Microsoft

Melhorando a Performance e Escalabilidade de Aplicações .Net

Arquitetos de Soluções
Microsoft Brasil

Melhorando a Performance e Escalabilidade de Aplicações .Net

Este documento apresenta a tradução de três Capítulos do Guia Melhorando a Performance e Escalabilidade de Aplicações .Net (*Improving .NET Application Performance and Scalability*). O documento completo, em inglês, pode ser encontrado no formato pdf em *http://msdn.microsoft.com/patterns*.

Os capítulos aqui apresentados tratam das principais considerações de Design que devem ser feitas para garantir boa performance e escalabilidade das aplicações .Net.

A premissa básica é que tanto a performance quanto a escalabilidade são conseguidas através do esforço e análise contínua que já começa no início da fase de design e percorre todo o ciclo de vida do desenvolvimento e operação da aplicação.

O documento original contém cerca de 1100 páginas e é uma ótima referência tanto para arquitetos e gerentes de projetos quanto para programadores e profissionais de teste. Estes capítulos iniciais em português são um convite para a sua leitura e curiosidade em prol da melhoria da qualidade das aplicações que desenvolvemos.

Sumário

Capítulo 2 - Modelagem do Desempenho 9
Objetivos 9
Como Usar Este Capítulo 9
Visão Geral 9
Por que modelar o desempenho? 11
Gerenciamento de Riscos 11
Orçamento. 12
O Que Você Deve Saber 12
Práticas Recomendadas 12
Informações no Modelo de Desempenho 12
Entrada 13
Saída 13
Processo 14
Etapa 1: Identificar Principais Cenários. 15
Etapa 2: Identificar Carga de Trabalho. 16
Etapa 3: Identificar Objetivos de Desempenho. 16
Etapa 4: Identificar Orçamento. 17
Etapa 5: Identificar Etapas do Processamento. 18
Etapa 6: Alocar Orçamento. 18
Etapa 7: Avaliar. 19
Etapa 8: Validar 19
Resumo 20
Recursos Adicionais 20
Capítulo 3 - Diretrizes de Design para o Desempenho de Aplicações 23
Objetivos 23
Visão Geral 23
Como Usar Este Capítulo 24
Princípios 24
Princípios do Processo de Design 24
Princípios de Design 25
Considerações de Implantação 27
Considere a Arquitetura da Implantação 28
Arquitetura Não Distribuída 28
Arquitetura Distribuída 28
Use um Design em Camadas 29
Mantenha-se no Mesmo Processo 30
Não Mantenha a Lógica do Aplicativo em um Local Remoto, a Menos que Seja Necessário. 30
Escalabilidade Vertical versus Horizontal 31
Questões de Arquitetura e de Design 34
Acoplamento e Coesão 36
Comunicação 38
Concorrência 44
Gerenciamento de Recursos 46
Armazenamento em Cache 49
Gerenciamento de Estados 52
Estruturas de Dados e Algoritmos 54
Resumo das Diretrizes de Design 55
Considerações sobre Aplicativos de Desktop 56
Considerações sobre o Cliente no Navegador 57
Considerações sobre a Camada da Web 57

Considerações sobre a Camada de Negócio **59**
Considerações sobre a Camada de Acesso a Dados 59
Resumo **60**
Recursos Adicionais **61**
Capítulo 4 - Revisão da Arquitetura e Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade **63**
Objetivos **63**
Visão Geral **63**
Como Usar Este Capítulo **64**
Processo de Revisão de Arquitetura e Design **64**
Implantação e Infra-estrutura **65**
Você Precisa de uma Arquitetura Distribuída? 66
Qual Comunicação Distribuída Você Deveria Utilizar? 66
Você Possui Interações Freqüentes Pelas Fronteiras? 67
Quais Restrições a sua Infra-Estrutura Impõe? 67
Você Considerou as Restrições na Largura de Banda da Rede? 68
Você Compartilha Recursos com Outros Aplicativos? 69
O seu Design Oferece Suporte para Escalabilidade Vertical? 69
O seu Design Oferece Suporte para Escalabilidade Horizontal? 69
Acoplamento e Coesão **71**
O seu Design é Fracamente Acoplado? 71
Quão Coeso é seu Design? 72
Você Usa *Late Binding*? 72
Comunicação **72**
Você Usa Interfaces que Exigem Muitas Interações? 73
Você Efetua Chamadas Remotas? 73
Como Você Troca Dados com um Servidor Remoto? 74
Você Possui Requisitos de Comunicação Seguros? 75
Você Usa Filas de Mensagens? 75
Você Efetua Chamadas de Longa Duração? 75
Você Pode Usar *Application Domains* em vez de Processos? 76
Concorrência **76**
Você Precisa Executar Tarefas Simultaneamente? 77
Você Cria *Threads* por Solicitação? 77
Você Faz o Design tipos seguros para *threads* por default? 78
Você Usa Bloqueios Granulados? 78
Você Adquire-o Mais Tarde e Libera-o Mais Cedo? 78
Você Usa a Primitiva de Sincronização Apropriada? 78
Você Usa o Nível de Isolamento da Transação Apropriado? 79
O seu Design Considera a Execução Assíncrona? 79
Gerenciamento de Recursos **80**
O seu Design Acomoda *Pool*? 81
Você Adquire-o Mais Tarde e Libera-o Mais Cedo? 82
Armazenamento em Cache **82**
Você Armazena Dados em Cache? 83
Você Sabe Quais Dados Devem Ser Armazenados em Cache? 83
Você Armazena Dados Voláteis em Cache? 83
Você Escolheu o Local Correto para o Armazenamento em Cache? 84
Qual é Sua Política de Expiração? 85

Gerenciamento de Estados **86**
Você Usa Componentes Sem Estado? 87
Você Usa o .NET Remoting? 87
Você Usa WebServices? 87
Você Usa Enterprise Services? 88
Você se Certificou de que os Objetos a Serem Armazenados em Sessão São Serializáveis? 88
Você Depende do *View State*? 88
Você Sabe o Número de Sessões Simultâneas e os Dados Médios de Sessão por Usuário? 88
Você Usa as Estruturas de Dados Apropriadas? 89
Você Precisa de Coleções Customizadas? 90
Você Precisa Estender com *IEnumerable* as suas Coleções Customizadas? 90
Acesso a Dados **90**
Como Você Transmite os Dados entre as Camadas? 91
Você Usa *Stored Procedures*? 91
Você Processa Somente os Dados Necessários? 92
Você Precisa Paginar Através dos Dados? 92
Suas Transações Abrangem Vários Sistemas de Armazenamentos de Dados? 93
Você Manipula BLOBs? 93
Você Está Consolidando Códigos Repetidos para Acesso a Dados? 93
Manipulação de Exceções **94**
Você Usa Exceções para Controlar o Fluxo do Aplicativo? 94
Os Limites de Manipulação de Exceções estão Bem-Definidos? 95
Você Usa Códigos de Erro? 95
Você Captura Exceções Somente Quando Necessário? 96
Considerações sobre Design das Classes **96**
A sua Classe Possui os Dados Sobre os Quais Ela Age? 96
As suas Classes Expõem Interfaces? 96
As suas Classes Contêm Métodos Virtuais? 96
As suas Classes Contêm Métodos que Recebem Parâmetros Variáveis? 97
Resumo **97**
Recursos Adicionais **97**

Parte

II

Adequando o Design para Obter Desempenho

Nesta Parte:

- **Capítulo 2** - Modelagem do Desempenho
- **Capítulo 3** - Diretrizes de Design para Desempenho de Aplicações
- **Capítulo 4** - Revisão da Arquitetura e Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade

Capítulo

2

Modelagem do Desempenho

Objetivos

- Projetar para obter desempenho de antemão.
- Gerenciar riscos quanto ao desempenho.
- Mapear exigências comerciais para objetivos de desempenho.
- Equilibrar o desempenho em relação a outras exigências de qualidade do serviço.
- Identificar e analisar os principais cenários de desempenho.
- Identificar e alocar orçamentos.
- Avaliar o modelo para garantir que os objetivos de desempenho sejam atendidos.
- Identificar métricas e casos de teste.

Como Usar Este Capítulo

Visão Geral

Este capítulo apresenta a modelagem do desempenho com base nos métodos usados pelas equipes da Microsoft. Métodos semelhantes são recomendados em outros lugares, como no livro *Performance Solutions* de Connie U. Smith e Lloyd G. Williams. Você e a sua organização terão que adaptar o processo ao seu ambiente.

A modelagem do desempenho é um método estruturado e repetitivo usado para modelar o desempenho do software. Ela começa durante as primeiras fases de design do aplicativo e continua durante toda a sua vida útil.

O desempenho costuma ser ignorado até tornar-se um problema. Existem vários problemas com este método reativo:

- Frequentemente, os problemas de desempenho são introduzidos no início da fase de design.
- Os problemas de design nem sempre são corrigidos por meio de ajuste ou de codificação mais eficiente.

- Corrigir problemas relacionados à arquitetura ou ao design em um estágio mais avançado do ciclo de vida do projeto nem sempre é possível. Na melhor das hipóteses, isto é ineficiente e, normalmente, muito trabalhoso.

Ao criar modelos de desempenho, você identifica os cenários do aplicativo e seus objetivos de desempenho. Os objetivos de desempenho são seus critérios mensuráveis, como tempo de resposta, taxa de transferência (quanto trabalho e em quanto tempo) e utilização de recursos (CPU, memória, E/S do disco e E/S da rede). Você divide os cenários de desempenho em etapas e atribui orçamentos a eles. O orçamento define os recursos e as restrições nos objetivos de desempenho.

A modelagem do desempenho oferece vários benefícios importantes:

- O desempenho torna-se parte do seu design.
- A modelagem ajuda a responder à pergunta: “O design oferece suporte aos seus objetivos de desempenho?”. Ao criar e analisar modelos, você pode avaliar as compensações antes de realmente desenvolver a solução.
- Você sabe explicitamente quais decisões de design recebem influência do desempenho e as restrições por ele impostas nas futuras decisões relacionadas ao design. Frequentemente, essas decisões não são capturadas e podem causar esforços de manutenção contrários às metas originais.
- Você evita surpresas em relação ao desempenho quando seus aplicativos são colocados em produção.
- Como resultado, você tem um documento com cenários organizados em itens, o qual o ajuda a ver rapidamente o que é importante. Isso se traduz no onde instrumentalizar, no que testar e no como saber se você está no caminho certo para alcançar as metas de desempenho.

A modelagem antecipada do desempenho não substitui o teste de carga baseado em cenários nem a criação de protótipos para validar o design. Na verdade, é preciso criar um protótipo e testar para determinar os custos e verificar se o seu plano faz sentido. Os dados dos protótipos podem ajudar a avaliar as decisões na fase inicial de design, antes de implementar um design que não permita atender às suas metas de desempenho.

O modelo de desempenho apresentado neste capítulo tem duas partes:

- **Uma estrutura de informações** para ajudar a capturar as informações relacionadas ao desempenho. Essas informações podem ser fornecidas de forma parcial, com suposições e requisitos, e podem se estender de forma mais abrangente, conforme as suas necessidades.
- **Um processo** que o ajuda a definir e a capturar de forma incremental as informações que auxiliam as equipes a trabalharem em sua solução e focadas no uso, captura e compartilhamento das informações apropriadas.

Para usar esse modelo de desempenho, proceda da seguinte forma:

- **Defina metas.** Capture as informações parciais relacionadas ao desempenho que você já tenha, incluindo as métricas do protótipo do aplicativo, cenários importantes, cargas de trabalho, objetivos ou orçamentos. O modelo de desempenho apresentado neste capítulo foi criado para você usar como entrada as informações parciais que você já tem. Você não precisa ter todos os dados nem ter uma compreensão completa de seus próprios requisitos e soluções.

- **Avalie.** Execute as tarefas sugeridas no processo para definir metas e avaliar o resultado da ação de forma interativa, usando o modelo parcialmente concluído como um guia central. Isso permite adicionar informações ao modelo e refiná-las. Os novos dados fornecerão informações para a próxima etapa de definição e avaliação da meta.

Por que modelar o desempenho?

Um modelo de desempenho oferece uma maneira de descobrir o que não se sabe. Os benefícios da modelagem do desempenho incluem:

- O desempenho torna-se uma parte do processo de desenvolvimento e não uma reflexão tardia.
- Você calcula as compensações *(tradeoffs)* no início do ciclo de vida com base nas avaliações.
- Os casos de teste mostram se você está ou não no caminho certo, rumo aos objetivos de desempenho por todo o ciclo de vida do aplicativo.

A modelagem permite avaliar o design antes de investir tempo e recursos para implementar um design com falhas. Ter as etapas do processamento dos cenários de desempenho estabelecidas permite compreender a natureza do trabalho do aplicativo. Ao conhecer a natureza desse trabalho e as restrições que o afetam, você pode tomar decisões mais fundamentadas.

O modelo pode revelar as seguintes informações sobre o aplicativo:

- Quais são os caminhos relevantes do código e como eles afetam o desempenho?
- Em que ponto o uso de recursos ou de computações afetam o desempenho?
- Quais são os caminhos de código executados com mais freqüência? Isso ajuda a identificar onde gastar o tempo com ajustes.
- Quais são as principais etapas que acessam recursos e levam à contenção?
- Onde está o seu código em relação aos recursos (local ou remoto)?
- Quais compensações você fez para obter desempenho?
- Quais componentes têm relação com quais outros componentes ou recursos?
- Onde estão as chamadas síncronas e assíncronas?
- Qual é o trabalho vinculado à E/S (Entrada/Saída) e qual é o trabalho vinculado à CPU?
- E o modelo pode revelar sobre as metas o seguinte:
- Qual é a prioridade das metas de desempenho e a possibilidade de alcançá-las?
- Em que ponto as metas de desempenho afetaram o design?

Gerenciamento de Riscos

O tempo, o trabalho e o dinheiro investidos antecipadamente na modelagem do desempenho devem ser proporcionais ao risco do projeto. Para um projeto com risco significativo, no qual o desempenho é crítico, você pode gastar mais tempo e energia antecipadamente no desenvolvimento do modelo. Para um projeto no qual o desempenho não é tão preocupante, o método de modelagem pode ser tão simples quanto escrever os cenários de desempenho em um quadro branco.

Orçamento

A modelagem do desempenho é essencialmente um exercício de elaboração de orçamento. O orçamento representa as restrições e permite especificar quanto você pode gastar (recurso por recurso) e como planejar os gastos. As restrições controlam o total de gastos e, assim, você pode decidir onde gastar até chegar ao total. Você atribui orçamento em relação a tempo de resposta, taxa de transferência, latência e utilização de recursos.

A modelagem do desempenho não precisa envolver um grande volume de trabalho antecipado. Na verdade, deve fazer parte do trabalho que você já faz. Para começar, você pode até mesmo usar um quadro branco para descrever rapidamente os principais cenários e dividí-los em etapas.

Se você conhecer as metas, poderá rapidamente avaliar se os cenários e as etapas estão dentro das fronteiras ou se precisa alterar o design para acomodar no orçamento. Se você não conhecer as metas (em particular, a utilização de recursos), você terá que definir as bases mestras. Em ambas as situações, logo você poderá começar a criar o protótipo e a medir para ter alguns dados com os quais trabalhar.

O Que Você Deve Saber

Os modelos de desempenho são criados na forma de documento usando a ferramenta de sua escolha (um simples documento do Word funciona bem). O documento torna-se um ponto de comunicação para os outros membros da equipe. O modelo de desempenho contém muitas informações importantes, incluindo metas, orçamentos (utilização de tempo e de recursos), cenários e cargas de trabalho. Use o modelo de desempenho para estabelecer as possibilidades e avaliar alternativas, antes de partir para um design ou de tomar uma decisão de implementação. Você precisa avaliar para saber o custo das suas ferramentas. Por exemplo, qual é o custo de uma determinada API (interface de programação de aplicativo)?

Práticas Recomendadas

Considere as práticas recomendadas a seguir ao criar modelos de desempenho:

- Determine os orçamentos de tempo de resposta e de utilização de recursos no seu design.
- Identifique o ambiente alvo da implantação.
- Não substitua o teste de carga com base no cenário pela modelagem do desempenho pelas seguintes razões:
  - A modelagem do desempenho sugere em quais áreas se deve trabalhar, mas não pode prever a melhoria resultante de uma alteração.
  - A modelagem do desempenho fornece informações sobre metas e avaliações úteis ao teste de carga baseado no cenário.
  - O desempenho modelado pode ignorar muitas condições de carga baseada no cenário que podem ter um impacto enorme sobre o desempenho geral.

Informações no Modelo de Desempenho

As informações no modelo de desempenho são divididas em áreas diferentes. Cada área concentra-se na captura de uma perspectiva. Cada área tem atributos importantes que ajudam a executar o processo. A Tabela 2.1 mostra as principais informações incluídas no modelo de desempenho.

Tabela 2.1: Informações no Modelo de Desempenho

Categoria	Descrição
Descrição do Aplicativo	O design do aplicativo em relação às suas camadas e à infra-estrutura alvo.
Cenários	Situações de uso críticas e significativas, diagramas de seqüência e histórias de usuários relevantes ao desempenho.
Objetivos do Desempenho	Tempo de resposta, taxa de transferência e utilização de recursos.
Orçamentos	Restrições definidas em relação à execução dos casos de uso, como tempo máximo de execução e níveis de utilização de recursos, incluindo CPU, memória, *E/S* de disco e *E/S* da rede.
Avaliações	Métricas reais de desempenho obtidas através da execução de testes relacionados a custos de recursos e a questões de desempenho.
Metas de Carga de Trabalho	Metas relacionadas ao número de usuários, usuários concorrentes, volume de dados e informações sobre o uso desejado do aplicativo.
Hardware Padrão	Descrição do hardware onde serão executados os testes, em relação à topologia da rede, largura de banda, CPU, memória, disco etc.

Outras informações que talvez sejam necessárias incluir são mostrados na Tabela 2.2.

Tabela 2.2: Informações Adicionais Que Podem Ser Necessárias

Categoria	Descrição
Requisitos de QoS (Qualidade de Serviço)	Os requisitos de QoS, como segurança, capacidade de manutenção e interoperabilidade, podem causar impacto no desempenho. É preciso entrar em acordo com as equipes de software e de infra-estrutura em relação às restrições e aos requisitos de QoS.
Requisitos de Carga de Trabalho	Número total de usuários, usuários concorrentes, volume de dados e informações sobre o uso esperado do aplicativo.

Entrada

É preciso obter alguns dados no processo de modelagem do desempenho. Esses dados incluem informações iniciais (talvez até mesmo experimentais) sobre:

- Documentação de cenários e de design sobre casos de uso críticas e significativas.
- Design do aplicativo e infra-estrutura visada, bem como qualquer restrição por ela imposta.
- Requisitos de QoS e restrições de infra-estrutura, incluindo SLAs (contratos de serviço).
- Requisitos de carga de trabalho oriundos de dados de marketing sobre clientes potenciais.

Saídas

O resultado da modelagem do desempenho é o seguinte:

- Um documento de modelo de desempenho.
- Casos de teste com metas.

Documento de Modelo de Desempenho

O documento de modelo de desempenho pode conter:

- Objetivos do desempenho.
- Orçamentos.
- Cargas de trabalho.
- Cenários divididos em itens com metas.
- Casos de teste com metas.

Um cenário dividido em itens é aquele dividido em etapas de processamento. Por exemplo, um cenário para um Pedido pode incluir autenticação, validação da entrada do pedido, validação das regras comerciais e os pedidos sendo confirmados no banco de dados. Os cenários divididos em itens incluem os orçamentos a ele atribuídos e os objetivos de desempenho para cada etapa.

Casos de Teste com Metas

Você usa os casos de teste para gerar métricas de desempenho. Elas validam seu aplicativo em relação aos objetivos de desempenho. Os casos de teste ajudam a determinar se os objetivos de desempenho serão ou não alcançados.

Processo

O modelo de processo da modelagem do desempenho está resumido na Figura 2.1.

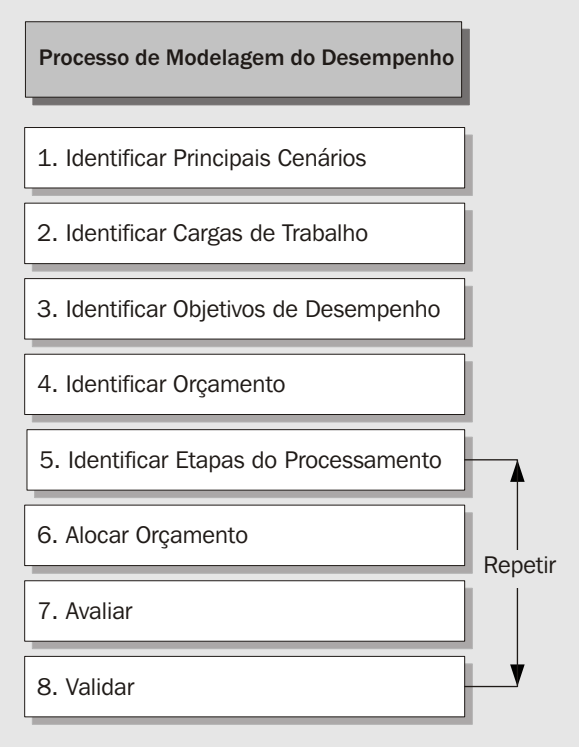

Figura 2.1
Modelo de desempenho de oito etapas

O processo de modelagem do desempenho envolve as seguintes etapas:

1. **Identificar principais cenários.** Identificar os cenários onde o desempenho é importante e aqueles que representam maior risco aos objetivos de desempenho.
2. **Identificar carga de trabalho.** Identificar quantos usuários e usuários concorrentes o sistema precisa acomodar.
3. **Identificar objetivos de desempenho.** Definir os objetivos de desempenho de cada um dos principais cenários. Os objetivos de desempenho refletem os requisitos comerciais.
4. **Identificar orçamento.** Identificar o orçamento ou as restrições. Isso inclui o tempo máximo de execução no qual uma operação deve ser concluída e as restrições de utilização de recursos, como CPU, memória, E/S do disco e E/S da rede.
5. **Identificar etapas do processamento.** Dividir os principais cenários em etapas do processamento.
6. **Alocar orçamento.** Dividir o orçamento (determinado na etapa 4) pelas etapas do processamento (determinadas na etapa 5) para atender aos objetivos de desempenho (definidos na etapa 3).
7. **Avaliar.** Avaliar o design em relação aos objetivos e ao orçamento. Talvez seja necessário modificar o design ou dividir o orçamento de tempo de resposta e de utilização de recursos de forma diferente para atender aos objetivos de desempenho.
8. **Validar.** Valide o modelo e as estimativas. Esta é uma atividade contínua e inclui criação de protótipo, avaliação e medição.

As próximas seções descrevem cada uma das etapas precedentes.

Etapa 1: Identificar Principais Cenários.

Identifique os cenários de seu aplicativo que são importantes do ponto de vista de desempenho. Se você tiver casos de uso ou histórias de usuários documentadas, use-as para ajudar a definir os seus cenários. Os principais cenários incluem:

- Cenários críticos.
- Cenários significativos.

Cenários Críticos

São os cenários que têm expectativas ou requisitos de desempenho específicos. Os exemplos incluem os cenários cobertos pelos SLAs *(Service Level Agreement)* ou aqueles que têm objetivos de desempenho específicos.

Cenários Significativos

Os cenários significativos não têm objetivos de desempenho específicos, como uma meta de tempo de resposta, mas podem causar impacto em outros cenários críticos.

Para ajudar a identificar os cenários significativos, identifique os cenários com as seguintes características:

- Cenários que executem de forma paralela ao cenário cujo desempenho é crítico.
- Cenários executados com freqüência.
- Cenários responsáveis por uma alta porcentagem de uso do sistema.
- Cenários que consumam uma quantidade significativa de recursos do sistema.

Não ignore os cenários significativos. Eles podem influenciar a capacidade dos cenários críticos de atender aos objetivos de desempenho. Além disso, não se esqueça de considerar o comportamento do sistema caso cenários diferentes, significativos ou críticos, sejam executados simultaneamente por usuários distintos. Essa "integração paralela" muitas vezes estimula as principais decisões sobre as unidades de trabalho do aplicativo. Por exemplo, para manter a rapidez de resposta das pesquisas, talvez seja necessário confirmar (dar *commit*) nos pedidos, um item de linha por vez.

Etapa 2: Identificar Carga de Trabalho.

Normalmente, a carga de trabalho é oriunda dos dados de marketing. Ela inclui:

- Total de usuários.
- Usuários ativos simultaneamente.
- Volume de dados.
- Volume e combinação de transações.

Para obter a modelagem do desempenho, você precisa identificar como essa carga de trabalho se aplica a um cenário específico. Os exemplos de requisitos são:

- Talvez seja necessário oferecer suporte a 100 usuários navegando concorrentemente.
- Talvez seja necessário oferecer suporte a 10 usuários colocando pedidos concorrentemente.

Observação: usuários concorrentes são aqueles que acessam um site exatamente no mesmo momento. Também são aqueles que têm conexões ativas num mesmo site.

Etapa 3: Identificar Objetivos de Desempenho.

Para cada cenário identificado na Etapa 1, anote os objetivos de desempenho. Os objetivos de desempenho são determinados pelos requisitos comerciais.

Normalmente, eles incluem:

- **Tempo de resposta.** Por exemplo, o catálogo de produtos deve ser exibido em menos de três segundos.
- **Taxa de transferência.** Por exemplo, o sistema deve oferecer suporte a 100 transações por segundo.
- **Utilização de recursos.** Um aspecto quase sempre negligenciado é a quantidade de recursos que o aplicativo está consumindo, em relação a CPU, memória, E/S do disco e E/S da rede.

Ao estabelecer os objetivos de desempenho, considere o seguinte:

- Requisitos de carga de trabalho.
- SLAs.
- Tempos de resposta.
- Expansão prevista.
- Vida útil do aplicativo.

Para a expansão prevista, é preciso considerar se o design atenderá às necessidades em seis meses ou daqui a um ano. Se o aplicativo tem uma vida útil de apenas seis meses, você está preparado para negociar alguma capacidade de extensão em relação ao desempenho? Se o aplicativo tiver uma provável vida útil longa, que desempenho você pretende negociar em relação à capacidade de manutenção?

Etapa 4: Identificar Orçamento.

Os orçamentos são as restrições. Por exemplo, qual é a maior quantidade aceitável de tempo necessário para concluir uma operação, a partir de que fronteira o aplicativo não consegue atender aos objetivos de desempenho.

Normalmente, o orçamento leva em consideração o seguinte:

- **Tempo de execução.**
- **Utilização de recursos.**

Tempo de Execução

As restrições de tempo de execução determinam a quantidade máxima de tempo necessário para executar determinadas operações.

Utilização de Recursos

Os requisitos de utilização de recursos definem os limites para o uso dos recursos disponíveis. Por exemplo, você pode ter um limite máximo de 75% na utilização do processador, e o consumo de memória não deve exceder 50 MB.

