky なインターフェイスを提供するファサードを実装することを検討できます。
- 別個の物理クラスターを使用して、エラーが発生する恐れのある他のプロセスから長時間に渡る重要なプロセスを分離することを検討し、フェールオーバーの方針を決定します。たとえば、通常、Web サーバーには、大量のメモリが搭載されて高い処理能力が備わっていますが、ハードウェアの障害時に迅速に置き換えられる堅牢なストレージ機能 (RAID によるミラーリング) が用意されていないことがあります。
- パフォーマンスと可用性を向上するサーバーやリソースを追加する計画を立てるための最適な方法を特定します。
- レイヤー間で物理的な境界を越えて通信する場合は、スケーラビリティとパフォーマンスに影響を及ぼすので、ティア間で状態を管理する方法を検討する必要があります。状態管理の一般的なオプションは次のとおりです。
    - **ステートレス**: ティアを呼び出すと、必要な状態がすべて提供されます。スケーラビリティが向上しやすい一方、多くの場合、クライアントが状態情報を提供する必要があります。
    - **ステートフル**: クライアント要求があるたびに状態が格納または回復されます。より多くのリソースが必要になるのでステートレスよりスケーラビリティの低いソリューションになりますが、クライアントで状態情報を追跡して提供する必要がないので、多くの場合に便利なオプションです。

分散配置にコンポーネントを配置する際の推奨事項

分散配置を設計する際には、各物理ティアに配置する論理レイヤーと論理コンポーネントを決定する必要があります。ほとんどの場合は、プレゼンテーション レイヤーをクライアントまたは Web サーバーに配置し、サービス レイヤー、ビジネス レイヤー、およびデータ レイヤーをアプリケーション サーバーに、データベースを専用のサーバーに配置します。場合によっては、この構成の変更が必要になることがあります。分散環境でコンポーネントの配置先を決定する際には、次のガイドラインを考慮します。

- 必要な場合にのみコンポーネントを分散します。分散配置を実装する一般的な理由は、セキュリティ ポリシー、物理的な制約、共有のビジネス ロジック、およびスケーラビリティです。
- 複数のプレゼンテーション コンポーネントで複数のビジネス コンポーネントが同時に使用される場合、ビジネス コンポーネントをプレゼンテーション コンポーネントと同じ物理ティアに配置します。この配置により、パフォーマンスが最大限に高まり、運用管理が容易になります。
- セキュリティ上の影響によりコンポーネント間の信頼境界が必要になる場合は、プレゼンテーション コンポーネントとビジネス コンポーネントを同じティアに配置しないようにします。たとえば、リッチ クライアント アプリケーションでは、プレゼンテーション コンポーネントをクライアントに配置し、ビジネス コンポーネントをサーバーに配置して、ビジネス コンポーネントとプレゼンテーション コンポーネントを分離する場合があります。
- コンポーネント間の信頼境界が必要になるセキュリティ上の影響がない限り、サービス エージェント コンポーネントは、コンポーネントを呼び出すコードと同じティアに配置します。
- ワークフロー コンポーネントと共に非同期に呼び出されるビジネス コンポーネントは、できる限り他のレイヤーとは別個の物理ティアに配置します。
- ビジネス エンティティは、エンティティを使用するコンポーネントと同じ物理ティアに配置します。

分散配置のパターン

多くのソリューションで採用されているアプリケーションの配置構造には、いくつかの一般的なパターンがあります。アプリケーションに最適な配置ソリューションを決定する際には、まず、一般的なパターンを確認します。さまざまなパターンを十分に理解したら、次にシナリオ、要件、およびセキュリティ上の制約を検討して最も適切なパターンを選択します。

クライアント/サーバーの配置

このパターンは、クライアントおよびサーバーという 2 つの主要コンポーネントで構成される基本的な構造を表します。通常、このシナリオでは、クライアントとサーバーは 2 つの別個のティアに配置されます。図 3 に、クライアントと Web サーバーが通信する一般的な Web アプリケーションのシナリオを示します。

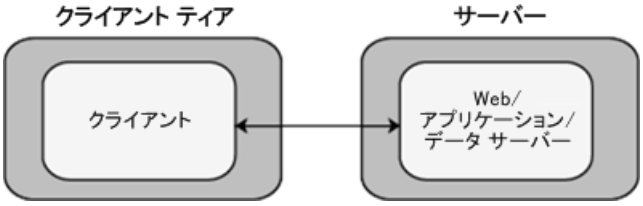

図 3

一般的な Web アプリケーションのシナリオ

アプリケーション サーバーにアクセスするクライアント、または別のデータベース サーバーにアクセスするスタンドアロン クライアントを開発する場合は、クライアント/サーバーのパターンの使用を検討します。

n ティアの配置

n ティアのパターンは、アプリケーションのコンポーネントが 1 台以上のサーバーに分割された汎用的な分散パターンを表します。一般的には、次のセクションで説明するように 2 ティア、3 ティア、または 4 ティアのいずれかのパターンを使用します。1 つのレイヤーのコンポーネントはすべて同じティアに配置する場合が多いですが、必ずしもこのように配置するとは限りません。レイヤーを単一のティアに限定する必要はありません。必要に応じて、複数のサーバーに負荷を分散できます。たとえば、ビジネス ロジックのさまざまな側面を含むサイド バイ サイドのティアを使用できます。

2 ティアの配置

実質的に、これはクライアント/サーバーのパターンと同じ物理レイアウトです。主な違いは、ティア上のコンポーネント間の通信方法です。図 4 のように、すべてのアプリケーション コードがクライアントに配置され、データベースが別個のサーバーに配置されることがあります。クライアントでは、データベース サーバーのストアド プロシージャや最小限のデータ アクセス機能が使用されます。

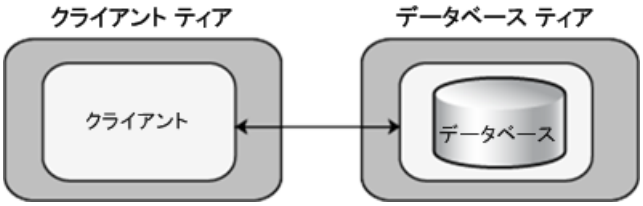

図 4

すべてのアプリケーション コードがクライアントに配置された 2 ティアの配置

アプリケーション サーバーにアクセスするクライアント、または別のデータベース サーバーにアクセスするスタンドアロン クライアントを開発している場合は、2 ティアのパターンの使用を検討します。

3 ティアの配置

3 ティアの設計では、クライアントは別個のサーバーに配置されたアプリケーション ソフトウェアと通信し、アプリケーション サーバーは別のサーバーに配置されたデータベースと通信します (図 5 参照)。これは、ほとんどの Web アプリケーションや Web サービスで非常に一般的なパターンであり、ほとんどの一般的なシナリオに十分対応できます。クライアントと Web ティアまたはアプリケーション ティアとの間にファイアウォールを実装し、Web ティアまたはアプリケーション ティアとデータベース ティアとの間に別のファイアウォールを実装することがあります。

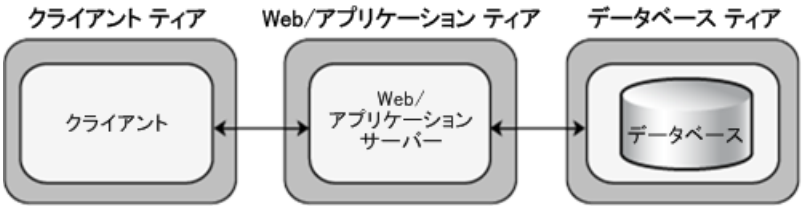

図 5

アプリケーション コードが別個のティアに配置された 3 ティアの配置

すべてのサーバーが社内ネットワーク内に配置されているイントラネット ベースのアプリケーションや、セキュリティ要件によって、一般に公開される Web サーバーやアプリケーション サーバーにビジネス ロジックを実装できないインターネット ベースのアプリケーションを開発する場合は、3 ティアのパターンの使用を検討します。

4 ティアの配置

このシナリオでは、図 6 のように、Web サーバーがアプリケーション サーバーと物理的に分離しています。このシナリオはセキュリティ上の理由で選択されることが多く、Web サーバーが境界ネットワークに配置され、別のサブネット上のアプリケーション サーバーにアクセスします。このシナリオでは、クライアント ティアと Web ティアとの間にファイアウォールを実装し、Web ティアとアプリケーション ティアまたはビジネス ロジック ティアとの間に別のファイアウォールを実装することがあります。

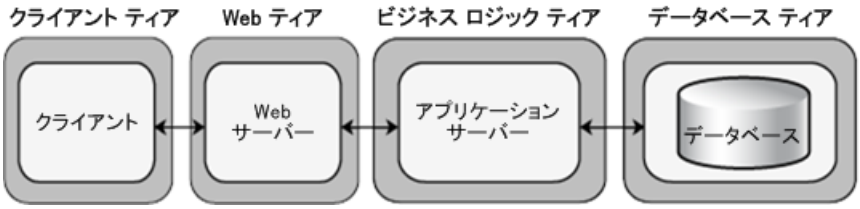

図 6

Web コードとビジネス ロジックが個別のティアに配置された 4 ティアの配置

セキュリティ要件によって、ビジネス ロジックを境界ネットワークに配置できない場合、またはサーバー リソースを大量に使用するアプリケーション コードがあり、その機能を別のサーバーにオフロードする必要がある場合は、4 ティアのパターンの使用を検討します。

Web アプリケーションの配置

セキュリティの懸念事項によりフロントエンド Web サーバーにビジネス ロジックを配置できない場合は、分散配置を Web アプリケーションに使用することを検討します。ビジネス レイヤーにメッセージ ベースのインターフェイスを使用し、パフォーマンスを最適化するために TCP プロトコルとバイナリ エンコードを使用したビジネス レイヤーとの通信を検討します。また、負荷分散によって要求を分散して要求がさまざまな Web サーバーで処理されるようにすることを検討し、スケーラビリティが高い Web アプリケーションを設計する際に

はサーバー アフィニティを回避し、Web アプリケーションで使用するステートレスなコンポーネントを設計する必要もあります。詳細については、この章の後半の「パフォーマンスに関するパターン」を参照してください。

リッチ インターネット アプリケーションの配置

リッチ インターネット アプリケーション (RIA) では、プレゼンテーション ロジックがクライアントに移動されるので、分散アーキテクチャは、RIA で最も使用される可能性が高い配置シナリオです。ビジネス ロジックを他のアプリケーションと共有している場合は、分散配置の使用を検討します。また、ビジネス ロジックにメッセージ ベースのインターフェイスを使用することも検討します。

リッチ クライアント アプリケーションの配置

n ティアの配置では、プレゼンテーション ロジックとビジネス ロジックをクライアントに配置するか、プレゼンテーション ロジックだけをクライアントに配置できます。図 7 に、プレゼンテーション ロジックとビジネス ロジックをクライアントに配置した場合を示します。