Os recursos comuns a considerar incluem:

- CPU.
- Memória.
- E/S da rede.
- E/S do disco.

Mais Informações

Para obter mais informações, consulte “Recursos do Sistema”, no capítulo 15: “Medindo o Desempenho do Aplicativo .NET”.

Considerações Adicionais

O tempo de execução e a utilização de recursos são úteis no contexto dos objetivos de desempenho. No entanto, o orçamento tem várias outras dimensões às quais talvez você esteja sujeito. Outras considerações relacionadas ao orçamento podem incluir:

- **Rede.** As considerações relacionadas à rede incluem largura de banda.
- **Hardware.** As considerações relacionadas ao hardware incluem itens como servidores, memória e CPUs.
- **Dependência de recursos.** As considerações relacionadas à dependência de recursos incluem itens como o número de conexões disponíveis ao banco de dados e conexões de WebServices.
- **Recursos compartilhados.** As considerações relacionadas a recurso compartilhado incluem itens como quantidade de largura de banda disponível, quantidade da CPU em caso de compartilhamento de um servidor com outros aplicativos e quantidade de memória disponível.
- **Recursos do projeto.** Do ponto de vista de um projeto, o orçamento do tempo e custo também são restrições.

Etapa 5: Identificar Etapas do Processamento.

Distribua os cenários em itens e divida-os em diferentes etapas do processamento, como as mostradas na Tabela 2.3. Se você estiver familiarizado com UML, os casos de uso e os diagramas de seqüência podem ser usados como entradas. Da mesma forma, as histórias dos usuários *(user stories)* do *Extreme Programming* podem fornecer dados úteis para essa etapa.

Tabela 2.3:

Etapas do Processamento
1. Um pedido é enviado pelo cliente.
2. O *token* de autenticação do cliente é validado.
3. A entrada do pedido é validada.
4. As regras comerciais validam o pedido.
5. O pedido é enviado a um servidor de banco de dados.
6. O pedido é processado.
7. Uma resposta é enviada ao cliente.

Um benefício adicional da identificação das etapas do processamento é que elas ajudam a identificar aqueles pontos do aplicativo nos quais você deve considerar a adição de instrumentação personalizada. A instrumentação ajuda a fornecer os custos reais e o tempo necessário quando você começa a testar o aplicativo.

Etapa 6: Alocar Orçamento.

Divida o seu orçamento (determinado na Etapa 4, “Identificar Orçamento”) entre as etapas do processamento (determinadas na Etapa 5, “Identificar Etapas do Processamento”) para atender aos objetivos de desempenho. É preciso considerar o tempo de execução e a utilização dos recursos. Alguns dos itens do orçamento podem se aplicar a apenas uma etapa do processamento. Parte do orçamento pode se aplicar ao cenário, e parte deste a vários cenários.

Atribuindo Tempo de Execução às Etapas

Ao atribuir tempo às etapas do processamento, se você não souber quanto tempo atribuir, basta dividir o tempo entre as etapas de forma igualitária. Nesse ponto, não é importante que os valores sejam precisos, pois o orçamento será reavaliado após a medição do tempo real, mas é importante ter uma idéia dos valores. Não se preocupe em atingir a perfeição, mas procure alcançar um nível de segurança razoável e você estará num bom caminho.

Você não quer ficar estagnado, mas, ao mesmo tempo, não quer esperar o aplicativo ser desenvolvido e instrumentado para obter os números reais. Onde não souber os tempos de execução, tente dividir o tempo de forma igualitária, veja onde pode haver problemas ou onde há tensão.

Se a divisão do orçamento mostrar que cada etapa do processamento tem tempo suficiente, não há necessidade de ampliar o exame. No entanto, para aquelas que parecerem arriscadas, faça alguns testes (por exemplo, com protótipos) para verificar se o que você precisa fazer é possível e, em seguida, prossiga.

Observe que uma ou mais etapas podem ter um tempo fixo. Por exemplo, você pode fazer uma chamada de banco de dados que você sabe que não será concluída em menos que três segundos. Outros tempos são variáveis. Os custos fixos e variáveis devem ser menores ou iguais ao orçamento alocado para o cenário.

Atribuindo Requisitos de Utilização de Recursos

Ao atribuir recursos às etapas do processamento, considere:

- Conheça o custo dos materiais. Por exemplo, qual é o custo da tecnologia x em comparação com a tecnologia y.
- Conheça o orçamento alocado para o hardware. Isso define o total de recursos disponíveis.
- Conheça os sistemas de hardware já estabelecidos.

A funcionalidade do aplicativo. Por exemplo, o processamento de documentos XML pesados pode exigir mais CPU, o acesso freqüente ao banco de dados ou a comunicação com o serviço de Web podem exigir mais largura de banda da rede, ou a carga de arquivos grandes pode exigir mais E/S do disco.

Etapa 7: Avaliar.

Avalie a possibilidade e a eficiência do orçamento antes de aplicar tempo e esforço nas etapas de criação de protótipo e de teste. Reveja os objetivos de desempenho e considere as seguintes perguntas:

- O orçamento atende aos objetivos?
- O orçamento é realista? É durante a primeira avaliação que você identifica novos testes que devem ser feitos para se obter números mais precisos para o orçamento.
- O modelo identifica um ponto de acesso ao recurso?
- Há alternativas mais eficientes?
- O design ou os recursos podem ser reduzidos ou modificados para atender aos objetivos?
- É possível melhorar a eficiência em relação ao consumo de recursos ou ao tempo?
- Um padrão, um design ou uma topologia de implantação alternativa fornece uma melhor solução?
- Quais são seus contrapesos? Você está negociando produtividade, escalabilidade, manutenabilidade ou segurança (contra o desempenho)?
- Considere as seguintes ações:
  - Modifique o seu design.
  - Reavalie os requisitos.
  - Altere a forma de alocar orçamento.

Etapa 8: Validar

Valide o modelo e as estimativas. Continue a criar protótipos e a avaliar o desempenho dos casos de uso por meio da captura das métricas. É uma atividade contínua e inclui criação de protótipo e medição. Continue a executar verificações de validação até as metas de desempenho a serem atendidas.

Quanto mais adiante você estiver no ciclo de vida do projeto, maior será a precisão da validação. No início do processo, a validação tem como base os *benchmarks* e os códigos dos protótipos disponíveis ou apenas os códigos de prova de conceito. Mais adiante, você poderá avaliar o código real à medida que o aplicativo se desenvolve.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre validação do desempenho do código Microsoft .NET, consulte “Código Gerenciado e Desempenho CLR” no Capítulo 13, “Revisão de Código: Desempenho do Aplicativo .NET”.
- Para obter mais informações sobre validação de código de protótipos e de produção, consulte “Como a Avaliação se Aplica ao Ciclo de Vida”, no Capítulo 15, “Medindo o Desempenho do Aplicativo .NET”.
- Para obter mais informações sobre o processo de validação, consulte “Processo de Otimização do Desempenho”, no Capítulo 17, “Otimizando o Desempenho do Aplicativo .NET”.

Resumo

Iniciar o quanto antes a modelagem do desempenho ajuda a expor os principais problemas e permite ver rapidamente onde é necessário fazer compensações no design, ou ajuda a identificar para onde direcionar os esforços. Uma etapa prática na direção certa é simplesmente capturar os principais cenários e dividí-los em operações ou etapas lógicas. Mas, o mais importante é identificar as metas de desempenho, como tempo de resposta, taxa de transferência e utilização de recursos, em cada cenário.

Conhecer os orçamentos relativos ao uso de CPU, memória, E/S do disco e E/S da rede que o aplicativo pode consumir. Estar preparado para fazer compensações ainda no momento do design - como usar uma tecnologia alternativa ou um mecanismo remoto de comunicação.

Ao adotar um método pró-ativo para o gerenciamento do desempenho e um processo de modelagem do desempenho, você obtém os seguintes resultados:

O desempenho torna-se um recurso do processo de desenvolvimento e não uma reflexão tardia.

Você calcula as compensações (*tradeoffs*) no início do ciclo de vida com base nas avaliações.

Os casos de teste mostram se você está ou não no caminho certo, rumo aos objetivo de desempenho, por todo o ciclo de vida do aplicativo.

Recursos Adicionais

Para obter mais informações e a literatura relacionada, consulte os seguintes recursos:

- Para obter informações relacionadas sobre engenharia de desempenho, consulte *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* de Connie U. Smith e Lloyd Williams.
- Para obter mais informações sobre engenharia de desempenho de software, consulte o site *“Software Performance Engineering Papers” em http://www.perfeng.com/paperndx.htm.*
- Para obter uma introdução ao uso de análise da situação de negócio para justificar o investimento na engenharia de desempenho de software, consulte "*Making the Business Case for Software Performance Engineering*" de Lloyd G. Williams, Ph.D. e Connie U. Smith, Ph.D., em *http://www.perfeng.com/papers/buscase.pdf.*
- Para obter informações sobre como avaliar se a arquitetura do software atenderá aos objetivos de desempenho, consulte "*PASA: An Architectural Approach to Fixing Software Performance Problems*" de Lloyd G. Williams e Connie U. Smith, em *http://www.perfeng.com/papers/pasafix.pdf.*

- Para obter exemplos de como dividir as funções comerciais e de funcionalidade do aplicativo para análise do desempenho, consulte "*How do you eat an elephant? or How to digest application performance in bite-size chunks*" em *http://www.whitespacesolutions.com/whitepapers/How_do_you_eat_an_elephant.pdf.*
- Para obter os conceitos e as idéias por tras da metodologia de modelagem do desempenho, consulte "*Performance Modeling Methodology*" em *http://www.hyperformix.com/FileLib /PerfModMeth.pdf.*
- Para obter uma solução alternativa para divisão de um aplicativo da Web para fins de análise, consulte "*Stepwise Refinement: A Pragmatic Approach for Modeling Web Applications*" em *http://www.whitespacesolutions.com/whitepapers/HyPerformix_Stepwise_Refinement.pdf*
- Para obter informações sobre como incorporar a modelagem do desempenho ao ciclo de vida do sistema, consulte "*eBusiness Performance: Risk Mitigation in Zero Time (Do It Right the First Time)*" em *http://www.whitespacesolutions.com/whitepapers/HyPerformix-Risk.Mitigation.in.Zero.pd* e "*Performance Engineering throughout the System Life Cycle*" em *http://www.whitespacesolutions.com/whitepapers/PE_LifeCycle.pdf*
- Para obter informações sobre caracterização de carga de trabalho, consulte "*Falling from a Log Techniques for Workload Characterisation*" em *http://www.whitespacesolutions.com/whitepapers/Falling_from_a_log.pdf*
- Para obter uma solução alternativa para a aplicação da modelagem do desempenho ao ciclo de vida do software, consulte "*Wells Fargo Performance Modeling Techniques for Integrating into Development Life-Cycle Processes*" em *http://www.cmg.org/conference/refs99/papers /99p2119.doc.*
- Para obter mais informações sobre engenharia de desempenho, consulte "*An Enterprise Level Approach to Proactive Performance Engineering*" em *http://www.whitespacesolutions.com/whitepapers/Cook--EnterprisePPE.pdf.*

Capítulo

3

Diretrizes de Design para o Desempenho de Aplicações

Objetivos

- Conhecer as compensações *(tradeoffs)* no design em relação ao desempenho e escalabilidade.
- Aplicar ao design um método baseado em princípios. Identificar e usar a estrutura de desempenho e de escalabilidade.
- Conhecer as considerações de design em relação à escalabilidade horizontal e vertical.
- Reduzir o excesso de comunicação e de transformação de dados.
- Melhorar a capacidade de concorrência do aplicativo.
- Gerenciar recursos de forma eficiente.
- Armazenar dados do aplicativo em cache de forma eficiente.
- Gerenciar o estado do aplicativo de forma eficiente.
- Criar uma camada de apresentação eficiente.
- Criar uma camada de negócio eficiente.
- Criar uma camada de acesso a dados eficiente.

Visão Geral

Desempenho e escalabilidade são duas considerações de QoS. Outros atributos de QoS incluem disponibilidade, capacidade de gerenciamento, integridade e segurança. Esses atributos devem ser equilibrados com desempenho e escalabilidade e, muitas vezes, isso envolve compensações na arquitetura e no design.

Durante a fase de design, identifique os objetivos de desempenho. Qual é a velocidade suficiente? Quais são as restrições de tempo de resposta e de taxa de transferência do aplicativo? Qual é o consumo aceitável de CPU, memória, E/S do disco e E/S da rede pelo aplicativo? Estes são fatores chaves que o seu design deve estar apto a acomodar.

As diretrizes incluídas neste capítulo o ajudarão a desenvolver aplicativos que atendam aos objetivos de desempenho e de escalabilidade. O capítulo começa com um conjunto de princípios comprovados

para o processo do design. Ele também aborda as questões de implantação que devem ser consideradas no design. As seções subseqüentes apresentam diretrizes de design organizadas pela estrutura de desempenho e de escalabilidade introduzida no Capítulo 1, “Princípios Básicos de Engenharia de Desempenho”. Finalmente, será apresentado um conjunto de recomendações centralizado no cliente, nas camadas de apresentação, de negócio e de acesso a dados.

Como Usar Este Capítulo

Use este capítulo como fonte de ajuda para o design de aplicativos e a avaliação das decisões de design. Você pode aplicar as diretrizes de design contidas neste capítulo a aplicativos novos e já existentes. Para aproveitar ao máximo este capítulo:

- **Vá direto para os tópicos ou leia do início ao fim.** Os títulos principais deste capítulo ajudam a localizar os tópicos de seu interesse. Como alternativa você pode ler o capítulo do início ao fim para obter uma apreciação completa das questões de desempenho e de escalabilidade do design.
- **Use a seção "Arquitetura" de cada capítulo técnico da Parte III deste guia.** Consulte as seções de arquitetura do capítulo técnico para criar um design melhor e fazer melhores escolhas no momento da implementação.
- **Use a seção "Considerações de Design" de cada capítulo técnico da Parte III deste guia.** Consulte as seções de considerações de design do capítulo técnico para obter diretrizes específicas de design relacionadas à tecnologia.
- **Use a seção “Listas de Verificação” deste guia.** Use a "Lista de Verificação: Análise da Arquitetura e do Design em Relação ao Desempenho e à Escalabilidade" para ver e avaliar rapidamente as diretrizes apresentadas neste capítulo.

Princípios

As orientações espalhadas por todo este capítulo são baseadas em princípios. Desempenho, como segurança e muitos outros aspectos da engenharia de software, adaptam-se a um método baseado em princípios, no qual os princípios comprovados são aplicados independentemente da tecnologia da implementação ou do cenário do aplicativo.

Princípios do Processo de Design

Considere os seguintes princípios ao aprimorar o processo de design:

- **Defina metas objetivas.** Evite metas ambíguas ou incompletas que não possam ser avaliadas, como “o aplicativo deve ter uma execução rápida” ou “o aplicativo deve carregar rapidamente”. Você precisa saber as metas de desempenho e de escalabilidade do aplicativo para poder (a) criar um design que as atenda e (b) planejar os testes com base nelas. As suas metas devem ser mensuráveis e verificáveis.

  Os requisitos a considerar em relação aos objetivos de desempenho incluem tempos de resposta, taxa de transferência utilização de recursos e carga de trabalho. Por exemplo, quanto tempo uma solicitação específica deve demorar? O aplicativo precisa oferecer suporte para quantos usuários? Qual é a carga máxima que o aplicativo deve processar? Ele deve oferecer suporte para quantas transações por segundo?

  É preciso considerar também as fronteiras de utilização de recursos. Qual é o consumo aceitável de CPU, memória, E/S do disco e E/S da rede pelo aplicativo?

- **Valide a arquitetura e o design no início do processo.** Identifique, crie o protótipo e valide as principais opções de design antecipadamente. Ao começar tendo como objetivo o resultado final, sua meta será avaliar se a arquitetura do aplicativo pode oferecer suporte às metas de desempenho. Algumas decisões importantes a serem validadas antecipadamente incluem topologia da implantação, balanceamento de carga, largura de banda da rede, estratégias de autenticação e de autorização, gerenciamento de exceções, instrumentação, design do banco de dados, estratégias de acesso a dados, gerenciamento de estados e armazenamento em cache. Esteja preparado para reduzir recursos e funcionalidade ou retrabalhar áreas que não atendam às metas de desempenho. Saiba os custos das opções e dos recursos específicos do design.
- **Elimine o que for desnecessário.** Muitas vezes, as maiores recompensas vêm com a descoberta de que seções inteiras de código podem ser removidas por serem desnecessárias. Frequentemente isso ocorre quando funções (bem-ajustadas) são criadas para executar alguma operação com melhor desempenho. Frequentemente isto faz com que códigos provisórios relativos à primeira versão desta função do sistema não sejam mais utilizadas. A eliminação desses caminhos "desnecessários" pode, em geral, causar uma melhoria global incrível.
- **Ajuste o desempenho de ponta a ponta.** Otimizar um único recurso poderia reduzir a capacidade de outro recurso e retardar o desempenho geral. Do mesmo modo, um único gargalo em um subsistema do aplicativo pode afetar o seu desempenho geral, independentemente da qualidade do ajuste dos outros subsistemas. Você obtém um benefício maior do teste de desempenho quando ajusta de ponta a ponta, em vez de gastar tempo e dinheiro consideráveis ajustando um subsistema específico. Identifique os gargalos e, em seguida, ajuste as partes específicas do aplicativo. Normalmente, o trabalho no desempenho move-se de um gargalo para outro.
- **Avalie todo o ciclo de vida.** É preciso saber se o desempenho do aplicativo atenderá ou não aos seus objetivos de desempenho. O ajuste do desempenho é um processo iterativo de melhoria contínua com a expectativa de ganhos fixos, caracterizado por perdas não planejadas, até atender aos seus objetivos. Avalie o desempenho do aplicativo em relação aos seus objetivos de desempenho por todo o ciclo de vida do desenvolvimento e garanta que o desempenho seja o principal componente desse ciclo de vida. Faça testes unitários do desempenho de partes específicas do código e verifique se o código atende aos objetivos definidos de desempenho antes de partir para o teste integrado do desempenho.
- **Quando o aplicativo estiver em produção,** continue a avaliar o seu desempenho. Fatores como número de usuários, padrões de uso e volume de dados mudam com o tempo. Novos aplicativos podem começar a disputar os recursos compartilhados.

Princípios de Design

Os princípios de design a seguir foram extraídos de arquiteturas que demonstraram desempenho e escalabilidade excelentes com o passar do tempo:

- **Crie serviços pouco granulados.** Serviços pouco granulados reduzem o número de interações entre cliente e serviço e ajudam a criar unidades de trabalho coesas. Eles também ajudam a remover partes internas do serviço do cliente e a fornecer um acoplamento fraco entre cliente e serviço. O acoplamento fraco aumenta a capacidade de encapsular as alterações. Se você já tem serviços específicos, considere a possibilidade de envolvê-los com uma camada de fachada *(façade)* para ajudar a alcançar os benefícios de um serviço pouco granulado.

- **Minimize o vaivém de solicitações e respostas *(round trip)* organizando o trabalho em lotes.** Minimize o vaivém de solicitações e respostas para diminuir a latência das chamadas. Por exemplo, reúna as chamadas em lote e crie serviços pouco granulados que permitam executar uma única operação lógica usando uma única ida e volta para a solicitação e resposta. Aplique esse princípio para reduzir a comunicação entre fronteiras, como *threads*, processos, processadores ou servidores. Esse princípio é especificamente importante ao fazer chamadas para o servidor remoto em uma rede.
- **Demore a adquirir e libere logo.** Reduza o tempo que você mantém os recursos compartilhados e limitados, como conexões de rede e de banco de dados. Liberar e adquirir novamente esses recursos do sistema operacional pode ser trabalhoso, portanto, considere um plano de reciclagem para apoiar o princípio "demore a adquirir e libere logo". Dessa forma, você poderá otimizar o uso de recursos compartilhados nas solicitações.
- **Avalie a afinidade com os recursos de processamento.** Quando determinados recursos estão disponíveis apenas em servidores ou processadores específicos, fica estabelecida uma afinidade entre o recurso e o servidor ou processador. Embora a afinidade possa por vezes melhorar o desempenho, ela também pode causar impacto na escalabilidade. Considere cautelosamente as suas necessidades de escalabilidade. Você terá que adicionar mais processadores ou servidores? Se as solicitações ao aplicativo estiverem limitadas devido à afinidade com um processador ou servidor específico, você poderá estar inibindo a capacidade de escalabilidade do aplicativo. À medida que a carga no aplicativo aumenta, a capacidade de distribuir o processamento pelos processadores, ou servidores, influencia a capacidade potencial do seu aplicativo.
- **Coloque o processamento mais perto dos recursos necessários.** Se o processamento envolve muitas interações entre cliente e serviço, talvez seja necessário colocar o processamento mais perto do cliente. Se o processamento interage de forma intensa com o armazenamento de dados, talvez seja conveniente colocar o processamento mais perto dos dados.
- **Coloque em *Pool* os recursos compartilhados.** Coloque em *pool* os recursos compartilhados que sejam escassos ou trabalhosos para criar, por exemplo, conexões de banco de dados ou de rede. Use um *pool* para ajudar a eliminar a sobrecarga no desempenho associada ao estabelecimento de acesso aos recursos e para melhorar a escalabilidade compartilhando um número limitado de recursos entre um número muito maior de clientes.
- **Evite trabalho desnecessário.** Use técnicas, como o armazenamento em cache, evitando o vaivém de solicitações e respostas, e a validação de entradas de dados no início do processo, para reduzir o processamento desnecessário. Para obter mais informações, consulte acima "Elimine o que for desnecessário".
- **Reduza a contenção.** Bloqueio e pontos de forte acesso são fontes comuns de contenção. O bloqueio é causado por tarefas de longa duração, como operações custosas de E/S. Os pontos de forte acesso resultam do acesso concentrado a determinados dados que todos precisam. Evite o bloqueio ao acessar recursos porque a contenção do recurso leva ao enfileiramento das solicitações. A contenção pode ser rara. Considere o cenário do banco de dados. Por um lado, tabelas grandes devem ser indexadas com cuidado para evitar o bloqueio causado pela E/S intensiva. No entanto, muitos clientes conseguirão acessar partes diferentes da tabela sem dificuldades. Por outro lado, tabelas pequenas têm pouca probabilidade de apresentar problemas de E/S, mas podem ser usadas com tanta freqüência por tantos clientes que serão disputadas de forma acirrada.

  As técnicas de redução da contenção incluem o uso eficiente de *threads* compartilhadas e a redução do tempo de retenção de bloqueios pelo código.

- **Use processamento progressivo.** Use práticas eficientes para lidar com alterações nos dados. Por exemplo, faça atualizações incrementais. Quando uma parte dos dados mudar, processe a parte alterada, não todos os dados. Além disso, considere fazer o processamento da saída de forma progressiva. Não bloqueie todo o conjunto de resultados quando puder fornecer ao usuário uma parte inicial e alguma interatividade no começo.
- **Processe tarefas independentes de forma simultânea.** Quando precisar processar várias tarefas independentes, você poderá executá-las de forma assíncrona para realizá-las simultaneamente. O processamento assíncrono oferece muitos benefícios às tarefas vinculadas a E/S, mas tem benefícios limitados quando as tarefas são CPU intensivas e restritas a um único processador. Se você planeja implantar sua aplicação em servidores que possuem apenas uma CPU, o uso de *threads* adicionais garante a troca de contexto, mas, como não há multisegmentação real, provavelmente haverá ganhos limitados apenas. *Threads* sob multiprocessamento em uma única CPU são executadas de forma relativamente lenta devido ao excesso de troca de contextos.

Considerações de Implantação

As considerações de tempo de execução reúnem funcionalidade do aplicativo, opções da arquitetura de implantação, requisitos operacionais e atributos de QoS. Esses aspectos são mostrados na Figura 3.1.

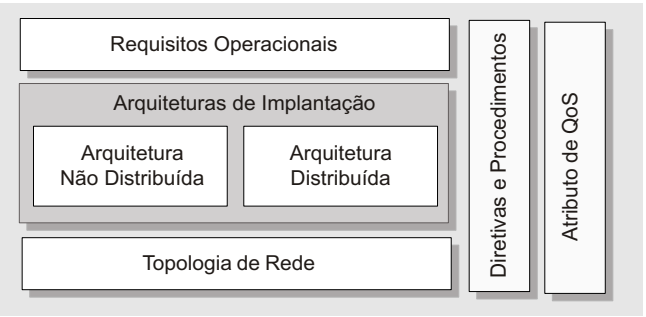

Figure 3.1
Considerações de Implantação

Durante a fase de design do aplicativo, analise as políticas e os procedimentos corporativos juntamente com a infra-estrutura na qual o aplicativo será implantado. Normalmente, o ambiente visado é rígido e o design do aplicativo deve refletir as restrições impostas. Também é preciso considerar outros atributos de QoS, como segurança e capacidade de manutenção. Às vezes, compensações no design serão necessárias, como por exemplo, por causa das restrições de protocolo ou das topologias de rede.

As principais questões relacionadas à implantação a serem reconhecidas no momento do design são as seguintes:

- **Considere a arquitetura da implantação.**
- **Identifique restrições e suposições no início do processo.**
- **Avalie a afinidade do servidor.**
- **Use um design em camadas.**
- **Mantenha-se no mesmo processo.**
- **Não mantenha a lógica do aplicativo em um local remoto, a menos que seja necessário.**

Considere a Arquitetura da Implantação

As arquiteturas distribuídas e não distribuídas são compatíveis com os aplicativos .NET. Ambos os métodos têm prós e contras diferentes em relação a desempenho, escalabilidade, facilidade de desenvolvimento, administração e operações.

Arquitetura Não Distribuída

Na arquitetura não distribuída, os códigos de apresentação, de negócio e de acesso a dados são separados logicamente, mas se encontram fisicamente em um único servidor Web. Isso é mostrado na Figura 3.2.

Figura 3.2
Arquitetura não distribuída do aplicativo: camadas lógicas em uma única camada física

Prós

- A arquitetura não distribuída é menos complexa que a distribuída.
- A arquitetura não distribuída apresenta vantagens no desempenho obtidas através de chamadas locais.

Contras

- Com a arquitetura não distribuída, é difícil compartilhar lógica de negócio com outros aplicativos.
- Com a arquitetura não distribuída, os recursos do servidor são compartilhados entre as camadas. Isso pode ser bom ou ruim. As camadas podem funcionar eficientemente juntas e o resultado será o uso otimizado, pois uma delas está sempre ocupada. No entanto, se uma camada exigir uma quantidade desproporcionalmente maior de recursos, estes recursos serão retirados de outra camada.

Arquitetura Distribuída

Com a arquitetura distribuída, a lógica de apresentação comunica-se remotamente com a lógica de negócio localizada em um servidor de aplicativos intermediário, como mostra a Figura 3.3.