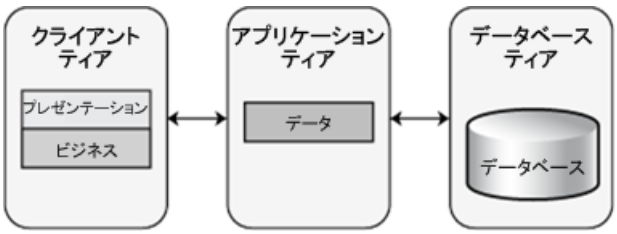

図 7

ビジネス レイヤーがクライアント ティアに配置されたリッチ クライアント

図 8 に、ビジネス ロジックとデータ アクセス ロジックをアプリケーション サーバーに配置した場合を示します。

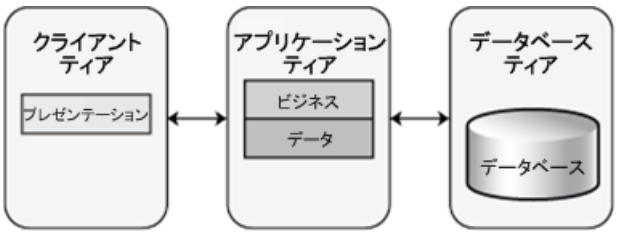

図 8

ビジネス レイヤーがアプリケーション ティアに配置されたリッチ クライアント

パフォーマンスに関するパターン

パフォーマンスに関する配置パターンは、一般的なパフォーマンスの問題に対する実証済みの設計ソリューションを表します。パフォーマンスの高い配置を検討している場合は、スケールアップまたはスケールアウトの手法

を使用できます。スケールアップすると、現在アプリケーションを実行しているハードウェアが強化されます。スケールアウトすると、アプリケーションが複数の物理サーバーに分散され、負荷が分散されます。スケールアウトを採用する予定の場合は、主に負荷分散の方針を使用します。この配置は、通常、負荷分散クラスターと呼ばれ、Web サーバーの場合は Web ファームと呼ばれます。次のセクションでは、これらのパターンについて説明します。スケールアップまたはスケールアウトするタイミングと方法の選択に関する詳細については、この章の後半の「スケールアップとスケールアウト」を参照してください。

負荷分散クラスター

負荷を共有するように構成された複数のサーバーに、サービスまたはアプリケーションをインストールできます (図 9 参照)。この種類の構成は、負荷分散クラスターと呼ばれます。

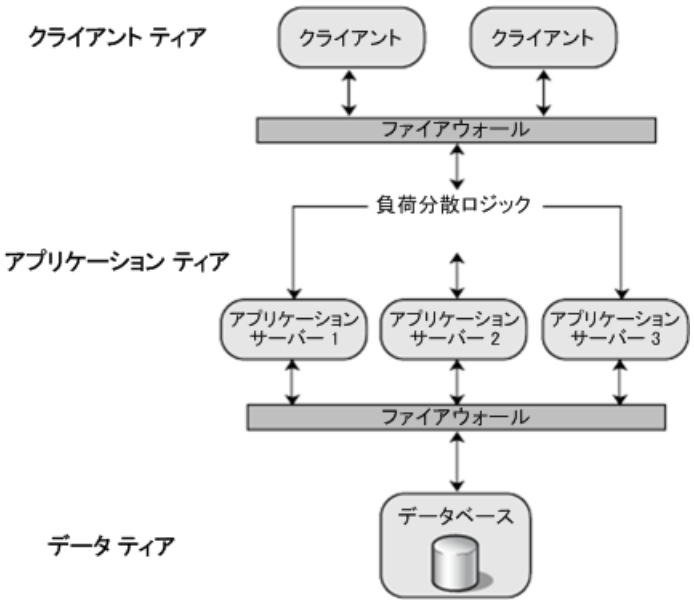

図 9

負荷分散クラスター

負荷分散では、クライアント要求を複数のサーバーに分散することで、サーバー ベースのプログラム (Web サーバーなど) のパフォーマンスが強化されます。この負荷分散テクノロジ (通称、ロードバランサー) では、受信要求を受け取ると、必要に応じて特定のホストにリダイレクトします。負荷分散するように構成された複数のホストでは、同じクライアントから複数の要求が行われた場合でも、同時に異なるクライアント要求に応答します。たとえば、Web ブラウザーでは、単一の Web ページに含まれる複数の画像をクラスター内の異なるホストから取得する場合があります。これによって負荷が分散され、処理速度が向上し、応答時間が短縮されます。使用するルーティング テクノロジによっては、障害の発生したサーバーが検出されてルーティング先の一覧から削除されるので、障害の影響を最小限に抑えられます。単純なシナリオでは、ルーティングはラウンド ロビン方式で行われ、DNS サーバーによって各サーバーのアドレスが順に指定されます。図 10 に、各サーバーで

データ ストアを除くすべてのアプリケーションのレイヤーがホストされている単純な Web ファーム (Web サーバーの負荷分散クラスター) を示します。

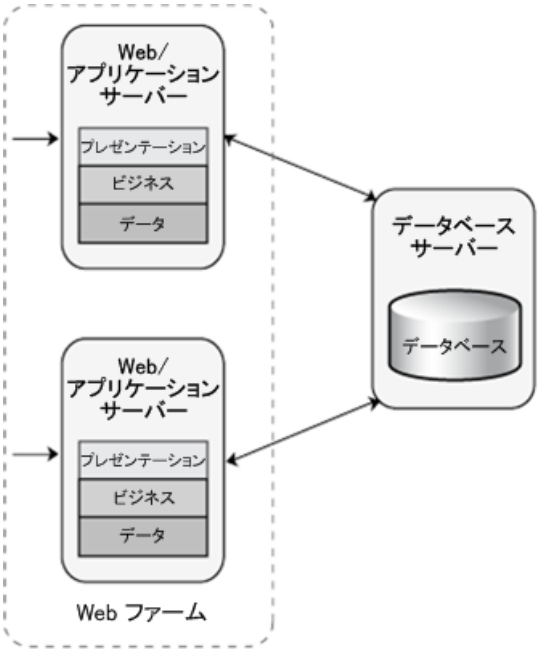

図 10

単純な Web ファーム

負荷分散クラスターは、各クライアントの要求間で情報を追跡して保存する必要がない場合 (つまりステートレスな場合) に、スケーラビリティと効率が最も高くなります。クラスターで状態を追跡する必要がある場合は、アフィニティとセッションの技法を使用しなければならないことがあります。

アフィニティとユーザー セッション

アプリケーションでは、同じクライアントからの要求間でセッション状態が保持されることが前提となっていることがあります。たとえば、Web サーバーでは要求間のユーザー情報を追跡しなければならない場合があります。同一ユーザーからのすべての要求を同一サーバーにルーティングするように Web ファームを構成して (この処理はアフィニティと呼ばれます)、この情報が Web サーバーのメモリに格納されている状態を保持できます。しかし、可用性と信頼性を向上するには、個別のセッション状態ストアを Web ファームと併用して、アフィニティの必要性をなくす必要があります。開発時にインターネット インフォメーション サービス (IIS) 6.0 以降を使用していれば、Web ガーデン モードで動作するように IIS を構成して、アプリケーションの開発時にセッション状態がアプリケーションで適切に処理されるようにできます。

ASP.NET の場合、アフィニティを実装しないときは一貫した暗号化キーと暗号化方法が ViewState の暗号化に使用されるように、すべての Web サーバーを構成する必要もあります。また、システムで Secure Sockets Layer (SSL) 暗号化がサポートされている場合は SSL 暗号化を使用するセッションでアフィニティを有効にし、SSL がサポートされていない場合は SSL 要求に別のクラスターを使用する必要があります。

アプリケーション ファーム

ビジネス レイヤーとデータ レイヤーがプレゼンテーション レイヤーと異なる物理ティアに配置されている場合は、Web サーバーや Web ファームと同様に、これらのレイヤーもアプリケーション ファームを使用してスケールアウトできます。プレゼンテーション ティアからの要求は、サーバー間の負荷がほぼ同じになるようにファーム内の各サーバーに分散されます。各レイヤーの要件、および予期される負荷とユーザー数によっては、ビジネス レイヤーのコンポーネントとデータ レイヤーのコンポーネントを異なるアプリケーション ファームに分離できます。

信頼性に関するパターン

信頼性に関する配置パターンは、一般的な信頼性の問題に対する実証済みの設計ソリューションを表します。配置の信頼性を向上する最も一般的な手法は、フェールオーバー クラスターを使用して、サーバーで障害が発生したときでもアプリケーションの可用性を確保することです。

フェールオーバー クラスター

フェールオーバー クラスターは、あるサーバーが使用できなくなると、障害が発生したサーバーの処理が別のサーバーに自動的に引き継がれて処理が続行されるように構成された一連のサーバーです。図 11 に、フェールオーバー クラスターを示します。

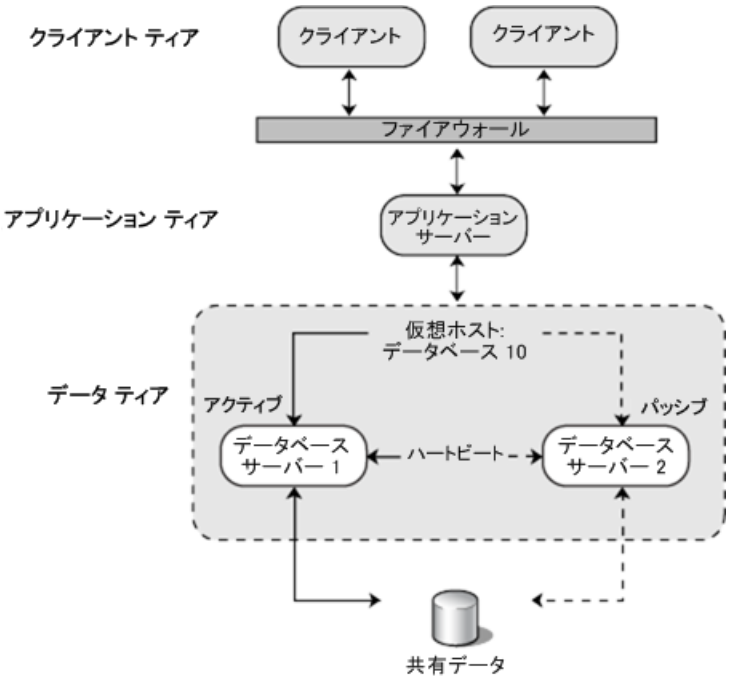

図 11

フェールオーバー クラスター

障害発生時に互いの処理が引き継がれるように構成された複数のサーバーに、アプリケーションまたはサービスをインストールします。障害が発生したサーバーの処理を別のサーバーで引き継ぐ処理は、一般にフェールオーバーと呼ばれています。クラスター内の各サーバーには、スタンバイ サーバーとして認識されているサーバーが少なくとも 1 台あります。

セキュリティに関するパターン

セキュリティに関するパターンは、一般的なセキュリティの問題に対する実証済みの設計ソリューションを提示します。偽装とデリゲートの手法は、最初の呼び出し元のコンテキストをアプリケーションのダウンストリーム レイヤーやコンポーネントに渡す必要がある場合に適したソリューションです。信頼されたサブシステムの手法は、アップストリーム コンポーネントで認証と承認を処理し、単一の信頼された ID でダウンストリーム リソースにアクセスする場合に適したソリューションです。

偽装とデリゲート

偽装とデリゲートの承認モデルでは、各ユーザーに許可されたリソースと操作の種類 (読み取り、書き込み、削除など) は、Windows のアクセス制御リスト (ACL)、またはアクセス先リソース (SQL Server のテーブルとプロシージャなど) の ACL に相当するセキュリティ機能を使用してセキュリティ保護されます。ユーザーは、偽装により各自の本来の ID を使用してリソースにアクセスします (図 12 参照)。この手法ではドメイン アカウントが必要になることがあるので、シナリオによっては適さない場合があることに注意してください。