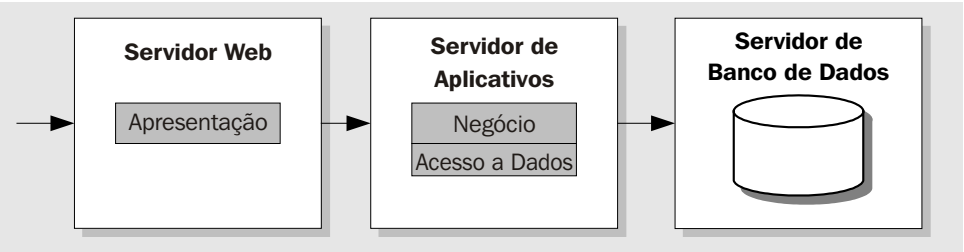

Figura 3.3
Arquitetura distribuída: camadas lógicas em várias camadas físicas

Prós

- A arquitetura distribuída tem a capacidade de fornecer escalabilidade horizontal e de fazer o balanceamento de carga da lógica de negócio de forma independente.
- A arquitetura distribuída tem recursos separados do servidor que estão disponíveis para camadas diferentes.
- A arquitetura distribuída é flexível.

Contras

- A arquitetura distribuída tem serialização adicional e excesso de latência da rede devido às chamadas remotas.
- A arquitetura distribuída é potencialmente mais complexa e mais cara em relação ao custo total de propriedade.

Identifique Restrições e Suposições no Início do Processo

Identifique qualquer restrição e suposição no início da fase de design para evitar surpresas posteriormente. Envolva os membros das equipes de rede e de infra-estrutura nesse processo. Analise qualquer diagrama de rede disponível, além da política de segurança e dos requisitos de operação.

Os ambientes visados normalmente são rígidos e o design do aplicativo precisa se ajustar às restrições impostas. Às vezes, são necessárias compensações no design por causa das considerações, como restrições de protocolo, *firewalls* e topologias específicas de implantação. Do mesmo modo, o design pode considerar suposições, como a quantidade de memória ou a capacidade da CPU, ou pode até mesmo desconsiderá-las. Considere a facilidade de manutenção. O requisito de facilidade de manutenção após a implantação normalmente afeta o design do aplicativo.

Avalie a Afinidade com o Servidor

A afinidade pode ter um impacto positivo ou negativo no desempenho e na escalabilidade. A afinidade com o servidor ocorre quando todas as solicitações de um cliente específico devem ser processadas num mesmo servidor. Frequentemente ela é introduzida quando no uso de caches locais atualizáveis ou no uso de estados da sessão que acontecem em processos específicos ou no processo local. Implementar posteriormente a escalabilidade vertical, no caso de um design que causa a afinidade com o servidor, acaba forçando o reprojeto ou o desenvolvimento de soluções complexas de sincronização para garantir o sincronismo dos dados nos vários servidores. Se você precisar ter escalabilidade horizontal, considere a afinidade com recursos que possam limitar a sua capacidade. Se você não precisar oferecer suporte à escalabilidade vertical, considere os benefícios de desempenho que a afinidade com um recurso pode trazer.

Use um Design em Camadas

Um design em camadas é aquele que divide a aplicação em lógica de apresentação, de negócio e de acesso a dados. Um bom design em camadas exibe seu alto nível de coesão mantendo os componentes que interagem entre si com freqüência em uma única camada, próximos uns dos outros. Um modelo de divisão em várias camadas com a separação da lógica de apresentação, de negócio e de acesso aos dados ajuda a desenvolver um aplicativo com uma capacidade maior de escalabilidade e de manutenção. Para obter informações adicionais, consulte a seção “Acoplamento e Coesão” neste capítulo.

Mantenha-se no Mesmo Processo

Evite as chamadas remotas de métodos e o vaivém de solicitações e respostas onde possível. As chamadas remotas em fronteiras físicas (processo e máquina) são dispendiosas por causa da serialização e da latência da rede.

É possível hospedar a lógica de negócio do aplicativo no servidor Web, juntamente com a camada de apresentação, ou em um servidor de aplicativos fisicamente separado. Você alcança o desempenho máximo colocando a lógica de negócio no servidor Web, junto ao processo de apresentação da Web. Independentemente do uso ou não da afinidade com o servidor no design, esse método oferece suporte à escalabilidade horizontal e vertical. Você pode adicionar mais hardware aos servidores já existentes ou adicionar mais servidores à camada da Web, como mostra a Figura 3.4.

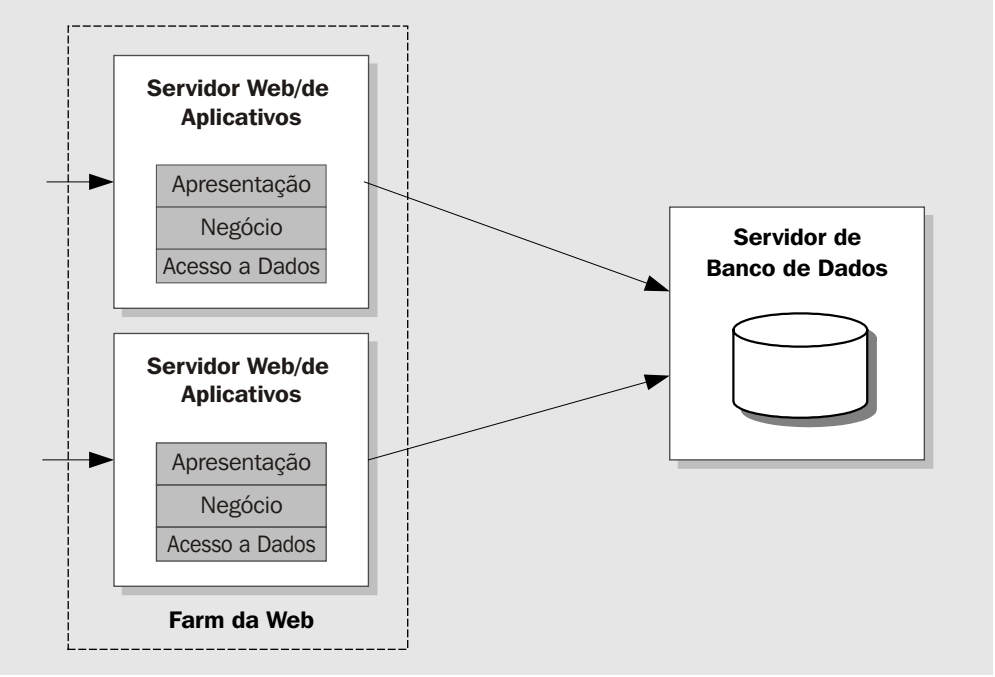

Figura 3.4
Realizando a escalabilidade vertical de servidores Web em um farm Web

Observação: essa arquitetura de implantação ainda pressuporá as camadas lógicas de apresentação, de negócio e acesso a dados. Você deve tornar as camadas lógicas uma meta do design, independentemente da arquitetura física da implantação.

Não Mantenha a Lógica do Aplicativo em um Local Remoto, a Menos que Seja Necessário

Não separe fisicamente a camada de lógica de negócio, a menos que seja necessário e que você tenha avaliado as compensações. A lógica remota pode aumentar a sobrecarga no desempenho. A sobrecarga no desempenho resulta no aumento do vaivém de solicitações e respostas pela rede com custos associados de latência da rede e de serialização.

No entanto, talvez seja necessário separar fisicamente a camada de negócio, como nos seguintes cenários:

- Talvez seja conveniente colocar servidores de *gateway* de negócio com os parceiros chaves.
- Talvez seja necessário adicionar um *front-end* da Web a um conjunto já existente de lógicas comerciais.
- Talvez seja conveniente compartilhar a lógica de negócio entre vários aplicativos clientes.
- A política de segurança da organização talvez proíba a instalação de lógica de negócio nos servidores Web *(front-end)*.
- Talvez seja conveniente descarregar o processamento em outro servidor porque a lógica de negócio talvez exija muitos recursos de computação.
- Se você precisar de uma camada remota de aplicativo, use os padrões de design para ajudar a reduzir a sobrecarga no desempenho. Para obter mais informações, consulte “Comunicação” neste capítulo.

Escalabilidade Vertical versus Horizontal

O método de implementar a escalabilidade é uma consideração crítica no design porque, se você planeja realizar a escalabilidade horizontal da solução através de um *farm* Web, ou uma camada intermediária com balanceamento de carga ou um banco de dados particionado, você precisa verificar se o design oferece suporte a isso.

Ao redimensionar o seu aplicativo, você pode optar por dois itens básicos e pode combiná-los:

- **Escalabilidade vertical: obter uma caixa maior.**
- **Escalabilidade horizontal: obter mais caixas.**

Escalabilidade Vertical: Obter uma Caixa Maior

Com esse método, você adiciona hardware, como processadores, RAM e placas de interface de rede, aos servidores já existentes para oferecer suporte à capacidade maior. É uma opção simples e que pode reduzir os custos. Ela não introduz custos adicionais de suporte e manutenção. No entanto, o ponto único de falha permanece - o que é um risco. Além de um determinado limite, adicionar mais hardware aos servidores existentes talvez não produza os resultados desejados. Para que a escalabilidade vertical de um aplicativo seja eficiente, a estrutura subjacente, o tempo de execução e a arquitetura do computador também devem escalar verticalmente. Ao fazer a escalabilidade vertical, considere os recursos aos quais o aplicativo está vinculado. Se fizer uso intensivo de memória ou de rede, adicionar recursos de CPU não ajudará.

Escalabilidade Horizontal: Obter Mais Caixas

Para realizar a escalabilidade horizontal, você deverá poder adicionar mais servidores e usar soluções de balanceamento de carga e de *cluster.* Além de lidar com a carga adicional, o cenário de escalabilidade horizontal também protege contra falhas do hardware. Se um servidor falha, há outros servidores no *cluster* que podem assumir o controle da carga. Por exemplo, você pode hospedar vários servidores Web em um *farm* Web que hospeda camadas de apresentação e de negócio, ou pode particionar fisicamente a lógica de negócio do aplicativo, utilizando uma camada intermediária com balanceamento de carga, tendo em um farm separado na camada de apresentação. Se o seu aplicativo está limitado pela E/S e você precisa oferecer suporte a um banco de dados extremamente grande, você pode particionar o banco de dados em vários servidores de banco de dados. Em geral, a capacidade de um aplicativo de implementar a escalabilidade horizontal depende mais de sua arquitetura do que da infra-estrutura subjacente.

Diretrizes

Ao implementar a escalabilidade, considere os seguintes métodos:

- **Considere a necessidade de oferecer suporte à escalabilidade horizontal.**
- **Considere antecipadamente as implicações e as compensações relativas ao design.**
- **Considere a possibilidade de particionar o banco de dados na fase de design.**

Considere a Necessidade de Oferecer Suporte à Escalabilidade Horizontal

Aumentar o poder dos processadores e aumentar a memória, pode ser uma solução econômica. Ela também evita a introdução de custos adicionais de gerenciamento associados à escalabilidade horizontal e ao uso de *farms* Web e de tecnologia de *cluster.* É preciso analisar as opções de escalabilidade vertical primeiro e realizar testes de desempenho para verificar se a escalabilidade vertical da solução atende aos critérios definidos para a escalabilidade e se oferece suporte ao número necessário de usuários concorrentes com um nível aceitável de desempenho. Você deve ter um plano para a escalabilidade do sistema que acompanhe o crescimento observado.

Se a escalabilidade vertical da solução não fornecer capacidade de escalabilidade adequada porque os limites de CPU, E/S ou memória são alcançados, será preciso implementar a escalabilidade horizontal e introduzir servidores adicionais. Para garantir uma escalabilidade horizontal bem-sucedida do aplicativo, considere as seguintes práticas no design:

- **Você precisa estar apto a implementar a escalabilidade horizontal dos gargalos, onde quer que se encontrem.** Se os gargalos estiverem em um recurso compartilhado que não possa ser redimensionado, você tem um problema. No entanto, ter uma classe de servidores que tenham afinidade com um tipo de recurso poderá ser benéfico, mas eles devem ser escaláveis de forma independente. Por exemplo, se você tiver um único SQL Server™ que forneça um diretório, todos irão usá-lo. Nesse caso, quando o servidor tornar-se um gargalo, você poderá implementar a escalabilidade horizontal ao usar várias cópias. Criar uma afinidade entre os dados do diretório e os SQL Servers que fornecem os dados permite especializar esses servidores e não causa problemas posteriores de escalabilidade. Portanto, neste caso, a afinidade é uma boa idéia.
- **Defina um design em camadas e com acoplamento fraco.** Um design em camadas, com acoplamento fraco, com interfaces limpas e com capacidade de trabalhar remotamente, é mais fácil de escalar horizontalmente do que outras camadas de acoplamento rígido com interações intensas. Um design em camadas terá pontos de ajuste naturais, tornando-o ideal para a escalabilidade horizontal nas fronteiras de cada camada. O truque é encontrar as fronteiras certas. Por exemplo, a lógica de negócio pode ser relocada com mais facilidade para uma *farm* de servidores de aplicativos numa camada física intermediária e com balanceamento de carga.

Considere Antecipadamente as Implicações e as Compensações em Relação ao Design

Você precisa considerar os aspectos da escalabilidade que podem variar conforme a camada de lógica, camada física, ou o tipo de dado. Conheça os prós e contras de antemão e saiba onde você tem flexibilidade e onde não tem. Implementar a escalabilidade vertical e, em seguida, a horizontal, com servidores da Web ou de aplicativos, pode não ser o melhor método. Por exemplo, embora você possa ter um servidor com oito processadores nessa função, considerações financeiras provavelmente o direcionarão para um conjunto de servidores menores, e não ao uso de poucos servidores de grande porte. Por outro lado, implementar a escalabilidade vertical e, em seguida, a horizontal pode ser o método certo para os servidores de banco de dados, dependendo da função dos dados e de como são usados. Além das considerações técnicas e de desempenho, você também precisa considerar as implicações operacionais e de gerenciamento, bem como o custo total de propriedade relacionado.

Use os seguintes pontos para ajudar a avaliar a estratégia de escalabilidade.

- **Componentes sem armazenamento de estado *(stateless).*** Se você tiver componentes *stateless* (por exemplo: um *front-end* da Web e componentes de negócio sem estados), esse aspecto do seu design oferecerá suporte à escalabilidade vertical e horizontal. Geralmente, você otimiza o preço e o desempenho dentro das fronteiras das outras restrições que você possa ter. Por exemplo, servidores Web ou de aplicativos com dois processadores podem ser excelentes em relação a preço e desempenho se comparados com servidores que têm quatro processadores. Ou seja, quatro servidores com dois processadores pode ser uma opção melhor que dois servidores com quatro processadores. Você também precisa considerar outras restrições, como o número máximo de servidores que pode ter atrás de uma infra-estrutura específica de balanceamento de carga. Em geral, não haverá compensações no design se você optar por um design *stateless.* Você otimiza preço, desempenho e capacidade de gerenciamento.
- **Dados.** Em relação aos dados, as decisões dependem em grande parte do seu tipo:
  - **Dados estáticos, de referência e somente leitura.** Para esse tipo de dado, você pode ter facilmente muitas réplicas nos lugares certos se isso ajudar no desempenho e na escalabilidade. Isso tem um impacto mínimo no design e pode ser amplamente direcionado pelas considerações de otimização. Consolidar vários bancos de dados independentes e separados logicamente em um único servidor de banco de dados pode ou não ser apropriado, mesmo que você possa fazer isso em termos de capacidade. Espalhar réplicas destes dados mais próximas dos consumidores pode ser um método igualmente válido. No entanto, não se esqueça de que, sempre que você replicar, terá um sistema fracamente sincronizado.
  - **Dados dinâmicos (muitas vezes transitórios) que podem ser facilmente particionados.** Esses são dados relevantes apenas a um usuário ou sessão específica (e, se solicitações subseqüentes puderem chegar aos servidores Web ou de aplicativos diferentes, todos eles precisarão acessá-los), mas os dados do usuário A não estão relacionados de qualquer forma aos dados do usuário B. Por exemplo, os carrinhos de compra e estados da sessão pertencem a essa categoria. O manuseio desses dados é um pouco mais complicado que o manuseio de dados estáticos e somente leitura, mas mesmo assim é possível otimizar e distribuir com bastante facilidade. Isso acontece porque esse tipo de dado pode ser particionado. Não há dependências entre grupos, até o nível do usuário individual. O aspecto importante desses dados é que você não os consulta através das várias partições. Por exemplo, você solicita o conteúdo do carrinho de compras do usuário A, mas não solicita a exibição de todos os carrinhos que contêm um item específico.
  - **Dados Principais.** Esse tipo de dado deve ser bem mantido e protegido. Este é o principal caso em que o método “escalabilidade vertical e, em seguida, horizontal” normalmente se aplica. Geralmente, você não deseja manter esse tipo de dado em muitos lugares devido à complexidade de sua sincronização. É uma situação clássica na qual você gostaria da escalabilidade vertical até o nível máximo possível (em teoria, mantendo uma única instância lógica, com *cluster* apropriado) e, apenas quando não fosse suficiente, consideraria a possibilidade de fazer a escalabilidade horizontal com partições e distribuição. Avanços na tecnologia de banco de dados (como visões particionadas distribuídas) tornaram o particionamento muito mais fácil, embora isso deva ser feito apenas quando necessário. Isto é raramente usado devido ao banco ser muito grande, mas é frequentemente forçado por outras considerações, como a quem pertence os dados, a distribuição geográfica, a proximidade em relação aos consumidores e a disponibilidade.

Considere a Possibilidade de Particionar o Banco de Dados na Fase de Design

Se o aplicativo usa um banco de dados muito grande e você prevê um gargalo de E/S, antecipe o design do particionamento do banco de dados. Mover posteriormente para um banco de dados particionado muitas vezes resulta em uma quantidade significativa de retrabalho dispendioso e, freqüentemente, na recriação completa do banco de dados.

O particionamento fornece vários benefícios:

- A capacidade de restringir consultas a uma única partição, limitando, assim, o uso de recursos a apenas uma fração dos dados.
- A capacidade de envolver várias partições, obtendo, assim um paralelismo maior e um desempenho superior, pois você tem mais discos trabalhando para recuperar os dados.

Não se esqueça de que, em algumas situações, o uso de várias partições talvez não seja o método apropriado e que ele pode causar um impacto negativo. Por exemplo, algumas operações que usam vários discos poderiam ser executadas com mais eficiência com os dados concentrados. Portanto, ao particionar, considere os benefícios juntamente com os métodos alternativos.

Mais Informações

Para obter mais informações sobre escalabilidade vertical versus escalabilidade horizontal, consulte os seguintes recursos:

- "Implantação e Infra-estrutura" no Capítulo 4, "Análise da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".
- "SQL: Escalabilidade Vertical versus Escalabilidade Horizontal" no Capítulo 14, "Aprimorando o Desempenho do SQL Server".
- "Procedimento: Executar Planejamento de Capacidade de Aplicativos .NET" na seção "Procedimentos" deste guia.

Questões de Arquitetura e de Design

Além de afetar o desempenho, o design inadequado pode limitar a escalabilidade do aplicativo. O desempenho está relacionado com o atendimento dos objetivos de tempo de resposta, taxa de transferência e utilização de recursos. A escalabilidade refere-se à capacidade de lidar com carga de trabalho adicional, sem afetar negativamente o desempenho, por meio da adição de recursos, como maior capacidade de CPU, memória ou armazenamento.

As vezes, uma decisão de design envolve uma compensação entre desempenho e escalabilidade. A Figura 3.5 destaca alguns dos principais problemas que podem ocorrer nas camadas de aplicativos distribuídos.

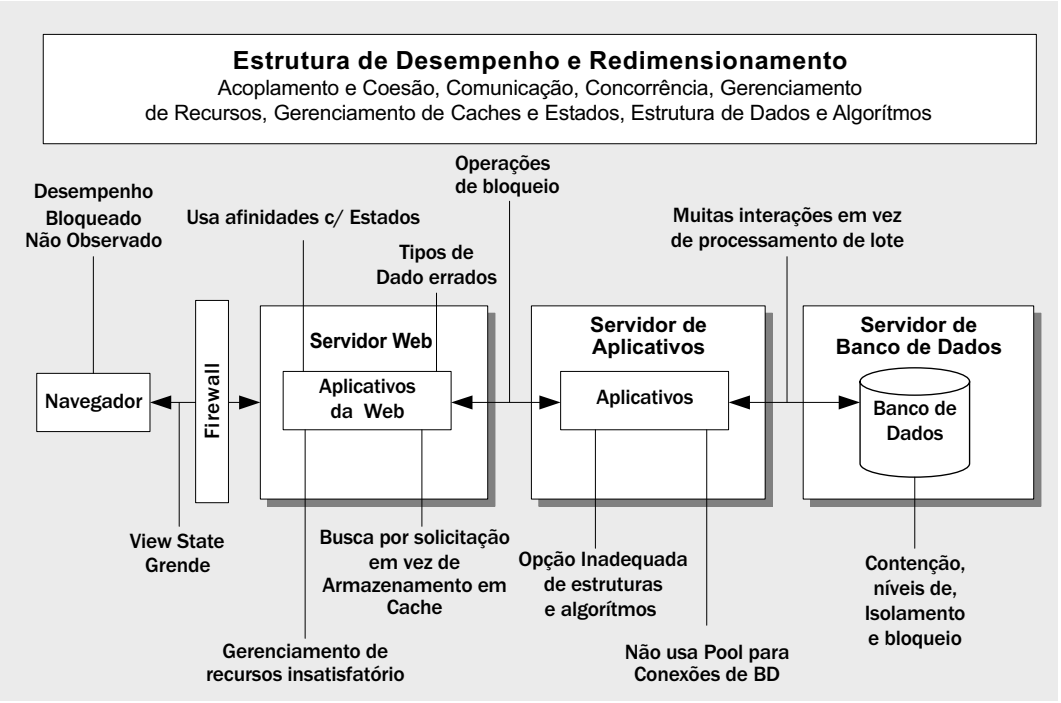

Figura 3.5
Questões de desempenho comuns entre camadas do aplicativo

As questões em destaque podem se aplicar à todas as camadas do aplicativo. Por exemplo, um aplicativo que não está respondendo pode ser o resultado de problemas de concorrência no código de uma página Web, na camada intermediária do aplicativo ou no banco de dados. Como alternativa, pode ser o resultado direto de problemas de comunicação causados pelo design de uma interface com muitas interações ou pela falha em consolidar recursos compartilhados. Nestes casos, o desempenho insatisfatório pode se tornar aparente apenas quando vários usuários acessarem o aplicativo simultaneamente.

A Tabela 3.1 lista os principais problemas que podem resultar de um design insatisfatório. Esses problemas foram organizados em categorias definidas pela estrutura de desempenho e de escalabilidade introduzida no Capítulo 1, “Princípios Básicos de Engenharia de Desempenho”.

Tabela 3.1: Problemas Potenciais de Desempenho com Design Insatisfatório

Categoria	Problema Potencial Causado por Design Insatisfatório
Acoplamento e Coesão	Escalabilidade limitada devido à afinidade com o servidor e com recursos. Mistura da lógica de negócio com a de apresentação, o que limita as opções de escalabilidade horizontal do aplicativo.
Comunicação	Tráfego da rede e latência maiores devido às chamadas freqüentes entre camadas. Protocolos de transporte e formatos de conexão inapropriados. Grande volume de dados em redes com largura de banda limitada.
Concorrência	Chamadas com bloqueio e bloqueios não granulares que emperram a interface de usuário do aplicativo. Sobrecarga do processador e da memória devido ao uso ineficiente de *Threads* Contenção no banco de dados devido aos níveis impróprios de isolamento de transação.
Gerenciamento de Recursos	Conjuntos grandes de trabalho devido ao gerenciamento ineficiente da memória. Escalabilidade limitada e taxa de transferência reduzida devido à falha em liberar e consolidar recursos compartilhados. Desempenho reduzido devido ao uso excessivo de *late binding* e criação/destruição de objetos de forma ineficiente.
Armazenamento em Cache	Armazenamento em cache de recursos compartilhados, perdas de cache, falha em expirar itens, design insatisfatório do cache e falta de um mecanismo de sincronização do cache para obter escalabilidade horizontal.
Gerenciamento de Estados	Afinidade de estado, escalabilidade reduzida, design inapropriado do estado e armazenamento de estado inadequado.
Estruturas de Dados e Algoritmos	Conversão excessiva de tipos. Pesquisas ineficientes. Escolha incorreta de estrutura de dados para várias funções, como pesquisa, classificação, enumeração e tamanho dos dados.

As próximas seções deste capítulo apresentam recomendações de design, organizadas pela categoria de desempenho.

Acoplamento e Coesão

Reduzir o acoplamento e aumentar a coesão são dois princípios fundamentais para se aumentar a capacidade de escalabilidade do aplicativo. Acoplamento é o nível de dependência (no design ou no momento da execução) que existe entre partes de um sistema. Coesão avalia quantos componentes diferentes aproveitam o processamento e os dados compartilhados. Um aplicativo desenvolvido de forma modular contém um conjunto de componentes altamente coesos que se acoplam livremente.

Para ajudar a garantir os níveis de acoplamento e de coesão em seu design considere as seguintes recomendações.

- **Inclua no design o acoplamento fraco.**
- **Inclua no design a coesão alta.**
- **Particione a funcionalidade do aplicativo em camadas lógicas.**
- **Use *early binding* onde possível.**
- **Avalie a afinidade de recursos.**

Inclua no Design o Acoplamento Fraco

Procure minimizar o acoplamento no aplicativo e entre os seus componentes. Se você tiver acoplamento rígido e precisar fazer alterações, as mudanças serão, provavelmente, sentidas pelos componentes totalmente integrados. Com componentes acoplados de forma fraca, as alterações são limitadas, pois a complexidade dos componentes individuais é encapsulada pelos consumidores. Além disso, o acoplamento fraco fornece maior flexibilidade na escolha de estratégias otimizadas de desempenho e de escalabilidade para os diferentes componentes do sistema, de forma independente.

Pode haver determinados cenários críticos em relação ao desempenho nos quais seja necessário acoplar a lógica de negócio, de apresentação e de acesso a dados de forma integral, por você não ter recursos suficientes para a pequena sobrecarga devido ao acoplamento fraco. Por exemplo, a inserção de código via *inlining* remove a sobrecarga da criação e da chamada de vários objetos, do acionamento de uma pilha de chamada para chamar métodos diferentes, da execução de pesquisas em tabela virtual etc. No entanto, na maioria dos casos, os benefícios do acoplamento fraco superam essas pequenas perdas no desempenho.

Alguns dos padrões e princípios que permitem o acoplamento fraco são:

- **Separe a interface da implementação.** Fornecer fachadas *(façades)* nas fronteiras críticas do aplicativo leva à melhor capacidade de manutenção e ajuda a definir unidades de trabalho que encapsulam a complexidade interna. Para obter um bom exemplo desse método, consulte a implementa do "*Exception Management Application Block for .NET*" *no MSDN®,* em *http://msdn.microsoft.com /library/default.asp?url=/library/en-us/dnbda/html/emab-rm.asp.*
- **Comunicação baseada em mensagens.** As filas de mensagens oferecem suporte à solicitações assíncronas e você poderá usar uma fila no cliente se precisar de respostas. Isso fornece flexibilidade adicional na determinação de quando as solicitações devem ser processadas.