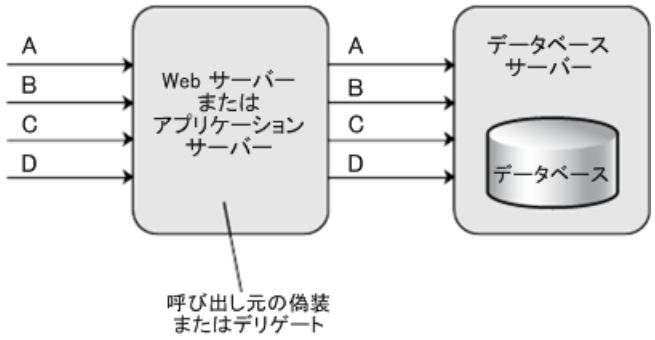

図 12

偽装とデリゲートの承認モデル

信頼されたサブシステム

信頼されたサブシステム (信頼されたサーバー) モデルでは、ユーザーはアプリケーションで定義された論理的な役割に分類されます。特定の役割のメンバーは、アプリケーション内で同じ特権を共有します。操作へのアクセス (通常、メソッドの呼び出し) は、呼び出し元に割り当てられている役割に基づいて承認されます。この役

割ベース (操作ベース) のセキュリティ手法では、操作 (ネットワーク リソース以外) へのアクセスは呼び出し元に割り当てられている役割に基づいて承認されます。役割はアプリケーションの設計時に分析および定義され、アプリケーション内で同じセキュリティ特権や機能を共有しているユーザーをまとめる論理的なコンテナーとして使用されます。中間ティア サービスでは、固定 ID を使用してダウンストリーム サービスやダウンストリーム リソースにアクセスします (図 13 参照)。

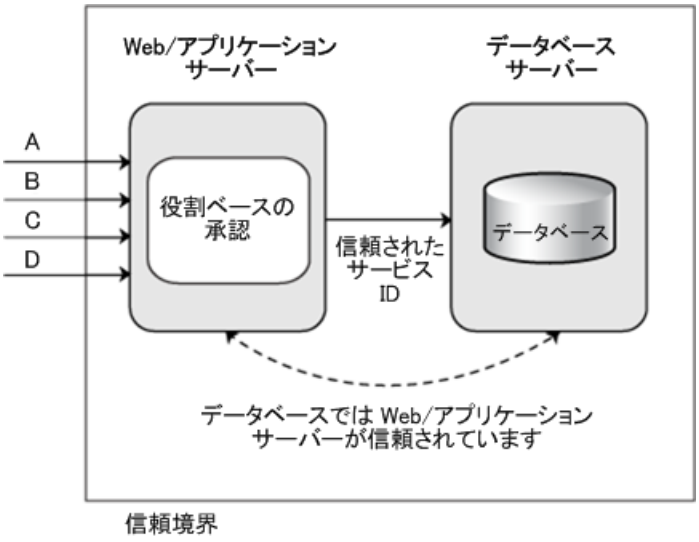

図 13

信頼されたサブシステム (信頼されたサーバー) モデル

複数の信頼されたサービス ID

状況によっては、複数の信頼された ID が必要なことがあります。たとえば、2 つのユーザー グループがあり、一方のグループは読み書き操作の実行が承認される必要があり、もう一方のグループは読み取り専用の操作の実行が承認される必要があるとします。複数の信頼されたサービス ID を使用すると、スケーラビリティに大きな影響を及ぼすことなく、リソースのアクセスと監査をより細かく制御できるようになります。図 14 に、複数の信頼されたサービス ID のモデルを示します。

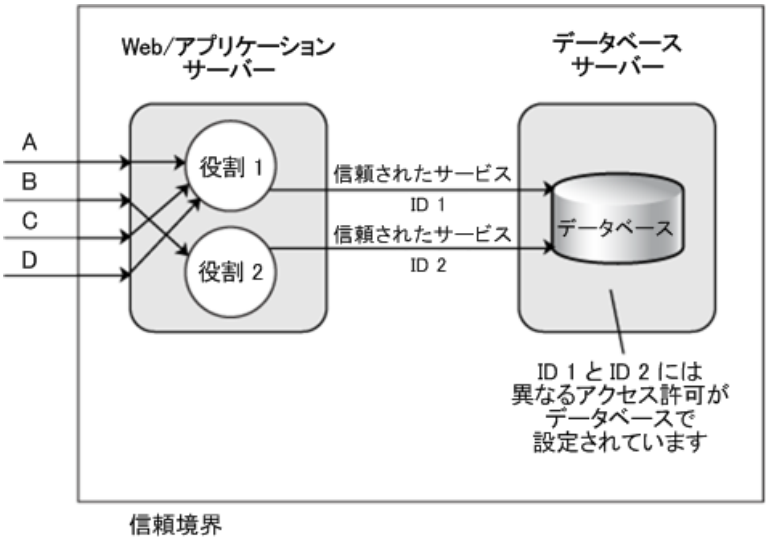

図 14

複数の信頼されたサービス ID のモデル

スケールアップとスケールアウト

スケーリングの手法は、設計に関する重要な考慮事項です。ソリューションのスケールアウトに負荷分散クラスターと分割データベースのどちらを使用する予定でも、選択するオプションは設計でサポートされるようにする必要があります。一般的に、アプリケーションを拡張する際には、スケールアップ (コンピューターの機能拡張) とスケールアウト (コンピューターの台数増加) の 2 つの基本的なオプションを選んで組み合わせることができます。

スケールアップの手法では、プロセッサ、RAM、ネットワーク インターフェイス カード (NIC) などのハードウェアを既存のサーバーに追加して、より高い性能をサポートします。これは単純なオプションで、保守とサポートのコストが増加しないので、ある程度まではコスト効率のよいオプションです。しかし、単一障害点が残り、これはリスクになります。また、一定のしきい値を超えると、既存のサーバーにハードウェアを追加しても目的の結果が得られなくなることがあり、アップグレードによって 1 台のサーバーで論理的に向上できるとされているパフォーマンスの最後の 10% を達成するためのコストは非常に高くなることがあります。

アプリケーションを効率的にスケールアップするには、基になるフレームワーク、ランタイム、およびコンピューターのアーキテクチャもスケールアップする必要があります。スケールアップする際には、アプリケーションのパフォーマンスを制限しているリソースを考慮します。たとえば、メモリやネットワークに関するリソースがパフォーマンスのボトルネックとなっている場合は、CPU リソースを追加しても効果はありません。