Inclua no Design a Coesão Alta

As entidades relacionadas de forma lógica, como classes e métodos, devem ser agrupadas. Por exemplo, uma classe deve conter um conjunto de métodos relacionados de forma lógica. Da mesma forma, um componente deve conter classes relacionadas de forma lógica.

A coesão fraca entre componentes costuma resultar em um aumento no vaivém de solicitações e respostas porque as classes ou os componentes não estão agrupados de forma lógica e podem terminar em camadas diferentes. Isso pode forçá-lo a exigir uma mistura de chamadas locais e remotas para concluir uma operação lógica. Você pode evitar isso com agrupamento apropriado. Isso também ajuda a reduzir a complexidade, eliminando seqüências complexas de interações entre vários componentes.

Particione a Funcionalidade do Aplicativo em Camadas Lógicas

Usar camadas lógicas para particionar o aplicativo evita que as lógicas de negócio, de apresentação e de acesso a dados fiquem espalhadas. Essa organização lógica leva a um design coeso onde as classes e os dados relacionados estão próximos, geralmente, dentro de uma única fronteira. Isso ajuda a otimizar o uso de recursos custosos. Por exemplo, agrupar todas as classes da lógica de acesso aos dados garante a possibilidade de compartilhar um *pool* de conexões do banco de dados.

Use *Early Binding* Onde Possível

Dê preferência ao uso de *early binding* (decisão de chamada de um método em tempo de compilação) onde possível, pois isso reduz a sobrecarga do tempo de execução e é a maneira mais eficiente de chamar um método.

O *late binding* (decisão de chamada de um método em tempo de execução) fornece um acoplamento fraco, mas afeta o desempenho, pois os componentes devem ser localizados e carregados de forma dinâmica. Use a *late binding* apenas quando for absolutamente necessário, como para obter capacidade de extensão.

Avalie a Afinidade de Recursos

Compare e verifique os prós e os contras. A afinidade com um recurso específico pode melhorar o desempenho em algumas situações. No entanto, embora a afinidade possa satisfazer suas metas de desempenho hoje, a afinidade de recursos pode dificultar o escalabilidade do aplicativo. Por exemplo, a afinidade com um recurso específico pode limitar ou impedir o uso eficiente de hardware adicional nos servidores, como mais processadores e memória. A afinidade com os servidores também pode impedir a escalabilidade horizontal.

Alguns exemplos de afinidades que podem causar problemas de escalabilidade incluem:

- **Usar armazenamento de estados num único processo.** Como resultado, todas as solicitações de um cliente específico devem ser encaminhadas para o mesmo servidor.
- **Usar lógica do aplicativo que introduza afinidade de *threads*.** Isso força as *threads* a serem executadas em um conjunto específico de processadores. Isso tolhe a capacidade do *scheduler* de agendar *threads* entre processadores, o que causa uma redução nos ganhos de desempenho causada pela falta de processamento paralelo.

Mais Informações

Para obter mais informações sobre acoplamento e coesão, consulte "Acoplamento e Coesão" no Capítulo 4, "Análise da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Comunicação

Os benefícios das arquiteturas distribuídas, como escalabilidade, tolerância à falhas e manutenabilidade, estão bem documentados. No entanto os níveis crescentes de comunicação e de coordenação inevitavelmente afetam o desempenho.

Para evitar armadilhas comuns e reduzir a sobrecarga no desempenho, considere as seguintes diretrizes:

- **Escolha o mecanismo apropriado de comunicação remota.**
- **Crie interfaces concisas.**
- **Considere a forma de transferência de dados entre camadas.**
- **Reduza a quantidade de dados enviados pela conexão.**

- **Agrupe o trabalho em lotes para reduzir chamadas pela rede.**
- **Reduza as transições entre fronteiras.**
- **Considere a comunicação assíncrona.**
- **Considere o enfileiramento de mensagens.**
- **Considere um modelo de invocação *"fire and forget"* (dispare e esqueça).**

Escolha o Mecanismo Apropriado de Comunicação Remota

Sua escolha do mecanismo de transporte é influenciada por vários fatores, incluindo largura de banda disponível, quantidade de dados a serem transferidos, média de usuários concorrentes e restrições de segurança, como *firewalls*.

Serviços é o tipo de comunicação preferencial entre fronteiras do aplicativo, incluindo fronteiras de plataforma, implantação e confiança. A tecnologia do objeto, como Enterprise Services ou .NET remoto, deve ser usada apenas na implementação de um serviço. Use o Enterprise Services apenas se precisar do conjunto de recursos adicionais (como *pool* de objetos, transações distribuídas declarativas, segurança baseada em função e componentes enfileirados) ou quando o aplicativo se comunicar entre componentes de um servidor local e você tiver problemas de desempenho nos serviços da Web.

Você deve escolher os protocolos de transporte seguro, como HTTPS, apenas onde necessário e somente para aquelas partes de um site que exijam.

Crie Interfaces Concisas

Crie interfaces concisas e evite as interfaces com muitas interações. Interfaces com muitas interações exigem muito vaivém de solicitações e respostas para executar uma única operação lógica, o que consume recursos do sistema e, possivelmente, da rede. As interfaces concisas permitem transferir todos os parâmetros de entrada necessários e concluir uma operação lógica em um número mínimo de chamadas. Por exemplo, você pode empacotar várias chamadas de GET e SET em uma única chamada de método. O empacotador coordenaria, então, o acesso às propriedades internamente.

Você pode ter uma fachada *(façade)* com interfaces concisas que empacotam componentes existentes visando reduzir o vaivém de solicitações e respostas. Esta fachada encapsulará a funcionalidade de um conjunto de componentes que se tornam ocultos, e fornecerá uma interface mais simples ao cliente. Esta interface deve coordenar internamente a interação entre vários componentes da camada de serviço. Dessa maneira, o cliente estará menos propenso a alterações que afetam a camada de negócio e a fachada *(façade)* também ajuda a reduzir o vaivém de solicitações e respostas entre cliente e servidor.

Considere a Forma de Transferência de Dados Entre Camadas

Transferir dados entre camadas envolve sobrecarga de processamento para serialização e utilização da rede. As suas opções incluem o uso de objetos ***DataSet*** do ADO.NET, objetos ***DataSet*** fortemente tipados, coleções, XML ou objetos customizados e tipos passados por valor.

Para tomar uma decisão de design fundamentada, considere as seguintes perguntas:

- **Em que formato os dados são recuperados?**

  Se o cliente recupera dados em um determinado formato, talvez seja trabalhoso transformá-los. A transformação é uma exigência comum, mas você deve evitar várias transformações à medida que os dados fluem pelo aplicativo.

- **Em que formato os dados são consumidos?**

  Se o cliente exige dados na forma de um conjunto de objetos de um tipo específico, uma coleção fortemente tipada é uma opção lógica e correta.

- **Quais são os recursos exigidos pelo cliente?**

  Um cliente pode esperar a disponibilização de determinadas facilidades pelos objetos que recebe no retorno de uma chamada à camada de negócio. Por exemplo, se o cliente precisa estar apto a exibir os dados de várias maneiras, precisa atualizar dados no servidor usando concorrência otimista, e precisa lidar com relações complexas entre os vários conjuntos de dados, um ***DataSet*** adequa-se perfeitamente a esse tipo de exigência.

  No entanto, a criação do ***DataSet*** é trabalhosa tanto devido à sua hierarquia interna de objetos quanto ao uso de uma grande quantidade de memória. Além disso, a serialização padrão do ***DataSet*** incorre em um custo de processamento significativo, mesmo ao usar ***BinaryFormatter.***

  Outras exigências do cliente podem incluir a necessidade de validação, vinculação (*binding*) de dados, classificação e compartilhamento de conjuntos entre cliente e servidor.

  Para obter mais informações sobre como melhorar o desempenho da serialização de ***DataSet*** consulte "Procedimento: Melhorar o Desempenho da Serialização", na seção "Procedimentos" deste guia.

- **Os dados podem ser agrupados de forma lógica?**

  Se os dados exigidos pelo cliente representam um agrupamento lógico, como os atributos que descrevem um funcionário, considere o uso de um tipo customizado. Por exemplo, você poderia retornar detalhes do funcionário em um objeto do tipo *struct* que contivesse nome, endereço e número do funcionário como membro.

  O principal benefício das classes personalizadas para o desempenho é que elas lhe permitem criar seus próprios mecanismos otimizados de serialização para reduzir o espaço de comunicação entre computadores.

- **Você precisa considerar a interoperabilidade entre plataformas?**

  XML é um padrão aberto e é a representação de dados ideal para interoperabilidade entre plataformas e comunicação com sistemas externos (e heterogêneos).

  As questões de desempenho devem considerar a análise do esforço necessário para processar grandes seqüências de caracteres XML. As seqüências de caracteres grandes e verborrágicas também consomem grande quantidade de memória. Para obter mais informações sobre processamento XML, consulte o Capítulo 9, "Melhorando o Desempenho de XML".

Mais Informações

Para obter mais informações sobre como transferir dados entre camadas, consulte "Acesso a Dados" no Capítulo 4, "Análise da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Reduza a Quantidade de Dados Enviados pela Rede

Evite enviar dados redundantes pela rede. Você pode otimizar a comunicação de dados usando alguns padrões de design:

- **Use *wrappers* para concisão.** Você pode desenvolver um objeto *wrapper* com uma interface concisa para encapsular e coordenar a funcionalidade de um ou mais objetos que não foram projetados para acesso remoto eficiente. O objeto *wrapper* abstrai a complexidade e as relações entre vários objetos comerciais, fornece uma interface concisa otimizada para acesso remoto e

ajuda a fornecer um sistema de acoplamento fraco. Ele fornece aos clientes uma interface simples em funcionalidade para vários objetos de negócio. Ele também define unidades de trabalho menos granuladas e encapsula alterações. Esse método é descrito pelos padrões de design de fachada *(façade)*.

- **Empacote e retorne os dados necessários.** Em vez de fazer uma chamada remota para buscar itens individuais de dados, você busca por valor um objeto de dados em uma única chamada remota. Você, então, opera localmente nos dados armazenados em cache local. Isso pode ser suficiente em muitos cenários. Em outros cenários, nos quais você precisa, ao final, atualizar os dados no servidor, o objeto empacotador expõe um único método que é chamado para enviar os dados de volta para o servidor. Esse método é demonstrado no seguinte fragmento de código:

```
struct Employee {
                private int _employeeID;
                private string _projectCode;
                public int EmployeeID {
                        get {return  employeeID;}
                }
                public string ProjectCode {
                        get {return _projectCode;}
                }
                public SetData () {
                // Send the changes back and update the changes
                // on the remote server
                }
        }
```

Além de encapsular os dados relevantes, o objeto pode expor um ***SetData*** ou um método de atualização dos dados de volta no servidor. As propriedades públicas agem localmente nos dados em cache, sem fazer uma chamada de método remota. Esses métodos individuais também podem executar validação de dados. Essa abordagem às vezes é chamada de “padrão de design de transferência de dados por objetos”.

- **Serialize apenas o necessário.** Analise como os seus objetos implementam a serialização para garantir que apenas os dados necessários serão serializados. Isso reduz o tamanho dos dados e a sobrecarga da memória. Para obter mais informações, consulte "Procedimento: Melhorar o Desempenho da Serialização" na seção "Procedimentos" deste guia.
- **Use paginação de dados.** Use uma solução de paginação quando precisar apresentar grande volume de dados ao cliente. Isso ajuda a reduzir a carga de processamento no servidor, no cliente e na rede e fornece uma melhor experiência para o usuário. Para obter mais informações sobre várias técnicas de implementação, consulte “Procedimento: Paginar Registros nos Aplicativos .NET”, na seção “Procedimentos” deste guia.

- **Use paginação de dados.** Use uma solução de paginação quando precisar apresentar grande volume de dados ao cliente. Isso ajuda a reduzir a carga de processamento no servidor, no cliente e na rede e fornece uma melhor experiência para o usuário. Para obter mais informações sobre várias técnicas de implementação, consulte “Procedimento: Paginar Registros nos Aplicativos .NET”, na seção “Procedimentos” deste guia.
- **Considere as técnicas de compactação.** Em situações nas quais seja absolutamente necessário enviar grande quantidade de dados e, nas quais a largura de banda da rede seja limitada, considere as técnicas de compactação, como a compactação do HTTP 1.1.

Agrupe o Trabalho em Lotes para Reduzir Chamadas pela Rede

Agrupe o seu trabalho em lotes para reduzir a quantidade de chamadas remotas pela rede. Alguns exemplos de agrupamento em lote incluem:

- **Atualizações em lote.** O cliente envia várias atualizações num único lote a um servidor de aplicativos remoto, em vez de fazer várias chamadas remotas de atualizações de uma transação.
- **Consultas em lote.** Várias consultas SQL podem ser agrupadas em lote separadas por ponto-e-vírgula ou usando os procedimentos armazenados.

Reduza as Transições Entre Limites

Mantenha as entidades que interagem com freqüência dentro da mesma fronteira, como o mesmo domínio de aplicativos, processo ou máquina, a fim de reduzir a sobrecarga na comunicação. Ao fazer isso, considere as compensações entre desempenho e escalabilidade. Um domínio *(Application Domain)* com um processo único e um único aplicativo fornece um desempenho ótimo, mas uma solução com vários servidores fornece benefícios representativos de escalabilidade e permite a você redimensionar a sua solução.

As principais fronteiras que você precisa considerar são:

- Entre código gerenciado e não gerenciado.
- Entre processos.
- Entre servidores.

Considere a Comunicação Assíncrona

Para evitar o bloqueio de *threads*, considere o uso de chamadas assíncronas para qualquer tipo de operação de E/S. As chamadas síncronas continuam bloqueadas dentro de *threads* enquanto aguardam resposta. As chamadas assíncronas oferecem a flexibilidade de liberar o processamento da *thread* para realizar algum trabalho útil (como, talvez, lidar com novas solicitações de aplicativos do servidor). Como resultado, as chamadas assíncronas são úteis para chamadas de execução potencialmente longas que não sejam CPU intensivas. O .NET Framework fornece um padrão de design assíncrono para implementação de comunicação assíncrona.

Observe que, na verdade, cada chamada assíncrona usa uma *thread* de trabalho do *pool* de *threads* do processo. Se elas forem usadas excessivamente em um sistema com apenas uma CPU, isso poderá levar à parada da *thread* (*starvation*) e à troca excessiva de *threads*, degradando, assim, o desempenho. Se os seus clientes não precisam de resultados retornados imediatamente, considere o uso de filas no cliente e no servidor como um método alternativo.

Considere o Enfileiramento de Mensagens

Um método voltado para mensagens e acoplado de forma livre permite:

- Separar o tempo de vida do cliente em relação ao servidor, o que ajuda a reduzir a complexidade e aumenta a capacidade de recuperação dos aplicativos distribuídos.
- Melhorar a capacidade de resposta e a taxa de transferência, pois a solicitação atual não depende da conclusão de um processo potencialmente lento.
- Descarregar trabalhos de uso intensivo de processamento em outros servidores.
- Incluir processos consumidores adicionais que lêem de uma fila comum de mensagens para ajudar a melhorar a escalabilidade.
- Transferir o processamento para períodos que não sejam de pico.
- Reduzir a necessidade de acesso sincronizado aos recursos.

O método básico de enfileiramento de mensagens é mostrado na Figura 3.6. O cliente envia solicitações de processamento na forma de mensagens à fila de solicitações. A lógica de processamento (a qual pode ser implementada como vários processos paralelos para escalabilidade) lê as solicitações da fila, executa o trabalho necessário e coloca as mensagens de resposta na respectiva fila, e elas, por sua vez, são lidas pelo cliente.

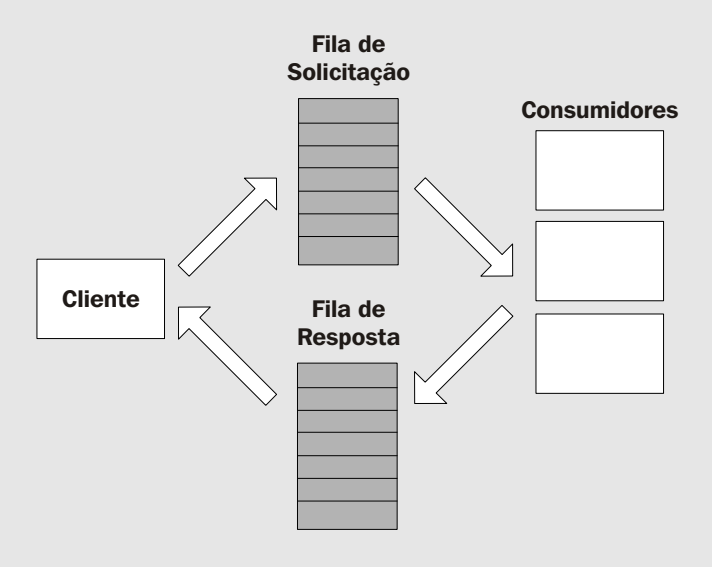

Figura 3.6
Enfileiramento de mensagens com respostas

O enfileiramento de mensagens apresenta outros desafios para o design:

- Como será o comportamento do aplicativo se as mensagens não forem entregues ou recebidas?
- Como será o comportamento do aplicativo se chegarem mensagens duplicadas ou fora da seqüência? Seu design não pode ter dependências de ordem e tempo.

Considere um Modelo de Invocação *"Fire and Forget"*

Se o cliente não precisa aguardar os resultados de uma chamada, você pode usar um método *"fire and forget"* para melhorar o tempo de resposta e evitar o bloqueio enquanto a longa execução da chamada pelo servidor é concluída. O método *"fire and forget"* pode ser implementado de várias maneiras.

- O cliente e o servidor podem ter filas de mensagens.
- Se você estiver usando os serviços da Web do ASP.NET ou a parte remota do .NET, poderá usar o atributo ***One Way***.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre o modelo de invocação assíncrona do .NET Framework, consulte o Capítulo 5, "Melhorando o Desempenho do Código Gerenciado", e "*Asynchronous Programming Design Pattern*", no Kit de Desenvolvimento de Software (SDK) do .NET Framework no MSDN.
- Se você precisar de outros *Enterprise Services* além da comunicação assíncrona, considere o uso de Componentes Enfileirados COM+. Para obter mais informações sobre como usar Componentes Enfileirados, consulte o Capítulo 8, "Melhorando o Desempenho dos Enterprise Services".
- Para obter mais informações sobre como usar o método "*fire and forget*" com WebServices, consulte o Capítulo 10, "Melhorando o Desempenho dos WebServices ".
- Para obter mais informações sobre *"fire and forget"* com o .NET Remoting, consulte o Capítulo 11, "Melhorando o Desempenho do *Remoting*".

Mais Informações

Para obter mais informações sobre comunicação, consulte "Comunicação" no Capítulo 4, "Revisão Arquitetura e Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Concorrência

Um dos principais benefícios das arquiteturas distribuídas é que você pode oferecer suporte a altos níveis de concorrência e de processamento paralelo. No entanto, há muitos fatores, como contenção dos recursos compartilhados, bloqueio de chamadas assíncronas e bloqueio, que reduzem a concorrência. O seu design precisa considerar esses fatores. Uma meta de design fundamental é reduzir a contenção e aumentar a concorrência.

Use as diretrizes a seguir para ajudar a alcançar essa meta:

- **Reduza a contenção diminuindo o tempo de bloqueio.**
- **Faça o balanceamento entre bloqueios granulados e não-granulados.**
- **Escolha um nível apropriado de isolamento de transação.**
- **Evite transações atômicas de execução longa.**

Reduza a Contenção Diminuindo o Tempo de Bloqueio

Se você usa primitivas de sincronização para sincronizar o acesso a recursos ou dados compartilhados, não se esqueça de reduzir o tempo de bloqueio. A alta contenção de recursos compartilhados resulta no enfileiramento das solicitações e no aumento do tempo de espera do chamador. Por exemplo, mantenha bloqueios apenas sobre aquelas linhas do código que precisam de atomicidade. Ao executar operações de banco de dados, considere o uso de particionamento e de grupos de arquivos para distribuir as operações de E/S pelos vários discos rígidos.

Faça o Balanceamento entre Bloqueios Granulados e Não-Granulados

Analise e teste o código em relação à sua política de quantidade de bloqueios, de tipo de bloqueio, e do ponto de acionamento e liberação do bloqueio. Determine o balanceamento ideal entre bloqueios granulados e não-granulados. Os bloqueios não-granulados podem resultar em maior contenção dos recursos. Os bloqueios granulados que bloqueiam apenas as linhas relevantes do código para que se utilize uma quantidade mínima de tempo, são os preferenciais, pois geram menos contenção de bloqueio e melhoram a concorrência. No entanto, ter em excesso bloqueios granulados pode sobrecarregar o processamento, bem como aumentar a complexidade do código e as chances de erros e travamentos.

Escolha um Nível Apropriado de Isolamento de Transação

Você precisa selecionar o nível de isolamento apropriado para garantir a preservação da integridade dos dados sem afetar o desempenho do aplicativo. Níveis diferentes de isolamento trazem consigo garantias diferentes em relação à integridade dos dados, bem como diferentes níveis de desempenho. O SQL Server oferece suporte a quatro níveis de isolamento ANSI:

- Leitura Não Confirmada *(Read Uncommited)*
- Leitura Confirmada *(Read Commited)*
- Leitura Repetitiva *(Repeatable Read)*
- Serializável *(Serializable)*

> **Observação:** o suporte aos níveis de isolamento pode variar de um banco de dados para outro. Por exemplo, o Oracle 8i não oferece suporte ao nível de isolamento *Leitura Não Confirmada*.

A leitura *Não Confirmada* oferece o melhor desempenho, mas fornece as menores garantias de integridade. A leitura Serializável oferece o menor desempenho, mas o máximo em integridade dos dados.

É preciso avaliar cuidadosamente o impacto da alteração do nível de isolamento padrão do SQL Server (Leitura Confirmada). Alterá-lo para um valor superior ao necessário pode aumentar a contenção nos objetos do banco de dados. Reduzí-lo pode aumentar o desempenho, prejudicando, porém, a integridade dos dados.

Escolher os níveis de isolamento adequados exige a compreensão da forma como o banco de dados lida com bloqueios e do tipo de tarefa que o aplicativo executa. Por exemplo, se a sua transação envolve algumas linhas de uma tabela, é pouco provável que ela interfira muito em outras transações, ao contrário daquela que envolve muitas tabelas e que talvez precise bloquear muitas linhas ou tabelas inteiras. As transações que contém muitos bloqueios provavelmente levarão um tempo considerável para serem concluídas e exigirão um nível de isolamento maior que os exigidos por aquelas transações que bloqueiam apenas algumas linhas.

A natureza e o nível de importância de uma transação também são muitos significativos na escolha dos níveis de isolamento. O isolamento está relacionado aos estados intermediários observados pelos leitores. Está menos relacionado à corretude da atualização dos dados.

Em alguns cenários, por exemplo, se você precisar de uma estimativa básica das contas inativas do cliente, talvez seja necessário sacrificar a precisão usando um nível de isolamento menor a fim de evitar a interferência nos outros usuários do banco de dados.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre transações e níveis de isolamento, consulte "Transações" no Capítulo 12, “Melhorando o Desempenho do ADO.NET”.
- Para obter mais informações sobre desempenho de bancos de dados, consulte o Capítulo 14, “Melhorando o Desempenho do SQL Server”.

Evite Transações Atômicas de Execução Longa

Mantenha as transações atômicas tão curtas quanto possível para reduzir o tempo de retenção dos bloqueios e de contenção. As transações atômicas executadas por um longo período retém os bloqueios do banco de dados, o que pode reduzir significativamente a taxa de transferência geral do aplicativo. As sugestões a seguir ajudam a reduzir o tempo da transação:

- Evite encapsular operações de somente leitura em uma transação. Para consultar dados de referência (por exemplo, para exibi-los na interface do usuário), o isolamento implícito, fornecido pelo SQL Server para as operações concorrentes, é suficiente para garantir a consistência dos dados.
- Use estratégias otimistas de concorrência. Reúna dados para operações não granuladas fora do escopo da transação e, quando a transação for submetida, forneça dados suficientes para detectar se os dados de referência subjacentes foram alterados o suficiente para serem considerados inválidos. Os métodos comuns incluem comparação de carimbos de data/hora (*timestamps*) para verificar alterações nos dados e a comparação de campos específicos dos dados de referência no banco de dados contra os dados anteriormente lidos.
- Não faça com que as transações cruzem mais fronteiras que o necessário. Reúna dados externos e do usuário antes da transação e defina o seu escopo ao redor de um objeto não granulado ou de uma chamada de serviço.
- Encapsule (*wrap*) apenas as operações que precisam de gerenciadores de recursos transacionais, como SQL Server ou o Enfileiramento de Mensagens do Microsoft Windows.
- Considere o uso de transações de compensação onde você precisa de qualidades transacionais e onde o custo da transação síncrona de longa duração seria muito alto.

Mais Informações

Para obter mais informações sobre concorrência, consulte "Concorrência" no Capítulo 4, "Análise da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Gerenciamento de Recursos

Geralmente, os recursos são limitados e muitas vezes precisam ser compartilhados entre vários clientes. O gerenciamento ineficiente de recursos freqüentemente é a causa dos gargalos no desempenho e na escalabilidade. Por vezes, a plataforma pode fornecer maneiras eficientes para gerenciar recursos, mas você também precisa adotar os padrões corretos de design.

Ao incluir o gerenciamento de recursos no design, considere as seguintes recomendações:

- **Trate as *threads* como um recurso compartilhado.**
- **Agrupe recursos compartilhados ou escassos com o uso de *Pool*.**
- **Demore a adquirir e libere logo.**
- **Considere a criação e a destruição eficientes dos objetos.**
- **Considere a regulagem (*throttling*) de recursos.**

Trate as *Threads* Como um Recurso Compartilhado

Evite criar *threads* por demanda. Se forem criadas *threads* de forma indiscriminada, especialmente para aplicativos num servidor de alto volume, isso poderá prejudicar o desempenho, pois consome recursos (principalmente em servidores com apenas uma CPU) e provoca o excesso de troca de *threads* no processador. Um método melhor é usar um *pool* compartilhado de *threads,* como o *pool* de *threads* do processo. Ao utilizar um *pool* compartilhado, não se esqueça de otimizar a forma de uso das *threads:*

- Otimize o número de *threads* no *pool* compartilhado. Por exemplo, é necessário um ajuste específico do *pool* de *threads* para um aplicativo da Web de alto volume que faça chamadas de saída para um ou mais WebServices. Para obter mais informações sobre como ajustar o *pool* de *threads* nessa situação, consulte o Capítulo 10, "Melhorando o Desempenho dos WebServices".
- Reduza o tamanho do processamento em execução nas *threads* compartilhadas.