スケールアウトの手法では、サーバーを追加し、負荷分散とクラスタリングのソリューションを使用します。スケールアウトのシナリオでは、より多くの負荷を処理できるようになるだけでなく、ハードウェア障害も軽減されます。というのも、1 台のサーバーで障害が発生しても、負荷を引き継ぐことのできる他のサーバーがクラスター内にあるからです。たとえば、Web ファームに複数の、負荷分散するように構成された Web サーバーを配置して、これらのサーバーでプレゼンテーション レイヤーとビジネス レイヤーをホストできます。また、アプリケーションのビジネス ロジックを物理的に分割してそのロジックに別個の、負荷分散するように構成された中間ティアを使用しながら、負荷分散するように構成されたフロント ティアでプレゼンテーション レイヤーをホストすることもできます。アプリケーションが I/O の制約を受けていて、非常に大きなデータベースをサポートする必要がある場合は、データベースを複数のデータベース サーバーに分配できます。一般に、アプリケーションをスケールアウトできるかどうかは、基になるインフラストラクチャよりもアプリケーションのアーキテクチャによって決まります。

スケールアップに関する考慮事項

プロセッサの処理能力の向上やメモリの増設によるスケールアップは、コスト効率のよいソリューションになる場合があります。この手法では、スケールアウトの手法や Web ファームとクラスタリング テクノロジの使用に伴う追加の管理コストが発生しません。まず、スケールアップのオプションを確認し、パフォーマンス テストを実施する必要があります。テストでは、ソリューションをスケールアップすることで、定義したスケーラビ

リティの条件が満たされるかどうか、および必要な数の同時接続ユーザーが許容できるパフォーマンスのレベルでサポートされるかどうかを確認します。実際の成長に応じたシステムの拡張計画を立てる必要があります。

スケールアウトのサポートに関する設計

CPU、I/O、またはメモリのしきい値に達したために、ソリューションをスケールアップしても適切なスケーラビリティが得られない場合は、スケールアウトして追加のサーバーを導入する必要があります。アプリケーションを正常にスケールアウトできるようにするには、次の設計の慣例を考慮します。

- **ボトルネックの特定とスケールアウト**: それ以上スケールアップできない共有リソースは、多くの場合ボトルネックになります。たとえば、複数のアプリケーション サーバーからアクセスする単一の SQL Server インスタンスがあるとします。この場合、複数の SQL Server インスタンスでデータが提供されるようにデータを分割すると、ソリューションをスケールアウトできます。データベース サーバーがボトルネックになることが予想される場合は、初期設計時にデータの分割も考慮すると、後で必要となる労力を大幅に節約できます。
- **疎結合されたレイヤー型の設計の定義**: 整っていてリモートで操作できるインターフェイスを備えた、疎結合されたレイヤー型の設計は、chatty な通信を行う密結合されたレイヤーを使用する設計よりも簡単にスケールアウトできます。レイヤー型の設計には元々クラッチ ポイントがあるので、レイヤーの境界でのスケールアウトに最適です。ここで重要なのは、適切な境界を見つけることです。たとえば、ビジネス ロジックの配置は、負荷分散するように構成された中間ティアのアプリケーション サーバー ファームに簡単に変更できます。

設計の影響とトレードオフ

アプリケーション レイヤー、ティア、またはデータの種類によって異なる可能性がある、スケーラビリティのさまざまな側面を考慮する必要があります。必要なトレードオフを特定して、柔軟性のある側面とない側面を把握するようにします。場合によっては、Web サーバーやアプリケーション サーバーをスケールアップしてからスケールアウトするのは、最適な手法ではないことがあります。たとえば、8 基のプロセッサが搭載されたサーバーを使用できたとしても、おそらく経済的な事情から大規模な 1 台のサーバーではなく、小規模な複数台のサーバーを使用するでしょう。

一方、データの役割やデータの使用方法によっては、スケールアップしてからスケールアウトするのがデータベース サーバーにとって最適な手法になることがあります。負荷分散またはフェールオーバーを実行できるサーバーの台数には制限があり、データベースの分割方法など他の問題によって処理に影響が及びます。また、技術とパフォーマンスに関する考慮事項以外に、運用と管理に関する影響や関連する総保有コストについても考慮する必要があります。

一般的に、その他すべての制約の境界内でコストとパフォーマンスを最適化します。たとえば、4 基のプロセッサを搭載した 2 台のサーバーを使用した場合と比較してコストとパフォーマンスを評価すると、2 基のプロセッサを搭載した Web サーバーまたはアプリケーション サーバーを 4 台使用する方が適していることがあります。ただし、特定の負荷分散インフラストラクチャに配置できるサーバーの最大数、データ センターの電力消費や床面積の制約など、他の制約も考慮する必要があります。

また、サーバー ファームを実装してサービスをホストするには、仮想サーバーの使用も検討します。この手法を使用すると、リソース使用率や投資収益率を最大限に高められるだけでなく、パフォーマンスとコストのバランスを取ることもできます。

ステートレスなコンポーネント

インプロセスの状態が保持されない Web フロントエンドに実装するコンポーネントなど、ステートレスなコンポーネントを使用すると、スケールアップとスケールアウトの両方をより適切にサポートする設計を作成できます。ステートレスな設計を実現するには、アプリケーションでは、多数の設計上のトレードオフが必要になる可能性が高くなりますが、一般にスケーラビリティに関しては、このデメリットを上回るメリットがあります。

データとデータベースの分割

アプリケーションで非常に大きなデータベースを使用していて、I/O がボトルネックとなることが予想される場合は、事前にデータベースの分割を設計するようにします。一般的に、後で分割データベースに移行すると、コストがかかる大量の手直しが必要となり、多くの場合はデータベースの完全な再設計が必要になります。分割には、いくつかのメリットがあります。たとえば、1 つのパーティションにクエリを制限したり (その結果、リソース使用量がわずかなデータに限定される)、複数のパーティションを使用したり (その結果、より多くのディスクでデータを取得できるので、並列処理が強化されてパフォーマンスが向上する) することができます。