Uma implementação eficiente de *pool* de *threads* oferece vários benefícios e permite a otimização dos recursos do sistema. Por exemplo, a implementação do *pool* de *threads* do .NET ajusta, de forma dinâmica, o número de *threads* do *pool* com base nos níveis atuais de uso da CPU. Isso ajuda a impedir a sobrecarga na CPU. O *pool* de *threads* também aplica um limite no número de *threads* ativas simultaneamente em um processo, com base no número de CPUs e em outros fatores.

Agrupe Recursos Compartilhados ou Escassos com o Uso de *Pool*

Agrupe em *pool* os recursos compartilhados que sejam escassos ou custosos para criar como, por exemplo, conexões de banco de dados ou de rede. Use o *pool* para ajudar a reduzir a sobrecarga no desempenho e a melhorar a escalabilidade com o compartilhamento de um número limitado de recursos entre um número bem maior de clientes. Os *pools* comuns incluem:

- ***Pool* de *threads*.** Use *pools* de *threads* do processo em vez de criar *threads* por solicitação.
- ***Pool* de conexões.** Para garantir o uso mais eficiente do *pool* de conexões, use o modelo de subsistema confiável *(trusted subsystem model)* para acessar sistemas e bancos de dados externos. Com esse modelo, você usa uma identidade única e fixa para se conectar aos sistemas externos. Dessa forma, a conexão pode ser consolidada com mais eficiência.
- ***Pool* de objetos.** Os objetos que demoram a inicializar são os candidatos ideais para este *pool.* Por exemplo, você pode usar um *pool* de objetos para reter um conjunto limitado de conexões com o *mainframe* que demoram bastante para serem estabelecidas. Vários objetos podem ser compartilhados por vários clientes, desde que nenhum estado específico do cliente seja mantido. Você também deve evitar qualquer afinidade com um recurso específico. A princípio, criar uma afinidade com um objeto específico neutraliza os benefícios do *pool* de objetos. Qualquer objeto do *pool* deve ser capaz de atender a qualquer solicitação e não deve ser bloqueado por qualquer solicitação específica.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações, consulte o Capítulo 14, "*Building Secure Data Access*", *de Improving Web Application Security: Threats and Countermeasures"* no MSDN, em *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/ThreatCounter.asp.*
- Para obter mais informações sobre *pool* de objetos COM+, consulte *"Pool* de Objetos", no Capítulo 8, "Melhorando o Desempenho dos *Enterprise Services*".

Demore a Adquirir e Libere Logo

Adquira recursos o mais tardiamente possível, imediatamente antes de precisar usá-los, e libere-os imediatamente depois de tê-los utilizado. Use construções de linguagem, como os blocos *finally,* para garantir a liberação de recursos em caso de exceção.

Considere a Criação e a Destruição Eficientes dos Objetos

Geralmente, a criação de objetos deve acontecer apenas no ponto em que seu uso ocorre de fato. Isso garante que os objetos não consumam recursos do sistema enquanto aguardam para ser usados. Libere os objetos imediatamente depois de tê-los utilizado.

Se os objetos exigem código de limpeza explícita e precisam liberar identificadores *(handles)* para recursos do sistema, como arquivos ou conexões de rede, garanta que será executada a limpeza explicita a fim de evitar qualquer vazamento de memória ou desperdício de recursos.

Mais Informações

Para obter mais informações sobre coleta de lixo, consulte o Capítulo 5, “Melhorando o Desempenho do Código Gerenciado”.

Considere a Regulagem *(throttling)* de Recursos

Você pode usar a regulagem de recursos para impedir uma única tarefa de consumir uma porcentagem desproporcional de recursos do total alocado para o aplicativo. A regulagem de recursos impede um aplicativo de ultrapassar o orçamento alocado de recursos do computador, incluindo CPU, memória, E/S do disco e E/S da rede.

Um aplicativo do servidor que tenta consumir grande quantidade de recursos pode provocar um aumento da contenção. Isso provoca o aumento do tempo de resposta e a redução da taxa de transferência. Exemplos comuns de designs ineficientes que podem causar essa degradação incluem:

- A consulta de um usuário que retorna um grande conjunto de resultados de um banco de dados. Isso pode aumentar o consumo de recursos no banco de dados, na rede e no servidor Web.
- Uma atualização que bloqueia um grande número de linhas entre tabelas acessadas com freqüência. Isso causa um aumento significativo na contenção.
- Para ajudar a resolver esses e outros problemas semelhantes, considere as seguintes opções de regulagem de registros:
- Faça a paginação de conjuntos de dados *(result sets*) grandes.
- Estabeleça tempos limite para operações de longa duração a fim de que uma única solicitação não bloqueie um recurso compartilhado além de um limite de tempo permitido.

Defina as prioridades dos processos e das *threads* de forma adequada. Evite atribuir prioridades maiores que o normal, a menos que o processo ou a *thread* seja muito crítica e exija atenção em tempo real do processador.

Se houver situações em que uma única solicitação ou todo um aplicativo precise consumir grande quantidade de recursos, considere a divisão do trabalho entre vários servidores ou a descarga do trabalho em horários que não sejam de pico, quando a utilização de recursos geralmente é baixa.

Mais Informações

Para obter mais informações sobre gerenciamento de recursos, consulte "Gerenciamento de Recursos" no Capítulo 4, "Revisão da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Armazenamento em Cache

O armazenamento em cache é uma das melhores técnicas que você pode usar para melhorar o desempenho. Use o armazenamento em cache para otimizar as pesquisas de dados de referência, evitar o vaivém de solicitações e respostas na rede e evitar o processamento duplicado e desnecessário. Para implementar o armazenamento em cache, você precisa decidir quando carregar os dados no cache. Experimente carregar o cache de forma assíncrona ou usando um processo em lote para evitar atrasos no cliente.

Ao incluir o armazenamento em cache no design, considere as seguintes recomendações:

- **Decida onde armazenar os dados em cache.**
- **Decida quais dados serão armazenados.**
- **Decida a política de expiração e o mecanismo de eliminação.**
- **Decida como carregar os dados no cache.**
- **Evite caches coerentes e distribuídos.**

Decida Onde Armazenar os Dados em Cache

Armazene os estados localmente onde eles possam salvar grande parte do processamento e do vaivém de solicitações e respostas na rede. Isto pode ocorrer no cliente, em um servidor *proxy,* na lógica de negócio, na lógica de apresentação do aplicativo, ou em um banco de dados. Escolha um local de cache que ofereça suporte à vida útil desejada para os seus itens. Se precisar armazenar dados por um período prolongado, use um banco de dados do SQL Server. Para armazenar por um pequeno período, use caches em memória.

Considere as seguintes situações:

- **Cache dos dados na camada de apresentação.** Considere o armazenamento em cache dos dados na camada de apresentação quando for necessário exibí-los para os usuários e quando estes dados não forem armazenados específicos de cada usuário. Por exemplo, se precisar exibir uma lista de estados, você poderá pesquisá-la uma vez no banco de dados e, em seguida, armazená-la no cache.

  Para obter mais informações sobre as técnicas de armazenamento em cache do ASP.NET, consulte o Capítulo 6, “Melhorando o Desempenho do ASP.NET”.

- **Cache dos dados na camada de negócio.** Você pode implementar mecanismos de armazenamento em cache usando tabelas ou outras estruturas de dados na lógica de negócio do aplicativo. Por exemplo, você poderia por em cache as regras de taxação que permitam calcular uma taxa. Considere o armazenamento em cache na camada de negócio quando os dados não puderem ser recuperados do banco de dados de forma eficiente. Os dados alterados com freqüência não devem ser armazenados em cache.

- **Cache de dados no banco de dados.** Armazene no cache do banco de dados quando houver uma grande quantidade de dados e quando precisar armazená-los por um longo período. Os dados podem ser fornecidos em pequenas partes ou de forma integral, dependendo da sua necessidade. Os dados serão armazenados em cache em tabelas temporárias, as quais consomem mais RAM e podem provocar gargalos na memória. É preciso sempre avaliar se o armazenamento em cache em um banco de dados está prejudicando ou melhorando o desempenho do aplicativo.

Decida Quais Dados Serão Armazenados

O armazenamento dos dados certos em cache é o aspecto mais crítico deste assunto. Se não for feito da forma correta, o resultado poderá ser a redução do desempenho, em vez de seu aprimoramento. Você poderá acabar consumindo mais memória e, ao mesmo tempo, sofrer devido a falta de dados no cache *(cachê misses)*, quando os dados não são de fato fornecidos pelo cache, mas buscados de novo da fonte original.

A seguir, algumas recomendações importantes que ajudam a decidir o que armazenar em cache:

- **Evite armazenar dados por usuário em cache.** O armazenamento de dados por usuário em cache pode provocar um gargalo na memória. Imagine um mecanismo de pesquisa que armazene em cache os resultados da consulta acionada por cada usuário, para que ele possa paginar os resultados de forma eficiente. Não armazene dados por usuário em cache a menos que a recuperação destes dados seja trabalhosa e a carga atual dos clientes não aumente a pressão na memória. Mesmo nesse caso, avalie ambos os métodos em relação ao melhor desempenho e considere a possibilidade de armazenar os dados em cache em um servidor dedicado. Nesses casos, você também pode considerar o uso de estado de sessão como um mecanismo de cache para aplicativos da Web, mas apenas para uma pequena quantidade de dados. Além disso, você deve armazenar em cache apenas os dados mais relevantes.
- **Evite armazenar dados voláteis em cache.** Armazene em cache os dados usados com freqüência e não os alterados com freqüência. Armazene em cache dados estáticos que sejam difíceis de recuperar ou de criar.

  É preciso evitar armazenar em cache dados voláteis que precisem ser precisos e atualizados pelos usuários em tempo real. Se você expirar freqüentemente o cache para manter a sincronização com os dados que mudam rapidamente, poderá ter que usar recursos do sistema, como CPU, memória e rede.

  Armazene em cache os dados que não são alterados com freqüência ou que sejam completamente estáticos. Se os dados forem alterados com freqüência, você deverá avaliar o limite de tempo aceitável durante o qual os dados antigos ainda poderão ser fornecidos ao usuário. Por exemplo, considere uma barra de cotações de ações, a qual mostre as cotações das ações. Embora o valor das ações seja continuamente atualizado, a barra de cotações de ações pode ser atualizada com segurança depois de um intervalo de tempo fixo de cinco minutos.

  Você pode então projetar um mecanismo de expiração para limpar o cache e recuperar dados novos da fonte original.
- **Não armazene em cache recursos compartilhados e caros.** Não compartilhe em cache recursos caros, como conexões de rede. Em vez disso, coloque estes recursos num *pool.*
- **Armazene em cache dados já transformados, tendo em mente a sua utilidade.** Se você precisar transformar dados antes de usá-los, transforme-os antes de armazenar em cache.
- **Tente evitar o armazenamento em cache de dados que precisem ser sincronizados entre servidores.** Esse método exige lógica de sincronização manual e complexa e deve ser evitado sempre que possível.

Decida a Diretiva de Expiração e o Mecanismo de Eliminação

Você precisa determinar o intervalo de tempo apropriado para atualização de dados e criar um processo de notificação para indicar que o cache precisa ser atualizado.

Se você armazena dados por bastante tempo, corre o risco de usar dados defasados e, se você expira os dados com freqüência, pode afetar o desempenho. Determine o algoritmo de expiração ideal para o seu cenário. As opções incluem:

- Menos utilizado recentemente.
- Menos freqüentemente utilizado.
- Expiração absoluta depois de um intervalo fixo.
- Expiração do armazenamento em cache com base em uma alteração em uma dependência externa, como um arquivo.
- Limpeza do cache se o limite de um recurso (como um fronteira de memória) for alcançado.

Observação: a melhor opção de mecanismo de eliminação também depende da opção de armazenamento para o cache.

Decida como Carregar os Dados no Cache

Para caches grandes, considere carregar o cache de forma assíncrona via uma *thread* separada ou usando um processo em lote.

Quando um cliente acessa um item de cache já expirado, o cache precisa ser novamente preenchido. Fazer isso de forma síncrona afeta o tempo de resposta do cliente e bloqueia a *thread* de processamento da solicitação.

Evite Caches Coerentes e Distribuídos

Em um *farm* Web, se você precisar manter sincronizados vários caches em diferentes servidores por causa das atualizações locais no cache, provavelmente você estará lidando com um estado transacional. Você deve armazenar esse tipo de estado em um gerenciador de recursos transacionais, como o SQL Server. Caso contrário, você precisará renegociar o nível de integridade e potencial atraso devido a sincronizações que será oferecido em contrapartida à melhora do desempenho e da escalabilidade.

Um cache local é aceitável mesmo em um *farm* de servidores, desde que seja usado apenas para fornecer as páginas de forma mais rápida. Se as solicitações ocorrerem em outros servidores que não tenham o mesmo cache já atualizado, ainda assim eles poderão fornecer as mesmas páginas, embora consultando o meio persistente para obter os mesmos dados.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre armazenamento em cache, consulte "Armazenamento em Cache" no Capítulo 4, "Análise da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e ao Escalabilidade".
- Para obter mais informações e diretrizes sobre armazenamento em cache, consulte o *"Caching Architecture Guide for .NET Framework Applications",* em *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/cachingarch.asp.*
- Para conhecer soluções intermediárias de armazenamento em cache, consulte "*Caching Application Block for .NET*", no MSDN, em *http://msdn.microsoft.com/library/en-us/dnpag/html/CachingBlock.asp*

Gerenciamento de Estados

O gerenciamento de estados inadequado provoca muitos problemas de desempenho e de escalabilidade. As diretrizes a seguir ajudam você a criar um gerenciamento de estados eficiente:

- **Avalie o design *stateful* (que armazena estados) versus o design *stateless* (que não armazena estados).**
- **Considere as opções de armazenamento de estado.**
- **Reduza os dados de sessão.**
- **Libere recursos de sessão assim que possível.**
- **Evite acessar variáveis de sessão na lógica de negócio.**

Avalie o Design *Stateful* versus *Stateless*

Os componentes *stateful* armazenam o estado nas variáveis internas para conclusão de uma atividade lógica que se estende por várias chamadas oriundas do cliente. O estado pode ou não ficar persistente após a conclusão de uma operação. Os componentes *stateful* podem produzir melhor desempenho em determinadas condições de carga. O cliente faz todas as solicitações a uma instância específica do componente para concluir a operação. Por isso, os componentes *stateful* resultam na afinidade dos clientes com o componente.

O cuidado em relação aos componentes *stateful* é que eles podem armazenar no servidor recursos entre chamadas até a atividade lógica ser concluída. Isso pode resultar no aumento da contenção dos recursos. O servidor continua a armazenar recursos mesmo que o cliente não faça chamadas subseqüentes para concluir a operação e as libera somente se houver um valor de tempo limite definido para a atividade.

A afinidade é uma questão importante no design *stateful.* O cliente é associado a uma instância específica por causa do estado localizado. Isso pode ser uma desvantagem, caso você tenha uma camada de aplicativo remoto e tenha metas de escalabilidade que resultem na necessidade de redimensionar a camada do aplicativo. A afinidade associa os clientes a um servidor específico, tornando impossível fazer o balanceamento de carga do aplicativo de forma correta.

Como abordado anteriormente, ter o estado nos componentes é uma questão de design e a decisão exige dados relativos às exigências de implantação (por exemplo, se você tem ou não uma camada de aplicativo remoto) e às suas metas de desempenho e de escalabilidade.

Se você optar por um design de componentes *stateless,* terá então que estabelecer onde persistir o estado fora dos componentes para que ele possa ser recuperado por solicitação.

Considere suas Opções de Armazenamento de Estado

Se você usar um design de componente que não possua estado, armazene o estado em um local onde possa ser recuperado com mais eficiência. Os fatores a serem considerados que influenciam sua escolha incluem a quantidade de estado, a largura de banda de rede entre o cliente e o servidor e se o estado precisa ser compartilhado entre vários servidores. Estão disponíveis as seguintes opções para armazenar o estado:

- **No cliente.** Se possuir pequenas quantidades de estado e largura de banda de rede suficiente, é possível considerar o armazenamento no cliente e enviá-lo de volta ao servidor com cada solicitação.

- **Na memória do servidor.** Caso possua muitos estados a serem transmitidos do cliente à cada solicitação, é possível armazená-lo na memória no servidor, seja em um processo de aplicativo Web ou em um processo local separado. O estado localizado no servidor é mais rápido e evita ciclos completos na rede para obter o estado. No entanto, isso se soma à utilização de memória no servidor.
- **Em um gerenciador de recursos dedicado.** Se tiver grandes quantidades de estado que precisam ser compartilhadas entre vários servidores, considere o uso do SQL Server. O aumento na escalabilidade oferecido por essa abordagem é obtida no custo do desempenho, por causa dos ciclos completos adicionais e da serialização.

Minimize os Dados da Sessão

Mantenha em um mínimo a quantidade de dados de sessão armazenada para um usuário específico, de modo a reduzir as sobrecargas no desempenho do armazenamento e da recuperação. O tamanho total dos dados da sessão para uma carga de destino de usuários concorrentes poderá resultar em mais pressão na memória quando o estado da sessão estiver armazenado no servidor, ou em mais congestionamento na rede se os dados forem mantidos em um armazenamento remoto.

Se você usar estados de sessão, existem duas situações que devem ser evitadas:

- **Evite o armazenamento de qualquer recurso compartilhado.** Estes são exigidos por muitas solicitações e podem causar contenção porque o recurso não é liberado até que o tempo limite da sessão se esgote.
- **Evite o armazenamento de coleções e objetos grandes em armazenamentos de sessão.** Considere o armazenamento em cache se forem exigidos por vários clientes.

Libere os Recursos da Sessão Assim Que Possível

As sessões continuam a prender os recursos do servidor até que os dados sejam explicitamente limpos ou até que o tempo fronteira da sessão se esgote.

Você pode seguir uma estratégia de dois passos para minimizar essa sobrecarga. No momento do design, é necessário certificar-se de que o estado da sessão seja liberado assim que possível. Por exemplo, em um aplicativo Web, é possível armazenar temporariamente um conjunto de dados em uma variável de sessão para que os dados fiquem disponíveis entre páginas. Esses dados devem ser removidos o mais rapidamente possível para reduzir a carga. Uma forma de se obter isso é liberando todas as variáveis de sessão que contém objetos assim que o usuário clicar em um item de menu.

Além disso, é recomendável ajustar o tempo fronteira da sessão para garantir que os respectivos dados não continuarão a consumir recursos por longos períodos.

Evite Acessar as Variáveis da Sessão pela Lógica de Negócio

O acesso às variáveis da sessão pela lógica de negócio faz sentido apenas quando essa lógica é espalhada junto com o código de apresentação como um resultado do forte acoplamento. Talvez isso seja necessário em algumas situações, conforme discutido anteriormente neste capítulo, em "Acoplamento e Coesão".

Entretanto, na maioria dos casos, você se beneficia do acoplamento fraco entre apresentação e lógica de negócio, particionadas em camadas lógicas separadas. Isso oferece melhores opções para a manutenabilidade e a escalabilidade. É mais freqüente que o estado relacionado à interface precise ser persistido durante as chamadas. Por essa razão, o estado relacionado à sessão deve fazer parte da camada de apresentação. Dessa forma, se mudarem os fluxos de trabalho da interface de usuário, somente o código da camada de apresentação será afetado.

Mais Informações

Para obter mais informações sobre o gerenciamento de estados, consulte "Gestão de Estado" no Capítulo 4, "Análise da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Estruturas de Dados e Algoritmos

O uso correto de estruturas de dados e algoritmos representa um papel importante na criação de aplicativos de alto desempenho. Suas escolhas nessas áreas podem afetar significativamente o consumo de memória e a carga da CPU.

As seguintes diretrizes ajudarão a usar de forma eficiente as estruturas de dados e algoritmos:

- **Escolha uma estrutura de dados apropriada.**
- **Pré-determine o tamanho para tipos de dados grandes e que crescem dinamicamente.**
- **Use os tipos de referência e valor de forma apropriada.**

Escolha uma Estrutura de Dados Apropriada

Antes de escolher o tipo de coleção para seus cenários, é recomendável gastar um tempo analisando os requisitos específicos por meio dos seguintes critérios comuns:

- **Armazenamento de dados.** Considere a quantidade de dados que será armazenada. Você irá armazenar alguns registros ou alguns milhares de registros? Você sabe por antecipação qual a quantidade de dados a ser armazenada ou, ao contrário, você só sabe no tempo da execução? Como você precisa armazenar os dados? É necessário armazená-los em ordem ou pode ser de forma aleatória?
- **Tipo.** Quais tipos de dados precisam ser armazenados? São dados fortemente tipados? Você armazena objetos variáveis ou valores tipados?
- **Crescimento.** Como seus dados crescerão? Qual é o tamanho do crescimento? Com qual freqüência?
- **Acesso.** Você precisa de acesso indexado? Você precisa acessar os dados por meio de um par de valores-chave? Você precisa classificar além de buscar?
- **Concorrência.** O acesso aos dados precisa estar sincronizado? Se os dados forem atualizados regularmente, é necessário o acesso sincronizado. Talvez a sincronização não seja necessária se os dados forem somente de leitura.
- ***Marshaling.*** Você precisa empacotar e enviar sua estrutura de dados entre fronteiras? Por exemplo, você precisa armazenar seus dados em cache ou num armazenamento de sessão? Se for necessário, você precisará certificar-se de que a estrutura de dados ofereça suporte à serialização de uma forma eficaz.

Pré-determine o Tamanho para Tipos de Dados Grandes e que Crescem Dinamicamente

Caso saiba que precisa adicionar vários dados em um tipo de dados dinâmico, atribua um tamanho apropriado antecipadamente sempre que for possível. Isso ajuda a evitar realocações desnecessárias de memória.

Use os Tipos Referência e Valor de Forma Apropriada

Os tipos valorados são baseados em pilhas e são transferidos por valor, enquanto as referências são baseados em *heap* e transferidos por referência. Use as seguintes diretrizes ao escolher entre semânticas de transferência-por-valor e transferência-por-referência:

- **Evite transferir por valor tipos valorados grandes para métodos locais.** Se o método de destino estiver no mesmo processo ou domínio do aplicativo, os dados são copiados na pilha. É possível melhorar o desempenho transferindo uma referência a uma grande estrutura por meio de um parâmetro de método, em vez de transferi-la pelo valor.
- **Considere passar por valor os tipos referência que passam pelas fronteiras do processo.** Se você transferir uma referência de um objeto por um fronteira do processo, é necessário um retorno de chamada (*callback*) ao processo cliente cada vez que os campos ou métodos do objeto forem acessados. Ao transferir o objeto por valor, você evita essa sobrecarga. Se transferir um conjunto de objetos ou de objetos conectados, certifique-se de que todos possam ser transferidos por valor.

Considere transferir uma referência quando o tamanho do objeto for muito grande ou o estado for relevante somente dentro das fronteiras atuais do processo. Por exemplo, objetos que mantém identificadores *(handles)* para recursos do servidor local, tais como arquivos.

Mais Informações

Para obter mais informações sobre estruturas de dados e algoritmos, consulte "Estruturas de Dados e Algoritmos" no Capítulo 4, "Revisão da Arquitetura e do Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade".

Resumo das Diretrizes de Design

A Tabela 3.2 resume as diretrizes de design discutidas neste capítulo e organiza-as por categorias de perfil de desempenho.

Tabela 3.2: Diretrizes de Design por Categoria de Perfil de Desempenho

Categoria de perfil de desempenho	Diretrizes
Acoplamento e Coesão	Inclua o fraco acoplamento no design. Inclua a coesão alta no design. Particione a funcionalidade do aplicativo em camadas lógicas. Use *early binding* onde possível. Avalie a afinidade de recursos.
Comunicação	Escolha o mecanismo apropriado de comunicação remota. Crie interfaces concisas. Considere a forma de transferência de dados entre camadas. Reduza a quantidade de dados enviados pela rede. Agrupe o trabalho em lotes para reduzir chamadas pela rede. Reduza as transições entre fronteiras. Considere a comunicação assíncrona. Considere o enfileiramento de mensagens. Considere um modelo de invocação „ *fire and forget* ‰ (dispare e esqueça).

Categoria de perfil de desempenho	Diretrizes
Concorrência	Reduza a contenção diminuindo o tempo de bloqueio. Faça o balanço entre bloqueios granulados e não-granulados. Escolha o nível apropriado de isolamento de transação. Evite transações atômicas de execução longa.
Gerenciamento de Recursos	Trate as *threads* como um recurso compartilhado. Ponha em *pool* os recursos compartilhados ou escassos. Demore a adquirir e libere logo. Considere a criação e a destruição eficientes dos objetos. Considere a regulagem *(throttling)* de recursos.
Armazenamento em Cache	Decida onde armazenar os dados em cache. Decida quais dados serão armazenados. Decida a política de expiração e o mecanismo de eliminação. Decida como carregar os dados no cache. Evite caches coerentes distribuídos.
Gerenciamento de Estados	Avalie o design *stateful* (que armazena status) versus o design *stateless* (que não armazena estado). Considere as opções de armazenamento de estado. Reduza os dados de sessão. Libere recursos de sessão assim que possível. Evite acessar variáveis de sessão na lógica de negócio.
Estruturas de Dados/Algoritmos	Escolha uma estrutura de dados apropriada. Pré-determine o tamanho para de tipos de dados grandes e que crescem dinamicamente. Use os tipos referência e valor de forma apropriada.

Considerações sobre Aplicativos de Desktop

Os aplicativos de desktop devem compartilhar recursos, incluindo CPU, memória, E/S de rede e de disco, com outros processos que são executados no computador. Um aplicativo que consome uma quantidade desproporcional de recursos afeta aos outros aplicativos, como também ao desempenho geral e ao poder de resposta do computador. Alguns dos aspectos mais importantes a serem considerados ao criar aplicativos de desktop incluem:

- **Considere o tempo de resposta da interface de usuário.** O tempo de resposta da interface de usuário é uma consideração importante para aplicativos de desktop. Você deveria considerar executar de forma assíncrona as tarefas de longa duração, delegando-as a uma *thread* em separado, em vez de fazer com que a *thread* principal da interface com o usuário execute todo o trabalho. Isso mantém o poder de resposta da interface de usuário. É possível realizar de várias maneiras o trabalho de forma assíncrona, como usando o *pool* de *threads* do processo, *threads* criadas por demanda ou usando filas de mensagens. No entanto, o processamento assíncrono adiciona complexidade e exige design e implementação cuidadosos.

- **Considere as prioridades do trabalho.** Ao criar aplicativos de desktop complexos, é necessário considerar as prioridades relativas de itens de trabalho dentro do aplicativo e relativas a outros aplicativos que o usuário possa estar executando. Tarefas em segundo plano *(background),* com prioridades baixas e interfaces que não bloqueiam as ações do usuário oferecem melhor desempenho para o usuário (tanto a real quanto a percebida) ao executar diferentes tarefas. Transferências de rede em *background* e carregamento progressivo de dados são duas técnicas que podem ser usadas para priorizar diferentes itens de trabalho

Considerações sobre o Cliente no Navegador

As seguintes diretrizes de design ajudarão a melhorar o desempenho real e detectado para clientes que usam navegador *(browser):*

- **Force o usuário a introduzir critérios de busca detalhados**. Ao validar que o usuário introduziu critérios de busca detalhados, é possível executar consultas mais específicas que resultam em um número menor de dados sendo recuperados. Isso ajuda a reduzir ciclos de dados completos ao servidor e reduz o volume de dados que precisa ser manipulado no servidor e no cliente.
- **Implemente a validação do cliente.** Execute a validação no cliente para evitar que dados inválidos sejam enviados ao servidor, minimizar ciclos completos desnecessários. Por motivos de segurança, sempre valide também os dados no servidor, além de usar a validação no cliente, pois a validação no cliente pode ser facilmente ignorada.
- **Exiba a barra de progresso para operações de longa duração.** Quando existem operações de longa duração que não podem ser evitadas, implemente uma barra de progresso no cliente para melhorar o desempenho percebido. Para obter mais informações sobre como efetuar chamadas de longa duração de um aplicativo ASP.NET, consulte "*How To*: Enviar e Pesquisar para Tarefas de Longa Duração" na seção *"How To"* deste guia.
- **Evite páginas complexas.** Páginas complexas podem resultar em várias chamadas do servidor à lógica de negócio e podem resultar em grandes quantidades de dados sendo transferidos pela rede. Considere as restrições de largura de banda ao criar páginas complexas e que contiverem gráficos.
- **Processe a saída em estágios.** É possível processar a saída de uma página Web por partes. A parte superior ou inferior das páginas da Web de um site é geralmente a mesma para todas as solicitações e pode ser exibida instantaneamente para cada solicitação. É possível gerar um fluxo destas partes específicas depois de ter concluído o processamento da solicitação. Mesmo nesses casos, a exibição do texto pode anteceder o fluxo de imagens.
- **Minimize o tamanho e o número de imagens.** Use imagens pequenas e compactadas e mantenha o número de imagens ao mínimo para reduzir a quantidade de dados que precisam ser enviados ao navegador. Os formatos GIF e JPEG usam a compactação, mas o formato GIF geralmente produz arquivos menores ao compactar imagens que relativamente possuem menos cores. O JPEG geralmente produz arquivos menores quando as imagens contêm cores.

Considerações sobre a Camada da Web

As seguintes recomendações aplicam-se ao design da camada da Web do aplicativo:

- **Considere as implicações da gestão do estado.** A gestão do estado da camada da Web envolve o armazenamento de estado (como as preferências do cliente) pelas várias chamadas para a duração de uma sessão de usuário, ou algumas vezes por várias sessões de usuário. É possível avaliar sua abordagem de gestão do estado usando os seguintes critérios: Qual quantidade de dados e quantos ciclos?

  Considere as seguintes opções:

  - **Armazene os dados no cliente.** É possível armazenar dados no cliente e enviá-los com cada solicitação. Se você armazenar dados no cliente, será necessário considerar as

restrições de largura de banda porque o estado adicional que precisa ser persistido pelas chamadas adiciona-se ao tamanho geral da página. É recomendável armazenar somente pequenas quantidades de dados no cliente de forma que o efeito no tempo de resposta para a largura de banda de destino seja mínimo, dada a carga representativa dos usuários concorrentes.

- **Armazene os dados no processo do servidor.** É possível armazenar dados por usuário no processo *host* do servidor. Se você escolher armazenar dados do usuário no servidor, lembre-se de que os dados consomem recursos no servidor até que a sessão seja encerrada. Se o usuário abandonar a sessão sem emitir qualquer notificação para o aplicativo Web, os dados continuarão a consumir desnecessariamente os recursos do servidor até que a sessão atinja seu tempo limite. O armazenamento de dados de usuário no processo do servidor também introduz a afinidade quanto ao servidor. Isso limita o escalabilidade do aplicativo e geralmente evita o balanceamento do tráfego de rede.
- **Armazene os dados no processo do servidor remoto.** É possível armazenar dados específicos de cada usuário em um armazenamento de estado remoto. O armazenamento de dados no servidor remoto introduz a sobrecarga de desempenho adicional. Isso inclui a latência na rede e o tempo de serialização. Qualquer dado armazenado deve ser serializável e é necessário criá-lo antecipadamente para minimizar o número de ciclos necessários para coletar os dados. A opção de armazenamento remoto permite escalar horizontalmente sua solução, por exemplo, usando vários servidores Web em uma Web *farm.* Um armazenamento remoto redimensionável, tolerante a falhas, como o banco de dados do SQL Server também melhora a capacidade de recuperação do aplicativo.

- **Considere como você cria a saída.** A saída Web pode ser em HTML, XML, texto, imagem ou em algum outro tipo de arquivo. As atividades necessárias para processar a saída incluem a recuperação, a formatação dos dados e envio dos dados ao cliente. Qualquer ineficiência nesse processo afetará todos os usuários do sistema.

  Alguns princípios básicos que ajudam a criar a saída de forma eficiente incluem o seguinte:

  - Evite intercalar a interface de usuário e a lógica de negócio.
  - Retenha os recursos do servidor, como memória, CPU, *threads* e conexões ao banco de dados pelo menor tempo possível.
  - Minimize a concatenação da saída e a sobrecarga do fluxo reciclando os *buffers* usados para essa finalidade.

- **Implemente a paginação.** Implemente o mecanismo de paginação de dados para páginas que exibem grandes quantidades de dados, como telas de resultados de busca. Para obter mais informações sobre como implementar a paginação de dados, consulte "*How To:* Páginar Registros nos Aplicativos .NET" na seção *"How To"* deste guia.
- **Minimize ou evite chamadas com bloqueio.** Chamadas que bloqueiam seu aplicativo Web resultam em solicitações colocadas em fila e possivelmente rejeitadas e que geralmente causam problemas de desempenho e escalabilidade. Minimize ou evite chamadas com bloqueio usando as chamadas à métodos assíncronos, no modelo de invocação "dispensar e esquecer" ou via o envio e enfileiramento de mensagens.
- **Mantenha os objetos o mais próximo possível.** Objetos que se comunicam pelos processos ou entre as fronteiras de máquinas incorrem numa sobrecarga de comunicação significativamente maior do que a acontece entre objetos locais. Escolha a localidade adequada para seus objetos, com base em suas necessidades de confiabilidade, desempenho e escalabilidade.

Considerações sobre a Camada de Negócio

Considere as seguintes diretrizes de design para ajudar a melhorar o desempenho e a escalabilidade de sua camada de negócio:

- **Instrumente seu código antecipadamente.** Instrumente seu aplicativo para reunir dados customizados relativos à integridade e desempenho, que possam ajudar a rastrear se os objetivos de desempenho estão sendo atingidos. A instrumentação também pode fornecer informações adicionais sobre a utilização de recursos associada às operações mais essenciais, executadas com freqüência pelo aplicativo.

  Crie sua instrumentação de forma que possa ser habilitada e desabilitada por meios das definições no arquivo de configuração. Ao fazer isso, será possível minimizar a sobrecarga por meio da habilitação somente dos contadores mais relevantes para a monitoração específica.

- **Prefira um design sem estados.** Ao seguir a abordagem do design sem estado para sua lógica de negócio, você ajuda a minimizar a utilização de recursos em sua camada de negócio e garante que os objetos de negócio não prendem os recursos compartilhados pelas chamadas. Isso ajuda a reduzir a contenção de recursos e aumentar o desempenho. Os objetos sem estado também garantem que você não introduzirá a afinidade com um servidor - o que restringe as opções de escalabilidade horizontal.

  De forma ideal, com um design sem estado, o tempo de vida dos objetos de negócio está ligado ao tempo de vida de uma única solicitação. Se você usar objetos *singleton,* é recomendável armazenar este estado fora do objeto em um gerenciador de recursos, como o banco de dados do SQL Server, e reidratar o objeto com o seu estado antes de atender, à cada solicitação.Observe que um design sem estado pode não ser um requisito se você precisar trabalhar apenas com um único servidor. Nesse caso, os componentes com estado podem realmente ajudar a melhorar o desempenho por meio da remoção da sobrecarga de armazenamento do estado fora dos componentes, ou por fazer com que os clientes enviem o estado necessário para atender à solicitação.

- **Particione sua lógica.** Evite intercalar sua lógica de negócio com sua lógica de apresentação ou de acesso a dados. Isso reduz significativamente a capacidade de manutenção de seu aplicativo e introduz problemas de versionamento. A lógica intercalada geralmente resulta em um sistema fortemente acoplado, com código difícil de otimizar e ajustar por partes.

- **Libere recursos compartilhados assim que possível.** É essencial para o escalabilidade que você libere recursos compartilhados e limitados, como conexões de bancos de dados, assim que terminar de usá-los. Também é necessário garantir que isso ocorra mesmo se exceções forem geradas.

Considerações sobre a Camada de Acesso a Dados

Considere as seguintes diretrizes de design para ajudar a melhorar o desempenho e o escalabilidade de sua camada de acesso a dados:

- **Considere a abstração versus o desempenho.** Se seu aplicativo usar um mesmo banco de dados, use o provedor de acesso específico deste banco. Se você precisar oferecer suporte a vários bancos de dados, geralmente será necessário ter uma camada de abstração, que o ajudará a conectar-se de forma transparente ao banco configurado no momento. As informações relacionadas ao banco de dados e ao provedor de acesso geralmente são especificadas em um arquivo de configuração. Embora essa abordagem seja muito flexível, ela pode gerar sobrecarga no desempenho caso não seja projetada adequadamente.

- **Considere a regulagem de recursos.** Em determinadas situações, é possível que uma mesma solicitação consuma um nível desproporcional de recursos do servidor. Por exemplo, uma consulta que se estende a um grande número de tabelas pode estressar o servidor do banco de dados. Uma solicitação que bloqueia um grande número de linhas de uma tabela usada com freqüência também causa problemas de contenção. Esse tipo de situação afeta outras solicitações, a taxa de transferência geral do sistema e os tempos de resposta.
- **Considere introduzir proteções e padrões de design para evitar esse tipo de problema.** Por exemplo, implemente técnicas de paginação a páginas por meio de uma grande quantia de dados em partes pequenas em vez de ler o conjunto completo de dados de uma só vez. Aplique a normalização de banco de dados apropriada e certifique-se de que somente bloqueará um intervalo pequeno das linhas relevantes.
- **Considere as identidades que fluem ao banco de dados.** Se a identidade usada para operar no banco de dados for a do usuário da chamada original, a conexão não poderá ser reutilizada por outros usuários, porque a solicitação de conexão ao banco de dados é autorizada com base na identidade do chamador. À menos que você tenha um requisito específico, possuindo uma ampla audiência de usuários confiáveis e não-confiáveis, é recomendável fazer todas as solicitações ao banco de dados usando uma mesma identidade. As chamadas com a mesma identidade melhoram a escalabilidade por meio da habilitação de um *pool* de conexão eficiente.
- **Separe as solicitações *somente de leitura* das transacionais.** Evite intercalar as solicitações *somente de leitura* dentro de uma transação. Isso tende a aumentar a duração da transação, o que aumenta os tempos de bloqueio e a contenção. Separe e conclua qualquer solicitação *somente de leitura* antes de iniciar alguma transação que exija seus dados como entrada.
- **Evite retornos de dados desnecessários.** Evite devolver dados desnecessariamente nas operações do banco de dados. O servidor do banco de dados devolve o controle com mais rapidez ao chamador quando usa operações que não retornam dados. É recomendável analisar suas *stored procedures* e operações de "escrita" no banco de dados para minimizar a devolução de dados que o aplicativo não necessita, como contadores de linhas, identificadores e códigos de retorno.

Resumo

Este capítulo mostrou um conjunto de princípios e padrões de design para ajudá-lo a projetar aplicativos capazes de atingir seus objetivos de desempenho e escalabilidade.

O design para desempenho e escalabilidade envolve compensações. Outros atributos de qualidade de serviço, incluindo disponibilidade, capacidade de manutenção, integridade e segurança, também devem ser considerados e balanceados com relação aos seus objetivos de desempenho. Certifique-se de que tenha uma idéia clara sobre quais são seus objetivos de desempenho (incluindo restrições de recursos) durante a fase de design.

Para obter mais informações sobre diretrizes de design específicas de tecnologia, consulte a seção "Considerações sobre Design" em cada um dos capítulos na Parte III, "Desempenho e Escalabilidade de Aplicativos".

Recursos Adicionais

Para obter mais informações, consulte os seguintes recursos:

- Para obter uma lista de verificação para imprimir, consulte "Lista de Verificação: Revisão de Arquitetura e Design para Desempenho e Escalabilidade" na seção "Listas de Verificação" deste guia.
- Para uma abordagem orientada por perguntas destinada à revisão da arquitetura e do design pela perspectiva de desempenho, consulte o Capítulo 4, "Revisão de Arquitetura e Design de Aplicativos .NET para Desempenho e Escalabilidade".
- Para uma abordagem orientada por perguntas destinada à revisão de códigos e da implementação pela perspectiva de desempenho, consulte o Capítulo 13, "Revisão de Código: Desempenho do Aplicativo .NET".
- Para obter informações sobre como avaliar se a arquitetura de software atenderá aos objetivos de desempenho, consulte *PASA: An Architectural Approach to Fixing Software Performance Problems,* por Lloyd G. Williams e Connie U. Smith, no site *http://www.perfeng.com/papers/pasafix.pdf.*
- Para obter informações sobre a arquitetura de aplicativos, consulte *"Application Architecture for .NET: Designing Applications and Services",* no MSDN, pelo endereço *http://msdn.microsoft.com /library/default.asp?url=/library/en-us/dnbda/html/distapp.asp.*
- Para obter mais informações sobre padrões, consulte *"Enterprise Solution Patterns Using Microsoft .NET",* no MSDN, pelo endereço *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnpatterns/html/ESP.asp.*
- Para obter mais informações sobre segurança, consulte "*Building Secure ASP.NET Applications: Authentication, Authorization and Secure Communication"*, no MSDN, pelo endereço *http://msdn.microsoft.com/library /default.asp?url=/library/en-us/dnnetsec/html /secnetIpMSDN.asp* e I*mproving Web Application Security: Threats and Countermeasures,* no MSDN, pelo endereço *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnnetsec/html/ThreatCounter.asp*.

Capítulo

4

Revisão da Arquitetura e Design de um Aplicativo .NET em Relação ao Desempenho e à Escalabilidade

Objetivos

Analisar e revisar os aspectos de desempenho e escalabilidade da arquitetura e design de aplicativos.

Saber o que procurar e quais as principais perguntas a serem feitas ao revisar a arquitetura e o design de aplicativos novos e existentes.

Visão Geral

As características de desempenho de seus aplicativos são determinadas por sua arquitetura e design. Seus aplicativos devem ser projetados e criados com base em princípios sólidos e práticas recomendadas. Nenhum ajuste fino no seu código pode camuflar as implicações de desempenho resultantes de decisões incorretas de arquitetura ou design.

Este capítulo inicia com a introdução de um processo de alto nível que pode ser seguido para revisões de arquitetura e design. Em seguida, são apresentadas considerações sobre implantação e infra-estrutura, seguidas por um conjunto abrangente de perguntas que podem ser usadas para ajudar a orientar suas revisões do aplicativo. As perguntas para a revisão são apresentadas nas próximas seções e são organizadas segundo a estruturação de desempenho e escalabilidade mencionada no Capítulo 1, "Princípios Básicos de Engenharia de Desempenho".

Como Usar Este Capítulo

Este capítulo apresenta uma série de perguntas que devem ser usadas para ajudar a executar uma revisão completa da arquitetura e do design do aplicativo. Existem várias formas de obter o máximo deste capítulo:

- **Vá direto para os tópicos ou leia do início ao fim.** Os títulos principais deste capítulo ajudarão a localizar os tópicos de seu interesse. Outra opção é ler este capítulo do início ao fim para obter uma apreciação completa dos problemas de design de desempenho e escalabilidade.
- **Integre a revisão de desempenho e escalabilidade ao seu processo de design.** Inicie a revisão o mais breve possível. Conforme o design evolui, revise as alterações e os aperfeiçoamentos usando as perguntas apresentadas neste capítulo.
- **Conheça os princípios fundamentais de desempenho e escalabilidade.** Leia o Capítulo 3 - "Diretrizes de Design para Desempenho do Aplicativo," para saber os princípios importantes que ajudarão a criar aplicativos Web que atendam a seus objetivos de desempenho e escalabilidade. É importante conhecer esses princípios básicos para melhorar os resultados do processo de revisão.
- **Desenvolva sua revisão de desempenho e escalabilidade.** Este capítulo fornece as perguntas que devem ser feitas para melhorar o desempenho e a escalabilidade de seu design. Para concluir o processo, é altamente provável que você precise adicionar perguntas específicas exclusivas de seu aplicativo.
- **Use a lista de verificação fornecida na seção "Listas de Verificação" deste guia.** Use a "Lista de Verificação: Revisão de Arquitetura e Design para Desempenho e Escalabilidade" para visualizar e avaliar rapidamente as diretrizes apresentadas neste capítulo.

Processo de Revisão de Arquitetura e Design

O processo de revisão analisa as implicações de desempenho da arquitetura e do design de seu aplicativo. Caso tenha concluído recentemente o design de seu aplicativo, a documentação do seu design poderá ajudá-lo nesse processo. Independentemente de quão abrangente seja sua documentação de design, é necessário decompor seu aplicativo e identificar os itens principais, incluindo fronteiras, interfaces, fluxos de dados, caches e armazenamentos de dados. Também é necessário conhecer a configuração física para a implantação do aplicativo.

Considere os seguintes aspectos ao revisar a arquitetura e o design de seu aplicativo:

- **Implantação e infra-estrutura.** Revise o design do aplicativo em relação ao ambiente de implantação de destino e a quaisquer restrições associadas que podem ser impostas pela empresa ou pelas políticas institucionais.
- **Estrutura de desempenho e escalabilidade.** Tenha particular atenção às abordagens de design adotadas para as áreas que mais comumente exibem os gargalos para o desempenho. Este guia refere-se a isso coletivamente como a estrutura de desempenho e escalabilidade.
- **Análise camada por camada.** Passe pelas camadas lógicas de seu aplicativo e examine as características de desempenho das várias tecnologias usadas dentro de cada camada. Por exemplo, ASP.NET na camada de apresentação, WebServices, Enterprise Services e Microsoft®.NET Remoting na camada de negócio e o Microsoft SQL Server™ na camada de acesso a dados.

A Figura 4.1 mostra essa abordagem de três aspectos do processo de revisão.

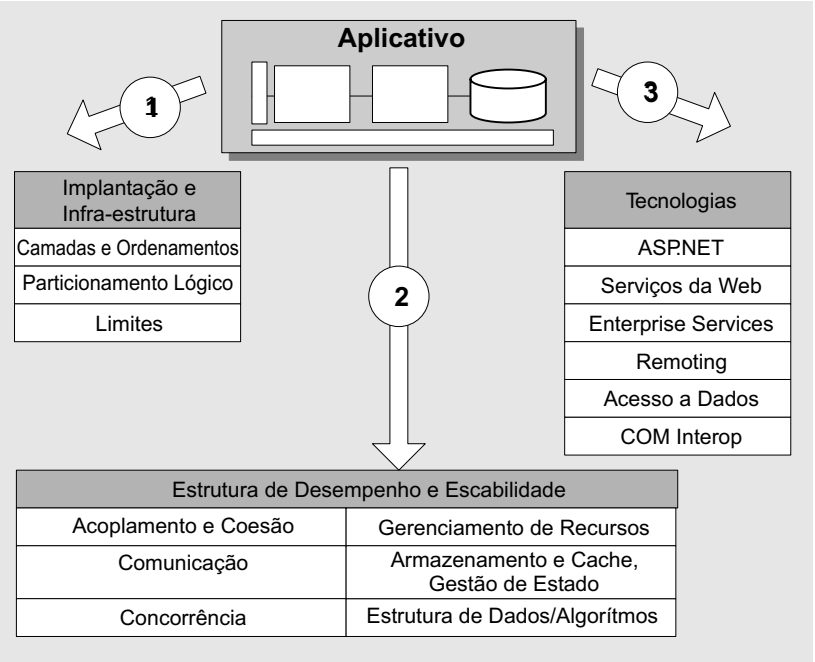

Figura 4.1
O processo de revisão do aplicativo

O restante deste capítulo apresenta as considerações principais e as perguntas a serem feitas durante o processo de revisão para cada uma dessas áreas distintas, exceto as tecnologias. Para obter mais informações sobre as perguntas a serem feitas para cada uma das tecnologias, consulte o Capítulo 13, "Revisão de Código: Desempenho do Aplicativo .NET".

Implantação e Infra-estrutura

Avalie cuidadosamente seu ambiente e quaisquer restrições antes da implantação. Os principais problemas que precisam ser considerados incluem:

- **Você precisa de uma arquitetura distribuída?**
- **Que comunicação distribuída você deveria utilizar?**
- **Você possui interações freqüentes entre fronteiras?**
- **Quais restrições a sua infra-estrutura impõe?**
- **Você considera as restrições na largura de banda da rede?**
- **Você compartilha recursos com outros aplicativos?**
- **O seu design oferece suporte para escalabilidade vertical?**
- **O seu design oferece suporte para escalabilidade horizontal?**

Você Precisa de uma Arquitetura Distribuída?

Se você hospedar sua lógica de negócio em um servidor remoto, será necessário estar ciente das implicações de desempenho importantes das sobrecargas adicionais, como a latência de rede, a serialização de dados e o empacotamento e, com freqüência, as verificações de segurança adicionais. A Figura 4.2 mostra as arquiteturas distribuída e não-distribuída.

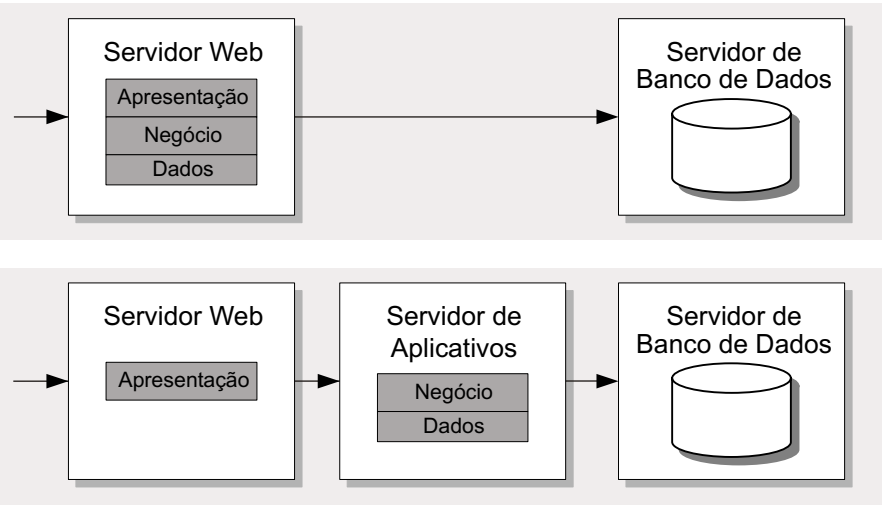

Figura 4.2

Arquiteturas distribuída e não-distribuída

Se você mantiver as camadas lógicas próximas, fisicamente, umas das outras - como, por exemplo, no mesmo servidor ou então no mesmo processo - você minimiza os ciclos de ida e vinda e reduz a latência das chamadas. Se usar uma camada de aplicativo remota, certifique-se de que seu design minimize a sobrecarga de comunicação. Onde for possível, junte as chamadas que representam uma mesma unidade de trabalho, crie serviços pouco granulados, e armazene, quando apropriado, os dados em cache localmente. Para obter mais informações sobre essas diretrizes de design, consulte o Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

A seguir encontram-se alguns exemplos de cenários onde é possível optar por uma camada de aplicativo remota:

- Talvez seja necessário adicionar um *front-end* Web a um conjunto de lógicas de negócio já existente.
- O *front-end* Web e a lógica de negócio poderão ter diferentes necessidades de escalabilidade. Se você necessitar escalar horizontalmente somente a parte da lógica de negócio, e se tanto o *front-end* quanto a lógica de negócio estiverem no mesmo computador, você acabará tendo desnecessariamente várias cópias do *front-end,* o que adiciona sobrecarga de manutenção.
- Talvez seja conveniente compartilhar a lógica de negócio entre vários aplicativos clientes.
- A política de segurança da organização talvez proíba a instalação de lógica de negócio nos servidores *front-end* Web.
- Sua lógica de negócio pode ser computacionalmente intensa, e assim, você pode querer descarregar o processamento em um servidor separado.

Qual Comunicação Distribuída Você Deveria Utilizar?

Serviços são a forma de comunicação preferida entre fronteiras de aplicativos, plataformas, implantação e fronteiras de segurança.

Se você usar os *Enterprise Services*, isto deve acontecer dentro de uma implementação de serviços, ou se tiver problemas de desempenho ao usar os WebServices na comunicação entre processos.

Certifique-se de usar os *Enterprise Services* somente se necessitar do conjunto de recursos adicionais (como *pool* de objetos, transações distribuídas, declarativas, segurança baseada em papéis e componentes enfileirados).

Se usar o .NET Remoting, deverá ser para a comunicação entre domínios *(Domains)* de aplicativos num mesmo processo e não para uma comunicação entre processos ou entre servidores. As outras situações em que poderá necessitar usar o .NET Remoting ocorrerão se você precisar dar suporte aos protocolos customizados. Entretanto, entenda que essa personalização não será portável nas futuras implementações da Microsoft.

Mais Informações

Para obter mais informações, consulte "Orientação Prescritiva para a Escolha entre WebServives, Enterprise Services e .NET Remoting" no Capítulo 11, "Melhorando o Desempenho Remoto".

Você Possui Interações Freqüentes Pelas Fronteiras?

Certifique-se de que o design posicione os componentes, que interagem com freqüência e que executam uma mesma unidade de trabalho, dentro da mesma fronteira ou o mais próximo possível. Os componentes que estão freqüentemente interagindo entre fronteiras podem prejudicar o desempenho devido ao aumento na sobrecarga associado à latência e serialização na chamada. As fronteiras que precisam ser consideradas, da perspectiva do desempenho, são: os domínios do aplicativo *(application domains) ,* compartimentos *(apartments),* processos e servidores. Esses itens são organizados em ordem crescente de custo na sobrecarga.

Quais Restrições a sua Infra-Estrutura Impõe?

Os ambientes de destino são com freqüência definidos rigidamente e o design de seu aplicativo precisa acomodar as restrições impostas. Identifique e avalie qualquer restrição imposta pela infra-estrutura de implantação, como restrições de protocolo e *firewalls.* Considere o seguinte:

- **Você usa *firewalls* internos?**

  Existe algum *firewall* interno entre seu servidor da Web e outros servidores remotos? Isso limita sua escolha de tecnologia no servidor remoto e os protocolos de comunicação relacionados que você pode utilizar. A Figura 4.3 mostra um *firewall* interno.