ただし、場合によっては、複数のパーティションの使用が適さず、マイナスの結果を招くことがあります。たとえば、複数のディスクを使用する操作には、一元化されたデータを使用した方が効率よく実行できるものがあります。

配置シナリオでデータ ストレージの分割による影響を考慮する際の決定事項は、主にデータの種類によって変わります。関連する要因には、次のようなものがあります。

- **静的で参照用の読み取り専用データ**: この種類のデータの場合、パフォーマンスとスケーラビリティが向上するのであれば、簡単に多数のレプリカを適切な場所で保持できます。このような操作が設計に及ぼす影響はごくわずかで、通常は最適化に関する考慮事項により実行されます。1 台のデータベース サーバーに、論理的に分離して独立したいくつかのデータベースを統合することは、ディスク容量が十分でも、適切な場合と不適切な場合があります。また、このようなデータのコンシューマーの近くにレプリカを配置することも、同様に有効な手法になる場合があります。ただし、

データを複製すると、適切な同期を維持するメカニズムが必要な、緩やかに同期されるシステムが必要になることに注意してください。

- **簡単に分割できる動的な (多くの場合は一時的な) データ**: これは、ショッピング カートなど特定のユーザーやセッションに関連するデータであり、ユーザー A のデータはユーザー B のデータとまったく関連がありません。このデータは、静的で読み取り専用のデータよりもやや扱いが難しくなりますが、この種類のデータも分割できるので、非常に簡単に最適化して分散できます。各ユーザーのレベルに至るまで、データのグループ間に依存関係はありません。このデータの重要な側面は、複数のパーティションにまたがってクエリしないことです。たとえば、クエリを実行してユーザー A のショッピング カートの内容を取得するのは問題ありませんが、特定の商品が含まれたすべてのカートについてのクエリは実行しないようにします。その後の要求が別の Web サーバーやアプリケーション サーバーに対して行われる可能性がある場合は、関連するパーティションに要求の送信先となるすべてのサーバーからアクセスできる必要があります。
- **コア データ**: 一般的にスケールアップしてからスケールアウトする手法が適用されるのは、主にこの場合です。データを同期するのが複雑になるので、通常はこの種類のデータを多数の場所で保持することはお勧めしません。これは、できる限りスケールアップする必要がある典型的な場合であり (理想的には、適切なクラスタリング機能を備えた単一の論理インスタンスのままにします)、スケールアウト以外に選択肢がないときにのみ分割と分散を検討します。分散パーティション ビューなどデータベース テクノロジの進歩により分割はずっと簡単になりましたが、分割は必要な場合にのみ使用するようにします。データベースのサイズが大きくなりすぎたことが原因で分割の使用を決定することはほとんどなく、データの所有者、データ使用の地理的分散、コンシューマーへの近さ、可用性など、他の考慮事項に基づいて決定される方が多いです。
- **後で同期されるデータ**: アプリケーションで使用されるデータには、すぐには同期する必要がないか、まったく同期する必要がないものもあります。このようなデータの好例は、"X を購入したユーザーは Y と Z も購入した" など、小売店のデータです。このデータはコア データから取得されますが、リアルタイムで更新する必要はありません。コア データを分割可能な (動的な) データにしてから静的なデータにする方針を設計することは、非常にスケーラビリティが高いアプリケーションの構築における重要な要素です。

データを移動して複製するパターンについては、「Data Movement Patterns」(http://msdn.microsoft.com/en-us/library/ms998449.aspx、英語) を参照してください。

ネットワーク インフラストラクチャのセキュリティに関する考慮事項

配置先の環境に用意されているネットワーク構造について理解し、フィルタリング規則、ポートの制限、サポートされるプロトコルなどに関するネットワークのベースライン セキュリティ要件を理解します。ネットワークのセキュリティを最大限に高めるための推奨事項は、次のとおりです。

- ファイアウォールとファイアウォールのポリシーがアプリケーションの設計と配置に影響を及ぼす可能性を特定します。ファイアウォールは、インターネットに公開されるアプリケーションを内部ネットワークから分離し、データベース サーバーを保護するために使用する必要があります。ファイアウォールを使用すると特別に構成されたポート経由の通信のみが許可されるので、一部のプロトコルをブロックしたり、通信オプションの一部を使用できないようにしたりすることができます。制限対象には、Windows 認証など、Web サーバーとファイアウォールの内側にあるアプリケーション サーバーまたはデータベース サーバーとの間の認証が含まれます。
- 境界ネットワークの Web サーバーから、またはリッチ クライアント アプリケーションから内部リソースにアクセスできるプロトコル、ポート、およびサービスを検討します。アプリケーションの設計に必要なプロトコルとポートを特定し、追加のポートを開いたり標準に準拠していないプロトコルを使用したりした場合に発生する潜在的な脅威を分析します。
- ネットワークとアプリケーション レイヤーのセキュリティ、および各コンポーネントで処理できるセキュリティ機能に関して立てられたすべての仮定を伝達して記録します。このようにすると、開発チームとネットワーク チームの両者が他方のチームが問題に対処しているであろうと見なすことが原因でセキュリティ コントロールやポリシーが見落とされることがなくなります。
- アプリケーションが使用するファイアウォール、パケット フィルター、ハードウェア システムなどのセキュリティ対策に注意し、これらの対策がネットワークで有効になっていることを確認します。
- ネットワーク構成の変更による影響と、それによってセキュリティが受ける影響について検討します。

管理容易性に関する考慮事項

アプリケーションの配置時に行う選択は、アプリケーションを管理および監視する機能に影響を及ぼします。次の推奨事項を考慮する必要があります。

- 複数のコンシューマーで使用されるアプリケーションのコンポーネントは、すべてのアプリケーションでアクセスできるサーバー ファームやアプリケーション ファームなど、単一の一元的な場所に配置して、重複しないようにします。