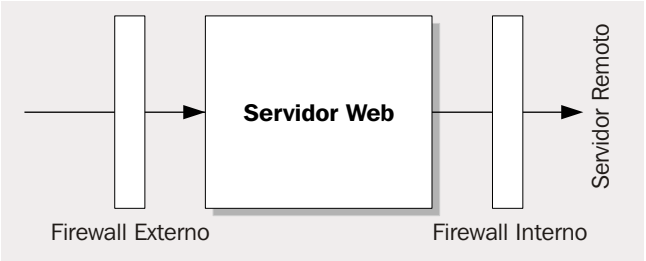

Figura 4.3

Firewalls interno e externo

Se seu servidor remoto hospeda a lógica de negócio e seu *firewall* interno abre somente a porta 80, é possível usar o HTTP e os serviços da Web para a comunicação remota. Isso requer o ISS *(Internet Information Server)* em seu servidor de aplicativos.

Se o servidor remoto executar o SQL Server, será necessário abrir a porta TCP 1433 (ou uma porta alternativa, conforme configurado no SQL Server) no *firewall* interno. Ao usar transações distribuídas envolvendo o banco de dados, também deverá abrir as portas necessárias para o DTC *(Distributed Transaction Coordinator)*. Para obter mais informações sobre como configurar o DCOM para usar um fronteira de portas específico, consulte "*Enterprise Services (COM+) Security Considerations*" no Capítulo 17, "*Securing Your Application Server*", em "*Improving Web Application Security: Threats and Countermeasures*", no MSDN®, disponível no endereço *http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnnetsec/html/ThreatCounter.asp*.

Você usa o protocolo SSL para seu aplicativo ASP .NET?

Se seu aplicativo ASP.NET usa o SSL, considere as seguintes diretrizes:

- Mantenha os tamanhos das páginas o menor possível e minimize o uso de gráficos. Avalie o uso de *View States* de e os *server controls* para suas páginas da Web. Ambas as opções tendem a apresentar um impacto significativo no tamanho da página. Para saber se os tamanhos das páginas são apropriados para sua situação, é recomendável conduzir testes para as larguras de banda que você tem como alvo. Para obter mais informações, consulte o Capítulo 16, "Testando o Desempenho de Aplicativos .NET".
- Use a validação no cliente para reduzir os ciclos completos. Por motivos de segurança, é recomendável usar também a validação no servidor. A validação no cliente é facilmente ignorada.
- Particione suas páginas seguras e não-seguras para evitar a sobrecarga do SSL para páginas anônimas.

Você Considerou as Restrições na Largura de Banda da Rede?

Considere as seguintes perguntas em relação à largura de banda disponível em um ambiente de implantação particular:

- **Você sabe qual é sua largura de banda na rede?**

  Para identificar se existem restrições devido à largura de banda da rede, é necessário avaliar o tamanho médio das solicitações e respostas e multiplicá-lo pela carga simultânea esperada de usuários. O valor total deverá ser consideravelmente inferior do que a largura de banda da rede disponível.

  Se você espera o congestionamento na rede ou que a largura de banda seja um problema, avalie cuidadosamente sua estratégia de comunicação e implemente vários padrões de design para ajudar a reduzir o tráfego de rede efetuando chamadas mais concisas. Por exemplo, proceda da seguinte forma para reduzir os ciclos de ida e volta na rede:

  - Use objetos empacotadores (wrappers) com interfaces pouco granuladas para encapsular e coordenar a funcionalidade de um ou mais objetos de negócio que não foram projetados para acesso remoto eficaz.
  - Agrupe e devolva os dados necessários por meio da devolução de um objeto passado por valor numa única chamada remota.
  - Implemente trabalhos em lote. Por exemplo, é possível colocar em um lote as consultas SQL e executá-las em lote no SQL Server.

  Para obter mais informações, consulte "Minimize a Quantidade de Dados Enviados pela Rede" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

- **Você considerou a largura de banda do cliente?**

  Certifique-se de que você saiba qual é a largura de banda mínima que os clientes provavelmente irão usar para acessar seu aplicativo. Com conexões em largura de banda baixa, a latência na rede responde por grande parte do tempo de resposta do seu aplicativo. As seguintes recomendações de design ajudam a solucionar o problema:

  - Minimize o tamanho de sua página e o uso de gráficos. Meça os tamanhos das páginas e avalie o desempenho usando uma variedade de larguras de banda.
  - Minimize as iterações necessárias para concluir a operação.
  - Minimize o uso do *View State*. Para obter mais informações, consulte "*View State*" no Capítulo 6, "Melhorando o Desempenho do ASP.NET".
  - Use a validação no cliente (em adição à validação no servidor) para ajudar a reduzir os ciclos de ida e vinda.
  - Recupere somente os dados necessários. Se você precisar exibir uma grande quantidade de informações ao usuário, implemente a técnica de paginação de dados.
  - Habilite a compactação HTTP 1.1. Por padrão, o IIS usa os métodos de compactação HTTP GZIP e DEFLATE. Ambos os métodos de compactação são implementados por um filtro ISAPI. Para obter mais informações sobre como habilitar a compactação HTTP, revise a documentação do IIS. É possível localizar mais informações sobre a Especificação de Formato de Arquivo GZIP (RFC 1952) e a Especificação de Formato de Dados Compactados DEFLATE (RFC 1951) no site http://www.ietf.org/.

Você Compartilha Recursos com Outros Aplicativos?

Se seu aplicativo é hospedado por um Provedor de Internet (ISP) ou é executado em outro ambiente hospedado, seu aplicativo compartilha recursos - como processador, memória e espaço em disco - com outros aplicativos. É necessário identificar as restrições de utilização dos recursos. Por exemplo, quanto da CPU seu aplicativo tem permissão para consumir?

O conhecimento sobre as restrições de recursos pode ajudá-lo durante o design inicial e o estágio de protótipos do desenvolvimento de seu aplicativo.

O seu Design Oferece Suporte para Escalabilidade Vertical?

A escalabilidade vertical ocorre por meio da adição de recursos, como processadores e RAM, a seus servidores existentes para que suportem uma capacidade maior. Embora esse tipo de escalabilidade seja, geralmente, mais simples que a escalabilidade horizontal, ela também possui suas armadilhas. Por exemplo, você pode falhar por não tirar proveito de múltiplas CPUs.

O seu Design Oferece Suporte para Escalabilidade Horizontal?

Você escala horizontalmente ao adicionar mais servidores ou usando soluções de balanceamento de carga e *cluster* para distribuir a carga de trabalho. Essa abordagem também fornece proteção contra algumas falhas no hardware, pois, se algum servidor parar de funcionar, outro o assume. Uma estratégia comum de escalabilidade é iniciar pela escalabilidade vertical e então passar para a escalabilidade horizontal, se for necessário.

Para oferecer suporte à estratégia de escalabilidade horizontal, é necessário evitar certas armadilhas no design do seu aplicativo. Para ajudar a garantir que o aplicativo possa ser escalado horizontalmente, revise as seguintes perguntas:

- **O seu design usa camadas lógicas?**
- **O seu design considera o impacto da afinidade de recursos?**
- **O seu design oferece suporte para o balanceamento de carga?**

O seu Design Usa Camadas Lógicas?

Use as camadas lógicas, como as camadas de apresentação, aplicativo e banco de dados, para agrupar componentes que são relacionados e mantêm interação freqüente. Você deve esforçar-se para obter um particionamento lógico e um design em que as interfaces agem como um contrato entre as camadas. Isso facilita a realocação da funcionalidade; por exemplo, se precisar realocar a lógica de negócio computacionalmente intensa para outro servidor. A falha em aplicar a divisão em camadas lógicas resulta em aplicativos monolíticos que são difíceis de manter, aperfeiçoar e redimensionar. A manutenção e o aperfeiçoamento são problemáticos, pois fica difícil medir o efeito da alteração de um componente nos componentes restantes do seu aplicativo.

Observação: A divisão em camadas lógicas não indica necessariamente que você terá o particionamento físico e várias camadas físicas ao implantar seu aplicativo.

O seu Design Considera o Impacto da Afinidade de Recursos?

A afinidade de recursos indica que a lógica do aplicativo é altamente dependente de um recurso particular para a conclusão bem-sucedida de alguma operação. Os recursos podem variar desde recursos de hardware, como CPU, memória, disco ou rede, até outras dependências, como conexões ao banco de dados e a serviços da Web.

O seu Design Oferece Suporte para o Balanceamento de Carga?

O balanceamento de carga é uma técnica fundamental para a maioria dos aplicativos Web voltados para a Internet. Ao criar algum aplicativo ASP.NET, existem várias opções. É possível usar o balanceamento do tráfego de rede para dividir o tráfego entre vários servidores em uma Web *farm*. Também é possível usar o balanceamento de carga na camada do aplicativo por meio do balanceamento de carga de componentes COM+ ou do balanceamento do tráfego de rede, por exemplo, se você usar o .NET Remoting via canal HTTP. Considere o seguinte:

- **Você considerou o impacto da afinidade do servidor com relação aos objetivos de escalabilidade?**

  Os designs que mais comumente causam afinidade do servidor são aqueles que associam o estado da sessão ou os caches de dados a algum servidor específico. A afinidade pode melhorar o desempenho em determinadas situações de escalabilidade vertical. No entanto, isso limita a eficiência para a escalabilidade horizontal. Ainda é possível fazer a escalabilidade horizontal usando sessões de aderência *(sticky sessions)* de forma que cada cliente continue voltando ao mesmo servidor ao qual estava conectado anteriormente. A limitação é que, em vez de "por solicitação", o balanceamento de carga trabalhará em uma base "por cliente"; o que é menos eficaz em algumas situações. Evite a afinidade com o servidor por meio do design de mecanismos apropriados armazenamento de estado e cache.

- **Você usa o estado de sessão no processo?**

  O estado no processo *(in-process state)* é armazenado no processo do servidor *host* da Web. Estado fora do processo *(out-of-process states)* move o armazenamento para um recurso compartilhado dedicado que pode ser compartilhado entre vários processos e servidores.

  Não é possível necessariamente mudar de um estado *in-process* para um estado *out-of-process* simplesmente alterando a configuração do arquivo *Web.config*. Por exemplo, seu aplicativo pode armazenar objetos que não podem ser serializados. Você precisa projetar e planejar para obter a escalabilidade horizontal considerando o impacto do ciclos de ida e vinda, bem como fazer com que todos os tipos sejam armazenados numa sessão serializável. Para obter mais informações, consulte "Gestão de Estado" posteriormente neste capítulo.

- **Você usa o cache de leitura-escrita?**

  O cache de leitura-escrita é um cache no servidor que é atualizado com dados de entrada do usuário. Esse cache deve funcionar somente como meio para retornar páginas Web mais rapidamente e não deve ser necessário para processar com sucesso as solicitações. Para obter mais informações, consulte "Armazenamento em Cache" posteriormente neste capítulo.

Acoplamento e Coesão

O *acoplamento* refere-se ao número e ao tipo de ligações (no design ou no momento da execução) que existem entre as partes do sistema. A *coesão* mede como vários componentes diferentes tiram proveito do processamento e dados compartilhados. O objetivo do design deve ser a garantia de que seu aplicativo será construído numa forma modular e que conterá um conjunto de componentes altamente coesos agrupados flexivelmente.

Os problemas de agrupamento e coesão que precisam ser considerados encontram-se destacados na Tabela 4.1.

Tabela 4.1: Problemas de Acoplamento e Coesão

Problemas	Implicações
Não usa as camadas lógicas	Mistura lógicas de funcionalidades diferentes (como apresentação e de negócio) sem claras opções de escalabilidade entre as fronteiras dos particionamentos lógicos.
Comunicação baseada em objetos por entre fronteiras	Interfaces com interação intensa, que executam vários ciclos de ida e volta.

Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte "Agrupamento e Coesão" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

Use as seguintes perguntas para avaliar o agrupamento e a coesão no design:

- **O seu design é fracamente acoplado?**
- **Quão coeso é seu design?**
- **Você usa *late binding*?**

O seu Design é Fracamente Acoplado?

O acoplamento fraco ajuda a fornecer a independência de implementação e de versão. Sistemas fortemente acoplados são mais difíceis de manter e redimensionar. As técnicas que encorajam o acoplamento fraco incluem o seguinte:

- **Programação baseada em interface.** As interfaces definem os métodos que encapsulam a complexidade da lógica de negócio.
- **Complexidade.** Os dados enviados em uma mesma chamada pelo cliente devem ser suficientes para concluir a operação lógica; como resultado disso, não será necessário persistir o estado entre as chamadas.

Quão Coeso é seu Design?

Revise seu design para garantir que as entidades relacionadas logicamente, como classes e métodos, estejam agrupadas apropriadamente. Por exemplo, verifique se suas classes contêm um conjunto de métodos relacionados logicamente. Verifique se os conjuntos contêm classes relacionadas logicamente. Coesões fracas podem acarretar no aumento dos ciclos de ida e vinda, pois as classes não estão agrupadas logicamente e poderão acabar residindo em camadas físicas diferentes.

Designs não-coesos freqüentemente requerem uma mistura de chamadas locais e remotas para concluir uma operação. Isso pode ser evitado se os métodos relacionados logicamente forem mantidos próximos e não exigirem uma seqüência complexa de interação entre vários componentes. Considere as seguintes diretrizes para alta coesão:

- Particione seu aplicativo em camadas lógicas.
- Organize os componentes de forma que as classes que contribuem para a execução de uma operação lógica particular sejam mantidas em conjunto no componente.
- Certifique-se de que as interfaces públicas expostas por um objeto executem uma única operação coerente nos dados que pertencem ao objeto.

Você usa *Late Binding*?

Revise seu design para garantir que, se usar o *late binding*, você o fará pelos motivos adequados e onde realmente for necessário. Por exemplo, pode ser apropriado carregar um objeto com base nas informações de configuração mantidas em um arquivo de configuração. Outro exemplo, a camada de acesso a dados, agnóstica a bancos de dados, pode carregar diferentes objetos, dependendo do banco de dados configurado atualmente.

Caso você utilize o *late binding*, esteja ciente das implicações no desempenho. A ligação posterior usa internamente a reflexão, que deve ser evitada num código crítico de desempenho. O *late binding* impede a identificação do tipo até o tempo de execução e exige processamento extra. Alguns exemplos de *late binding* incluem o uso do *Activator.CreateInstance* para carregar uma biblioteca no momento da execução, ou o uso do ***Type.InvokeMember*** para invocar um método na classe.

Comunicação

O aumento da comunicação entre fronteiras diminui o desempenho. Problemas comuns de design da comunicação incluem a escolha inapropriada de mecanismos de transporte, protocolos ou formatadores, o uso maior do que o necessário de chamadas e a transferência de mais dados entre fronteiras remotas do que é realmente necessário. Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte "Comunicação" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

Os principais problemas de comunicação que precisam ser considerados encontram-se destacados na Tabela 4.2.

Tabela 4.2: Problemas de Comunicação

Problemas	Implicações
Interfaces que permitem Conversação intensa	Requer vários ciclos de ida e vinda para executar uma única operação.
Enviar mais dados do que o necessário	Aumenta a sobrecarga da serialização e a latência da rede por enviar mais dados do que o necessário,
Ignora os custos de Fronteiras	Custos de fronteira incluem verificações de segurança, comutações de *threads* e serialização.

Para avaliar a eficiência de sua comunicação, revise as seguintes perguntas:

- **Você usa interfaces que exigem muitas interações?**
- **Você efetua chamadas remotas?**
- **Como você troca dados com um servidor remoto?**
- **Você possui requisitos de comunicação seguros?**
- **Você usa filas de mensagens?**
- **Você efetua chamadas de longa duração?**
- **Você pode usar *applications domains* em vez de processos?**

Você Usa Interfaces que Exigem Muitas Interações?

Interfaces que ocasionam interações intensas requerem várias idas e voltas para executar uma única operação. Elas resultam em um aumento na sobrecarga de processamento, nas autenticações e autorizações adicionais, no aumento da sobrecarga devido à serialização e em maior latência na rede. O custo exato depende do tipo de fronteira que a chamada tem que cruzar e da quantidade e tipo de dados transferidos na chamada.

Para ajudar a reduzir a interação intensa das suas interfaces, empacote os componentes com um objeto que implementa uma interface concisa. É esse objeto empacotador que coordena os objetos de negócio. Isso encapsula toda a complexidade da camada da lógica de negócio e expõe um conjunto de métodos agregados que ajudam a reduzir as idas e vindas. Aplique essa abordagem também ao COM *Interop* em adição às chamadas de métodos remotos.

Você Efetua Chamadas Remotas?

Várias chamadas remotas incorrem em utilização da rede, bem como mais sobrecarga de processamento. Considere as seguintes diretrizes para ajudar a reduzir as idas e vindas:

- Para aplicativos ASP.NET, use a validação no cliente para reduzir as idas e vindas ao servidor. Por motivos de segurança, também utilize a validação no servidor.
- Implemente o armazenamento em cache no cliente para reduzir as idas e vindas. Como o cliente está armazenando os dados em cache, o servidor precisa considerar a implementação da validação de dados antes de iniciar alguma transação com o cliente. O servidor pode, então, validar se o cliente está trabalhando com a versão obsoleta dos dados, que pode ser inadequada para esse tipo de transação.
- Coloque seu trabalho em lotes para reduzir idas e vindas. Quando você coloca seu trabalho em lotes, é possível que tenha que lidar com o sucesso ou falha parcial. Se você não projetou para isso, talvez aconteça um lote muito grande, que pode resultar em travamentos ou pode fazer com que um lote completo seja rejeitado devido a uma das partes estar fora de ordem.

Como Você Troca Dados com um Servidor Remoto?

O envio de dados pelas conexões incorre em custos da serialização e também no uso da rede. A serialização ineficaz e o envio de mais dados do que o necessário são as causas comuns de problemas de desempenho. Considere o seguinte:

- **Você usa o .NET Remoting?**

  Se você usar o .NET Remoting, o ***BinaryFormatter*** reduzirá o tamanho dos dados enviados pela conexão. O ***BinaryFormatter*** cria um formato binário menor em comparação ao formato *SOAP* criado pelo ***SoapFormatter.***

  Use o atributo ***[NonSerialized]*** para marcar qualquer membro de dados privado ou público que não queira serializar.

  Para obter mais informações, consulte o Capítulo 11, "Melhorando o Desempenho Remoto".

- **Você usa *DataSets* do ADO.NET?**

  Se você serializar ***DataSets*** ou outros objetos ADO.NET, avalie com cuidado se realmente é necessário enviá-los pela rede. Esteja ciente de que estão serializados como XML mesmo se usar o ***BinaryFormatter.***

  Considere as seguintes opções de design se precisar transferir objetos ADO.NET pela conexão em aplicativos onde é crítico o desempenho:

  - Considere a implementação da serialização personalizada de forma que seja possível serializar os objetos ADO.NET por meio do formato binário. Isso é particularmente importante quando o desempenho é crítico e o tamanho dos objetos transferidos é grande.
  - Considere usar a classe *DataSetSurrogate* para a serialização binária de *DataSets.*

  Para obter mais informações, consulte o Capítulo 12, "Melhorando o Desempenho do ADO.NET".

- **Você usa WebServices?**

  Os WebServices usam o ***XmlSerializer*** para serializar dados. O XML é transmitido como texto puro, que é maior que uma representação binária. Avalie com atenção os parâmetros e o tamanho da carga dos parâmetros do WebService. Certifique-se de que o tamanho médio da carga de solicitação e resposta, multiplicado pelo número esperado de usuários concorrentes esteja dentro de suas limitações de largura de banda de rede.

  Certifique-se de marcar qualquer membro público que não precise ser serializado com o atributo ***[XmlIgnore]***. Existem outras considerações de design que ajudam a reduzir o tamanho dos dados transmitidos pela conexão:

  - Prefira o design de estilo de mensagens centrado em dados para seus serviços da Web. Com essa abordagem, a mensagem age como um contrato de dados entre o WebServices e seus clientes. A mensagem contém todas as informações necessárias para concluir a operação lógica.
  - Use o formato de codificação de documento/literal para seus WebServices devido ao tamanho da carga ser significativamente reduzido em comparação aos formatos documento/codificado ou RPC/codificado.
  - Se você precisar transferir anexos binários, considere o uso da codificação Base64 ou, se usar os WSE (Aprimoramentos nos Serviços da Web) no cliente e no servidor, considere usar o DIME (Encapsulamento Direto de Mensagens da Internet). O cliente também pode possuir uma implementação diferente da WSE que ofereça suporte ao formato DIME. Para obter mais informações, consulte "*Using Web Services Enhancements to Send SOAP Messages with Attachments*", no MSDN, pelo endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/wsedime.asp.*

Para obter mais informações sobre os serviços da Web, consulte o Capítulo 10, "Melhorando o Desempenho dos Serviços da Web".

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre ***DataSetSurrogate,*** consulte o artigo 829740 do *Knowledge Base*, "*Improving DataSet Serialization and Remoting Performance*", no endereço *http://support.microsoft.com/default.aspx? scid=kb;en-us;829740.*
- Para obter mais informações sobre como medir a sobrecarga de serialização, consulte o Capítulo 15, "Medindo o Desempenho do Aplicativo .NET".
- Para obter mais informações sobre como melhorar o desempenho da serialização, consulte "Procedimento: Melhorar o Desempenho da Serialização" na seção “Procedimentos” deste guia.

Você Possui Requisitos de Comunicação Seguros?

Se for importante garantir a confidencialidade e integridade de seus dados, será necessário usar técnicas de *hash* de chave e criptografia; ambas apresentam um impacto inevitável no desempenho. No entanto, é possível minimizar a sobrecarga de desempenho usando os algoritmos e tamanhos de chave corretos. Considere o seguinte:

- **Você usa o algoritmo de criptografia e o tamanho de chave corretos?**

  Dependendo da confidencialidade dos dados e da quantidade de segurança necessária, é possível utilizar técnicas que variam desde simples soluções de codificação até criptografia forte. Se você usar a criptografia, onde for possível (quando ambas as partes são conhecidas com antecedência), use a criptografia simétrica em vez da assimétrica. A criptografia assimétrica fornece segurança melhorada, mas causa um impacto muito maior no desempenho. Uma abordagem comum é usar a assimétrica somente para trocar uma chave secreta e então usar novamente a criptografia simétrica.

Mais Informações

Para obter mais informações, consulte "*Cryptography*" no Capítulo 7, "*Building Secure Assemblies*", de I*mproving Web Application Security: Threats and Countermeasures,* no MSDN, pelo endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec /html/ThreatCounter.asp.*

Você Usa Filas de Mensagens?

O uso de filas de mensagens permite enfileirar o trabalho para algum componente sem bloquear para obter os resultados. Filas de mensagens são particularmente úteis para desagrupar componentes *front-end* e *back-end* em um sistema e para melhorar a sua confiabilidade. Quando o processamento estiver concluído, o servidor pode enviar os resultados de volta à fila de mensagens do cliente, em que cada mensagem pode ser identificada e reconciliada com o ID único da mensagem. Se necessário, é possível utilizar um processo de segundo plano *(background)* dedicado no cliente, para processar as respostas das mensagens.

Para usar um modelo de programação baseado em componente com enfileiramento de mensagem, considere a utilização dos *Enterprise Services Queue Components.*

Você Efetua Chamadas de Longa Duração?

Uma chamada de longa duração pode ser qualquer tipo de trabalho que demora a ser concluído. Longo é um termo relativo. Geralmente, as chamadas de longa duração resultam de chamadas a WebServices, servidor de banco de dados remoto ou algum componente remoto. No caso de algum aplicativo de servidor, as chamadas de longa duração podem bloquear o trabalho, as *threads* de E/S, ou ambos, dependendo da lógica de implementação.

Os seguintes designs ajudam a evitar o impacto de bloqueio com chamadas de longa duração:

- Use filas de mensagens. Se o cliente exigir um indicador de sucesso ou resultados dos processos no servidor, use uma fila no cliente.
- Se o cliente não precisar de quaisquer dados do servidor, considere o atributo ***[OneWay]***. Com esse modelo "dispensar e esquecer", o cliente emite a chamada e então continua sem esperar pelos resultados.
- Se o cliente fizer uma chamada de longa duração e não puder continuar sem os resultados, considere a invocação assíncrona. No caso de aplicativos do servidor, a invocação assíncrona permite à *thread* de trabalho invocar a chamada para continuar e executar processamentos adicionais antes de recuperar os resultados. Na maioria dos casos, se os resultados não estiverem disponíveis neste momento, a *thread* de trabalho bloqueia até que os resultados sejam devolvidos.

Mais Informações

Para obter mais informações sobre como manipular chamadas de longa duração do ASP.NET, consulte "Procedimento: Enviar e Pesquisar Tarefas de Longa Duração" na seção “Procedimentos” deste guia.

Para obter mais informações sobre como usar o atributo ***One Way*** com os serviços da Web, consulte "Comunicação Unidirecional (Dispensar e Esquecer)" no Capítulo 10, "Melhorando o Desempenho dos Serviços da Web".

Você Pode Usar Application Domains em vez de Processos?

A comunicação entre *Application Domains* é consideravelmente mais rápida do que a comunicação entre processos. Entre os cenários em que vários *Application Domains* podem ser apropriados temos:

- Seu aplicativo gera uma cópia de si mesmo com freqüência.
- Seu aplicativo passa muito tempo na comunicação entre processos com programas locais que trabalham exclusivamente com seu aplicativo.
- Seu aplicativo abre e fecha outros programas para realizar seu trabalho.

Embora a comunicação de domínios de aplicativos cruzados seja muito mais rápida do que a comunicação entre processos, o custo de iniciar e fechar aplicativos pode ser muito maior. Existem outras limitações: por exemplo, um erro fatal em um *Application Domain* pode destruir todo o processo e pode haver alguma limitação de recursos quando todos os aplicativos do domínio compartilham o mesmo espaço limitado de memória virtual do processo.

Concorrência

Use as perguntas desta seção para avaliar a forma como seu design minimiza a contenção e maximiza a concorrência.

Os principais problemas de concorrência que precisam ser considerados encontram-se destacados na Tabela 4.3.

Tabela 4.3: Problemas de Concorrência

Problemas	Implicações
Bloqueio das chamadas	Trava o aplicativo e reduz o tempo de resposta e a taxa de transferência.
Bloqueios não-granulares *threads* mal utilizadas	Trava o aplicativo e leva ao enfileiramento de solicitações e a *time outs* Sobrecarga adicional do processador e da memória devido à mudança do contexto e sobrecarga do gerenciamento de *threads*
Mantém bloqueios mais tempo do que o necessário	Causa aumento na contenção e concorrência reduzida.
Níveis de isolamento inapropriados	A escolha insatisfatória dos níveis de isolamento resulta em contenção, longo tempo de espera, *timeouts* e travamentos.