- バックアップと復元機能がアクセスできる場所にデータが格納されるようにします。
- 既存のソフトウェアやハードウェアに依存するコンポーネント (特定のコンピューターからしか接続できない専用ネットワークなど) は、物理的に同じコンピューターに配置する必要があります。
- 一部のライブラリやアダプターは、自由に配置するには追加のコストがかかったり、CPU 単位でコストがかかります。そのため、このような機能を一元化してコストを最低限に抑えることをお勧めします。
- 組織内のグループは、各グループがローカルで管理する必要があるサービス、コンポーネント、またはアプリケーションを所有していることがあります。
- System Center Operations Manager などの監視ツールでは、物理コンピューターにアクセスして管理情報を取得する必要があるので、配置オプションに影響を及ぼすことがあります。

関連する設計パターン

次の表に示すように、主要なパターンは、配置、管理容易性、パフォーマンスと信頼性、セキュリティなどのカテゴリに分類されます。各カテゴリの設計を行う際には、これらのパターンの使用を検討します。

カテゴリ	**関連するパターン**
配置	**Layered Application**: システムがレイヤーで構成されたアーキテクチャ パターンです。 **Three-Layered Services Application**: 他のアプリケーションで使用できるサービスを公開しながら、パフォーマンスを最大限に高めるようにレイヤーが設計されたアーキテクチャ パターンです。 **Tiered Distribution**: 物理的な境界を越えて設計のレイヤーを分散できるアーキテクチャ パターンです。 **Three-Tiered Distribution**: 3 つの物理ティアに設計のレイヤーが分散されるアーキテクチャ パターンです。 **Deployment Plan**: インフラストラクチャによって課される制約を考慮した、論理レイヤーを物理ティアに割り当てる手順を表します。
管理容易性	**Adapter**: 一般的なインターフェイスをサポートし、一般的なインターフェイスと、インターフェイスが異なる同様の機能を実装している他のオブジェクトとの間で操作を変換するオブジェクトです。 **Provider**: あらゆるカスタム実装がシームレスにプラグインされるようにするために、クライアント API とは異なる API が公開されるコンポーネントです。インストルメンテーションが用意されている多くのアプリケーションでは、アプリケーションの状態とパフォーマンスに関する情報、およびアプリケーションをホストしているシステムに関する情報の取得に使用できるプロバイダーが

	公開されます。
パフォーマンスと信頼性	**Server Clustering**: 複数のサーバーで負荷を共有し、クライアントでは単一のコンピューターまたはリソースとして認識されるように構成された分散パターンです。 **Load-Balanced Cluster**: 複数のサーバーが負荷を共有するように構成された分散パターンです。負荷分散を行うと、複数のサーバーに負荷を分散することでパフォーマンスが向上するだけでなく、1 台のサーバーで障害が発生しても残りのサーバーで引き続き負荷を処理できるので信頼性も向上します。 **Failover Cluster**: 非常に可用性の高いインフラストラクチャ ティアを提供することで、1 台のサーバーやそのサーバーでホストされているソフトウェアで発生した障害によるサービスの停止を防ぐ分散パターンです。
セキュリティ	**Brokered Authentication**: ブローカーに対して認証します。ブローカーでは、サービスやシステムにアクセスする際の認証に使用するトークンが提供されます。 **Direct Authentication**: アクセスするサービスやシステムに対して直接認証します。 **Impersonation and Delegation**: 一時的に異なる ID を偽装して、異なるセキュリティ コンテキストや資格情報を使用してリソースにアクセスできるようにする処理です。また、別のユーザーの代わりにサービスのアカウントでリモート リソースにアクセスできるようにします。 **Trusted Subsystem**: アプリケーションが信頼されたサブシステムとして機能して、追加のリソースにアクセスします。リソースにアクセスする際には、ユーザーの資格情報ではなくアプリケーション自体の資格情報を使用します。

Layered Application、Three-Layered Services Application、Tiered Distribution、Three-Tiered Distribution、および Deployment Plan の各パターンの詳細については、「配置のパターン」(http://msdn.microsoft.com/ja-jp/library/ms998478.aspx) を参照してください。

Adapter パターンの詳細については、Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 共著『オブジェクト指向における再利用のためのデザイン パターン』(ソフトバンク クリエイティブ、1999 年) の第 4 章「構造に関するパターン」を参照してください。

Provider パターンの詳細については、「Provider Model Design Pattern and Specification, Part 1」(http://msdn.microsoft.com/en-us/library/ms972319.aspx、英語) を参照してください。

Server Clustering パターン、Load-Balanced Cluster パターン、および Failover Cluster パターンの詳細については、「パフォーマンスと信頼性のパターン」(http://msdn.microsoft.com/ja-jp/library/ms998503.aspx) を参照してください。

Brokered Authentication、Direct Authentication、Impersonation and Delegation、および Trusted Subsystem の各パターンの詳細については、「Web Service Security」(http://msdn.microsoft.com/en-us/library/aa480545.aspx、英語) を参照してください。

関連情報

配置に関する方針の設計に役立つ Web リソースに、より簡単にアクセスするには、http://www.microsoft.com/architectureguide (英語) でオンライン版の参考文献を参照してください。

- 承認の技法の詳細については、「Designing Application-Managed Authorization」(http://msdn.microsoft.com/en-us/library/ee817656.aspx、英語) を参照してください。
- 配置シナリオと考慮事項の詳細については、「Deploying .NET Framework-based Applications」(http://msdn.microsoft.com/en-us/library/ee817655.aspx、英語) を参照してください。
- 設計パターンの詳細については、「Microsoft .NET を使用したエンタープライズ ソリューション パターン」(http://msdn.microsoft.com/ja-jp/library/ms998469.aspx) を参照してください。