Para avaliar os problemas de concorrência, revise as seguintes perguntas:

- **Você precisa executar tarefas simultaneamente?**
- **Você cria *threads* por solicitação?**
- **Você faz o design de tipos seguros para *threads* por *default?***
- **Você usa bloqueios granulados?**
- **Você adquire o mais tarde e libera o mais cedo?**
- **Você usa a primitiva de sincronização apropriada?**
- **Você usa o nível de isolamento da transação apropriado?**
- **O seu design considera a execução assíncrona?**

Você Precisa Executar Tarefas Simultaneamente?

A execução simultânea tende a ser mais adequada para tarefas que são independentes umas das outras. Você não se beneficia da implementação assíncrona se o trabalho for intensivo em CPU (especialmente para servidores de um único processador) em vez de ser intensivo em E/S. Se o trabalho for intensivo em CPU, uma implementação assíncrona resulta no aumento da utilização e na comutação de *threads* num processador já ocupado. Isso provavelmente prejudica o desempenho e a taxa de transferência.

Considere usar uma invocação assíncrona quando o cliente puder executar tarefas paralelas que sejam intensivos em E/S como parte da sua unidade de trabalho. Por exemplo, é possível utilizar uma chamada assíncrona a um WebService para liberar a *thread* de execução, de modo a poder realizar alguma tarefa paralela antes de bloquear para a chamada e espera dos resultados do WebService.

Você Cria *Threads* por Solicitação?

Revise o design e certifique-se de usar o *pool* de *thread.* O uso do *pool* de *thread* aumenta a probabilidade de que o processador encontre uma *thread* em um estado pronto para execução (para processamento), que resulta em um maior paralelismo entre as *threads.*

As *threads* são recursos compartilhados e caros de inicializar e gerenciar. Se você criar *threads* a cada solicitação feita ao aplicativo no servidor, isso afetará a escalabilidade devido ao aumento da probabilidade de bloqueios das *threads* e causará problemas de desempenho. Isto ocorre devido ao aumento na sobrecarga da criação das *threads,* da comutação do contexto do processador e da coleção de lixo.

Você Faz o Design tipos seguros para *threads* por *default?*

Evite gerar tipos seguros para *threads (thread safe)* por *default* . A implementação de tipos seguros para *threads* insere uma camada adicional de complexidade e sobrecarga aos tipos, o que freqüentemente é desnecessário se os problemas de sincronização forem lidados por uma camada de software de nível superior.

Você Usa Bloqueios Granulados?

Avalie a compensação entre ter bloqueios grosseiros e refinados. Os bloqueios refinados garantem a execução atômica de uma pequena quantia de código. Quando utilizados adequadamente, fornecem mais concorrência por meio da redução da contenção de bloqueio. Quando usado nos locais incorretos, os bloqueios refinados poderão adicionar complexidade e diminuir o desempenho e a concorrência.

Você Adquire o Mais Tarde e Libera o Mais Cedo?

A aquisição tardia e a liberação antecipada de recursos é a chave para reduzir a contenção. Você bloqueia um recurso compartilhado ao bloquear todos os caminhos de códigos que acessam o recurso. Certifique-se de minimizar a duração em que você prende e bloqueia recursos nesses caminhos de código, pois a maioria dos recursos tende a ser compartilhada e limitada. Se você liberar mais rápido o bloqueio, mais cedo o recurso estará disponível para outras *threads.*

A abordagem correta é determinar a granularidade ideal de bloqueio para sua situação:

- **Sincronização no nível do método.** É apropriado sincronizar no nível do método quando tudo que o método faz é agir no recurso que precisa de acesso sincronizado.
- **Acesso sincronizado somente na parte relevante do código.** Se o método precisa validar parâmetros e executar outras operações, além de acessar o recurso que requer acesso serializado, considere o bloqueio somente das linhas de código relevantes que acessam o recurso. Isso ajuda a reduzir a contenção e a melhorar a concorrência.

Você Usa a Primitiva de Sincronização Apropriada?

O uso da primitiva de sincronização apropriada ajuda a reduzir a contenção sobre os recursos. Pode haver situações em que você precise sinalizar outras *threads* em espera, de forma manual ou automática, com base no acionamento de algum evento. Outras situações variam pela freqüência das atualizações de leitura e escritas. Algumas dessas políticas que ajudam a escolher a primitiva de sincronização apropriada para sua situação são:

- Use ***Mutex*** para comunicação entre processos.
- Use ***AutoResetEvent*** e ***ManualResetEvent*** para sinalização de eventos.
- Use ***System.Threading.InterLocked*** para incrementos e decrementos sincronizados sobre tipos inteiros e longos.
- Use ***ReaderWriterLock*** para várias leituras concorrentes. Quando ocorre a operação de escrita, ela se torna exclusiva pois todas as outras threads de leitura e escrita ficam enfileiradas.
- Use ***lock*** quando desejar permitir que apenas um leitor ou gravador aja no objeto por vez.

Você Usa o Nível de Isolamento da Transação Apropriado?

Ao considerar unidades de trabalho (tamanho de transações), é necessário pensar sobre qual nível de isolamento deve ser usado e qual bloqueio será necessário para fornecer este nível de isolamento e, por conseqüência, qual será o risco real de travamento ou inconsistências devido ao uso deste nível. É necessário selecionar os níveis de isolamento apropriados de suas transações para garantir que a integridade dos dados seja preservada sem afetar indevidamente o desempenho do aplicativo.

A seleção de um nível de isolamento superior ao necessário indica que você está bloqueando objetos no banco de dados por períodos maiores e aumentando a contenção sobre esses objetos. A seleção de um nível de isolamento muito baixo aumenta a probabilidade de perda da integridade dos dados devido a leituras ou gravações sujas.

Se não tiver certeza do nível de isolamento correto para seu banco de dados, é recomendável usar a implementação padrão, que foi criada para funcionar adequadamente na maioria das situações.

Observação: É possível diminuir seletivamente o nível de isolamento usado em consultas específicas, em vez de alterar todo o banco de dados.

Para obter mais informações, consulte o Capítulo 14, "Melhorando o Desempenho do SQL Server".

O seu Design Considera a Execução Assíncrona?

A execução assíncrona do trabalho permite à *thread* do processamento principal descarregar o trabalho em outras *threads,* tornando possível a sua continuação para efetuar, se for exigido, algum processamento adicional antes de recuperar os resultados da chamada assíncrona.

Cenários que exigem o trabalho de E/S intensivos, como operações sobre arquivos e chamadas a WebServices, costumam ser de longa duração e podem bloquear a E/S ou as *threads* de trabalho, dependendo da lógica de implementação usada para concluir a operação. Ao considerar a execução assíncrona, avalie as seguintes perguntas:

- **Você está criando um aplicativo Windows *Forms?***

  Os aplicativos do Windows *Forms* que executam chamadas de E/S, como uma chamada a um WebService ou uma operação E/S sobre um arquivo, geralmente devem usar a execução assíncrona para manter o poder de resposta da interface do usuário. O .NET Framework oferece suporte para operações assíncronas em todas as classes relacionadas a atividades de E/S, com exceção do ADO.NET.

- **Você está criando um aplicativo no servidor?**

  Aplicativos no servidor devem usar a execução assíncrona sempre que o trabalho for intensivo quanto à E/S, como nas chamadas a WebServices onde o aplicativo possa executar algum trabalho útil enquanto a *thread* de trabalho em execução estiver liberada.

  É possível liberar a *thread* de trabalho completamente, submetendo trabalho e pesquisando seus resultados no cliente em intervalos regulares. Para obter mais informações sobre como fazê-lo, consulte "*How To:* Enviar e Pesquisar Tarefas de Longa Duração" na seção *"How To"* deste guia.

  Outras abordagens incluem a liberação parcial da *thread* de trabalho para executar algum trabalho útil antes de bloquear para receber os resultados. Essa abordagem usa derivados do *Mutex,* tais como *WaitHandle.*

Para aplicativos no servidor, não é recomendável chamar o banco de dados de forma assíncrona, pois o ADO.NET não possui suporte para tais operações e exige o uso de delegações *(delegates)*, que são executados nas *threads* de trabalho. Também é possível bloquear a *thread* original em vez de usar outra *thread* de trabalho para concluir a operação.

- **Você usa o padrão de design assíncrono?**

  O .NET Framework fornece um padrão de design para a comunicação assíncrona. A vantagem disso é que é o chamador quem decide se uma chamada em particular deverá ser assíncrona. Não é necessário que o receptor exponha código específico para a invocação assíncrona. Outras vantagens incluem o uso de tipos fortes.

  Para obter mais informações, consulte "*Asynchronous Design Pattern Overview*" no *.NET Framework Developer's Guide*, disponível no MSDN pelo endereço: *http://msdn.microsoft.com/library/default.asp? url=/library/en-us/cpguide/html /cpconasynchronousdesignpatternoverview.asp.*

Mais Informações

Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte "Concorrência" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

Gerenciamento de Recursos

Problemas comuns de gestão de recursos incluem a falha em liberar e operar no pool de recursos no tempo correto, e na falha em usar o armazenamento em cache - o que leva ao acesso em excesso de recursos. Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte "Gestão de Recursos" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

Os principais problemas de gestão de recursos que precisam ser considerados encontram-se destacados na Tabela 4.4.

Tabela 4.4: Problemas de Gestão de Recursos

Problemas	Implicações
Não faz *pool* dos recursos dispendiosos	Pode resultar na criação de várias instâncias dos recursos juntamente com sua sobrecarga de conexão. Aumento no custo da sobrecarga e afeta o tempo de resposta do aplicativo.
Prende os recursos compartilhados	Não libera os recursos compartilhados, como conexões (ou atrasa a liberação deles), levando ao esgotamento de recursos no servidor, limitando a escalabilidade.
Acessa ou atualiza grandes quantidades de dados	A recuperação de grandes quantidades de dados do recurso aumenta o tempo gasto para atender à solicitação, bem como a latência de rede. Isso deve ser evitado, especialmente em acesso de largura de banda baixa, pois afeta o tempo de resposta. O aumento no tempo gasto no servidor também afeta o tempo de resposta conforme aumenta o número de usuários concorrentes.
Não limpa adequadamente	Leva à carência de recursos e ao aumento no consumo de memória. Ambos os problemas afetam a escalabilidade.
Falha ao considerar como regular *(throttling)* dos recursos	Um grande número de usuários pode causar a falta de recursos e sobrecarregar o servidor.

Para avaliar de forma eficiente a gestão de recursos do seu aplicativo, revise as seguintes perguntas:

- **O seu design acomoda *pool*?**
- **Você adquire o mais tarde e libera o mais cedo?**

O seu Design Acomoda *Pool*?

Identifique os recursos que incorrem em inicialização demorada e certifique-se de usar o *pool,* quando for possível, para compartilhá-los de forma eficiente entre vários clientes. Os recursos adequados para pool incluem *thread*s, conexões de rede, *buffer*s de E/S e objetos.

Como uma diretriz geral, crie e inicialize *pool*s na inicialização do aplicativo. Certifique-se de que seu código no cliente libere o objeto do *pool* assim que terminar de usar o recurso. Considere o seguinte:

- **Você usa o Enterprise Services?**

  Considere o *pool* de objetos para objetos customizados que são caros de criar. O *pool* de objetos permite configurar e melhorar os tamanhos máximo e mínimo do *pool* de objetos. Para obter mais informações, consulte "*Pool* de Objetos" no Capítulo 8, "Melhorando o Desempenho dos Enterprise Services".

- **Você trata *threads* como recursos compartilhados?**

  Use o *pool* de *thread*s do .NET, quando for possível, em vez de criar *thread*s por demanda. Por padrão, o *pool* de *thread* é auto-ajustável e deve mudar suas opções default somente se você tiver requisitos específicos. Para obter mais informações sobre quando e como configurar o *pool* do *thread*, consulte "Explicação sobre Segmentação" no Capítulo 6, "Melhorando o Desempenho do ASP.NET" e "Segmentação" no Capítulo 10, "Melhorando o Desempenho dos WebServices".

- **Você usa o *pool* de conexão para bancos de dados?**

  É recomendável conectar-se ao banco de dados usando uma identidade única e confiável. Evite personificar o chamador original e usar essa identidade para acessar o banco de dados. O uso de uma abordagem de *subsistema confiável*, em vez da personificação permite usar o *pool* de conexão de forma eficaz.

Mais Informações

Para obter mais informações sobre o *pool* de conexão, consulte "Conexões" no Capítulo 12, "Melhorando o Desempenho do ADO.NET".

Você Adquire-o Mais Tarde e Libera-o Mais Cedo?

Minimize a duração com que você prende um recurso. Quando você trabalha com algum recurso compartilhado, quanto mais rápido liberá-lo, mais rápido ele estará disponível para outros usuários. Por exemplo, é recomendável bloquear um recurso poucos instantes antes de precisar executar a operação real, em vez de prendê-lo durante o estágio de pré-processamento. Isso ajuda a reduzir a contenção junto aos recursos compartilhados.

Armazenamento em Cache

Avalie a abordagem usada no seu aplicativo para armazenar em cache e identificar onde, quando e como seu aplicativo armazena dados no cache. Revise seu design para observar se perdeu alguma oportunidade de armazenamento. Essa técnica é uma das mais conhecidas para melhorar o desempenho.

Os principais problemas de armazenamento em cache que precisam ser considerados encontram-se destacados na Tabela 4.5.

Tabela 4.5: Problemas de Armazenamento em Cache

Problemas	Implicações
Não usa armazenamento em cache quando pode	Maior número de ciclos de idas e vindas no armazenamentos de dados para cada solicitação do usuário, aumentando a carga de armazenamento de dados.
O cache é atualizado com mais freqüência do que o necessário	Aumento no tempo de resposta ao cliente, desempenho reduzido e aumento no uso dos recursos do servidor.
Armazenando em cache com formatos de dados inapropriados	Aumento no consumo da memória, resultando em redução no desempenho, falhas de procura no cache e maior número de acessos ao armazenamento de dados.
Armazenando em cache dados voláteis ou específicos do usuário	A alteração freqüente de dados requer expiração freqüente do cache, resultando em utilização em excesso da CPU, memória e recursos de rede.
Prende os dados de cache por períodos prolongados	Com as políticas inapropriadas de vencimento ou com Mecanismos de eliminação, seu aplicativo serve dados obsoletos.
Não possui o mecanismo de sincronização em cache no Web *farm*	Isso indica que o cache nos servidores do farm não é o mesmo e pode levar a um comportamento funcional incorreto do aplicativo.

Para avaliar com que eficiência seu aplicativo usa o armazenamento em cache, revise as seguintes perguntas:

- **Você armazena dados em cache?**
- **Você sabe quais dados devem ser armazenados em cache?**
- **Você armazena dados voláteis em cache?**
- **Você escolheu o local correto para o armazenamento em cache?**
- **Qual é sua política de vencimentos?**

Você Armazena Dados em Cache?

Você faz buscas custosas por demanda? Se operar nos dados que são caros para recuperar, computar ou processar, provavelmente serão bons candidatos para o armazenamento em cache. Identifique as áreas do seu aplicativo que poderão beneficiar-se desse tipo de armazenamento.

Você Sabe Quais Dados Devem Ser Armazenados em Cache?

Identifique as oportunidades para armazenamento em cache durante o design do aplicativo. Evite considerar o armazenamento em cache somente em estágios posteriores do ciclo de desenvolvimento como uma medida de emergência para melhorar o desempenho.

Prepare uma lista de dados adequados para o armazenamento nas várias camadas do seu aplicativo. Se você não identificar os dados candidatos para armazenamento antecipadamente, muito provavelmente você irá gerar tráfego redundante e em excesso e irá realizar mais trabalho do que o necessário. Possíveis candidatos para armazenamento em cache incluem:

- **Páginas da Web relativamente estáticas.** É possível armazenar páginas em cache que não mudam com freqüência usando o recurso de cache de saída do ASP.NET. Considere usar os *User Controls* para conter as partes estáticas de uma página. Isso permite beneficiar-se do armazenamento em cache de fragmentos do ASP.NET.
- **Itens específicos de dados de saída.** É possível armazenar em cache, na classe **Cache** do ASP.NET, dados que precisam ser exibidos a usuários.
- **Parâmetros de *stored procedures* e resultados de consultas.** É possível armazenar em cache parâmetros de consulta e resultados de consultas usadas com freqüência. Geralmente, isso é feito na camada de acesso aos dados para reduzir o número de idas e vindas ao banco de dados. O armazenamento parcial em cache ajuda que páginas dinâmicas possam gerar um amplo conjunto de saídas (como menus e controles) a partir de um pequeno conjunto de resultados em cache.

Você Armazena Dados Voláteis em Cache?

Você sabe a freqüência com que os dados são modificados? Use estas informações para decidir se você deve armazenar os dados em cache. Também é recomendável estar ciente de quão desatualizados os dados exibidos podem estar, em relação aos dados originais. Tenha consciência do time out permitido para exibição dos dados obsoletos, mesmo quando os dados forem atualizados no seu local de origem.

De forma ideal, é recomendável armazenar tanto os dados em cache que ficam relativamente estáticos durante um período de tempo, quanto os dados que não precisam de alteração para cada usuário. No entanto, mesmo se seus dados forem voláteis e mudarem, por exemplo, a cada dois minutos, você ainda terá benefícios do armazenamento em cache. Por exemplo, se geralmente você espera receber solicitações de 20 clientes num intervalo de dois minutos, é possível economizar 20 idas e vindas ao servidor por meio do armazenamento dos dados em cache.

Para determinar se o armazenamento de conjuntos particulares de dados será benéfico, avalie o desempenho com e sem o armazenamento em cache.

Você Escolheu o Local Correto para o Armazenamento em Cache?

Certifique-se de armazenar os dados em cache num local que possa economizar a maioria dos processamentos e ciclos de idas e vindas. Também é necessário que seja um local que ofereça suporte ao tempo de vida necessário para os itens em cache. É possível armazenar os dados em cache em várias camadas do seu aplicativo. Revise as seguintes considerações sobre cada camada:

- **Você armazena em cache os dados na camada da apresentação?**

  É recomendável armazenar dados na camada de apresentação que precisem ser exibidos ao usuário. Por exemplo, é possível armazenar as informações que são exibidas num registro de cotação de ações. É recomendável evitar o armazenamento em cache dos dados de cada usuário, a menos que os dados por usuário sejam muito pequenos e que o tamanho total dos dados em cache não exija muita memória. No entanto, se os usuários tendem a ficar ativos por alguns momentos e depois vão embora novamente, o armazenamento em cache de dados por usuário durante curtos períodos pode ser a abordagem apropriada. Isso depende de sua política de armazenamento em cache.

- **Você armazena em cache os dados na camada de negócio?**

  Armazene os dados na camada de negócio se for necessário processar solicitações da camada de apresentação. Por exemplo, é possível armazenar em cache os parâmetros de entrada de *stored procedures* numa coleção.

- **Você armazena em cache os dados do banco de dados?**

  Considere o armazenamento de dados em tabelas temporárias no banco de dados caso precise dele durante períodos demorados. É útil armazenar em cache os dados dos bancos de dados quando for demorado processar as consultas para obter um conjunto de resultados. O conjunto de resultados poderá ter um tamanho grande, portanto será proibitivo enviar os dados através da rede para serem armazenados em outras camadas. Para grandes quantidades de dados, implemente um mecanismo de paginação que permita ao usuário recuperar os dados em cache, uma porção de cada vez. Também será necessário considerar a política de vencimento para os dados quando a fonte for atualizada.

- **Você sabe o formato em que os dados em cache serão usados?**

  Prefira o armazenamento de dados em cache em formato pronto para uso, de forma que você não precise de processamento ou transformações adicionais. Por exemplo, é possível armazenar uma página completa da Web usando o cache de saída. Isso reduz significativamente a sobrecarga de processamento do ASP.NET em seu WebService.

- **Você escreve no cache?**

  Se você escreve no cache as atualizações requisitadas pelo usuário antes de atualizar no banco de dados persistente, você está criando uma afinidade junto ao servidor. Isso pode ser problemático caso seu aplicativo tenha sido implantado num Web *farm,* já que a solicitação de um cliente em particular está ligada a um servidor particular, devido a atualizações locais no cache.

  Para evitar esse problema, é recomendável atualizar dados em cache somente para melhorar o desempenho e não para garantir o processamento correto da solicitação. Dessa forma, as solicitações ainda poderão ser servidas com sucesso por outros servidores no mesmo *cluster* no Web *farm.*

  Considere usar o armazenamento de estado de sessão para atualizações de dados específicos do usuário.

Qual é Sua Política de Expiração?

Uma política de expiração inapropriada pode resultar na invalidação freqüente dos dados em cache, o que vai contra os benefícios deste armazenamento. Considere o seguinte enquanto escolhe o mecanismo de vencimento:

- **Com que freqüência as informações em cache podem estar incorretas?**

  Tenha em mente que cada pedaço de dado em cache já está possivelmente obsoleto. Se você souber a resposta a esta pergunta, isto irá ajudá-lo a escolher entre o algoritmo de expiração absoluto ou não absoluto.

- **Existe alguma dependência cuja alteração invalida os dados em cache?**

  É necessário avaliar os algoritmos baseados em dependências. Por exemplo, a classe Cache do ASP.NET permite o vencimento dos dados caso sejam feitas alterações em algum arquivo em particular. Observe que, na maioria das situações, pode ser aceitável exibir dados que sejam um pouco antigos.

- **O tempo de vida depende da freqüência de utilização dos dados?**

  Se a resposta for sim, é necessário avaliar os algoritmos como o *menos usados recentemente* ou *o usado com menos freqüência*.

- **Você preenche novamente os caches para os dados de frequente alteração?**

  Se seus dados mudam freqüentemente, eles podem ou não ser bons candidatos para armazenamento em cache. Avalie os benefícios de desempenho do armazenamento em cache em relação aos custos de reconstruir os dados do cache. O armazenamento de dados de freqüente alteração pode ser uma idéia excelente se dados levemente obsoletos não forem um problema.

- **Você implementou alguma solução de armazenamento em cache que demora a carregar?**

  Se for necessário manter um grande cache e esse cache demorar a ser criado, considere ouso de uma *thread,* no plano de fundo, que crie o cache ou construa-o incrementalmente ao longo do tempo. Quando o cache atual expirar, será possível trocá-lo pelo atualizado - que foi criado no plano de fundo. Do contrário, haverá bloqueio das solicitações dos clientes enquanto esperam pela atualização do cache.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte "Armazenamento em Cache" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".
- Para obter mais informações e diretrizes sobre o armazenamento em cache, consulte *Caching Architecture Guide for .NET Framework Applications*, no MSDN, pelo endereço *http://msdn.microsoft.com/library/en-us/dnbda/html/CachingArch.asp.*
- Para obter informações sobre soluções em armazenamento em cache de camadas intermediárias, considere o texto *Caching Application Block for .NET,* no MSDN, pelo endereço *http://msdn.microsoft.com/library/en-us/dnpag/html/CachingBlock.asp.*

Gerenciamento de Estados

Os principais problemas de gestão de estado que precisam ser considerados encontram-se destacados na Tabela 4.6.

Tabela 4.6: Problemas de Gestão de Estado

Problemas	Implicações
Componentes com estado	Prendem os recursos do servidor e podem causar afinidade no servidor, o que reduz as opções de escalabilidade.
Uso de armazenamento de estado em memória	Limita a escalabilidade devido à afinidade com o servidor.
Armazenamento de estado no banco de dados ou servidor, quando o cliente for uma escolha melhor	Maior utilização dos recursos do servidor; escalabilidade do servidor limitado.
Armazenar mais estados no servidor, quando o banco de dados é uma melhor escolha	Estados armazenados *in-process* e no processo do servidor da Web limitam a capacidade do aplicativo Web de ser executado num Web farm. Grandes quantidades de estados mantidos na memória também criam pressão de memória no servidor.
Armazenar mais estado do que o necessário	Aumento na utilização dos recursos do servidor e mais tempo para armazenamento de estado e recuperação.
Sessões prolongadas	Valores de *time out* inapropriados resultam em sessões que consomem e prendem os recursos do servidor por mais tempo do que o necessário.

Para obter mais informações sobre as perguntas e problemas que surgiram nesta seção, consulte "Gestão de Estado" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

Para avaliar de forma eficiente a gestão do estado, revise as seguintes perguntas:

- **Você usa componentes sem estado?**
- **Você usa o .NET Remoting?**
- **Você usa WebServices?**
- **Você usa o Enterprise Services?**
- **Você se certificou de que são serializáveis os objetos a serem guardados nos armazenamentos de sessão?**
- **Você depende do *ViewState*?**
- **Você sabe o número de sessões simultâneas e os dados médios de sessão por usuário?**

Você Usa Componentes Sem Estado?

Considere cuidadosamente se precisa de componentes com estado. Um design sem estado geralmente é preferível porque oferece mais opções de escalabilidade. Eis algumas das principais considerações:

- **Quais são os requisitos de escalabilidade para o seu aplicativo?**

  Caso você precisa colocar seus componentes de negócio em um *cluster* remoto da camada intermediária, você precisará ou planejar-se para usar componentes sem estado *(stateless),* ou armazenar os estados em um outro servidor acessível por todos os servidores em seu *cluster* da camada intermediária.

  Caso não possua tais requisitos de escalabilidade, componentes com estado, em certos cenários, ajudam a melhorar o desempenho, pois o estado não precisa ser transmitido pelo cliente através das conexões ou recuperado de um banco de dados remoto.

- **Como você gerencia o estado em componentes sem estado?**

  Se você fizer o design planejando componentes sem estado e precisar abstrair o gerenciamento do estado, será necessário saber os requisitos de tempo de vida e o tamanho dos dados de estado. Se optar por componentes sem estado, algumas opções para gerenciamento são:

  - **Transferir o estado do cliente à cada chamada ao componente.** Esse método é eficiente se não forem necessárias várias chamadas para concluir uma única operação lógica e se a quantidade de dados for relativamente pequena. Isso é ideal se o estado for necessário para processar as solicitações e se for possível descartá-lo após a conclusão do processo.
  - **Armazenamento do estado no banco de dados.** Essa abordagem é apropriada se a operação fizer várias chamadas, tornando a transmissão do estado pelo cliente ineficiente, ou se o estado for acessado por vários clientes, ou em ambos os casos.

Você Usa o .NET Remoting?

O .NET Remoting oferece suporte a SAOs (objetos ativados pelo servidor) e CAOs (objetos ativados pelo cliente). Caso você possua requisitos específicos de escalabilidade e precise planejar para um ambiente com balanceamento de carga, é recomendável dar preferência a SAOs de chamadas únicas (*single call* SAOs). Esses objetos retêm o estado somente durante uma única chamada.

SAOs que são *singletons* podem ser *stateful* ou *stateless* e podem reidratar o estado por vários meios, dependendo dos requisitos. Use essas opções quando precisar fornecer acesso sincronizado a um recurso em particular.

Se seus objetivos de escalabilidade permitirem usar um único servidor, será possível avaliar o uso de CAOs, que são objetos *stateful.* Um objeto ativado pelo cliente só pode ser acessado somente por uma instância particular do cliente que o criou. Portanto, são capazes de armazenar estado entre as chamadas.

Para obter mais informações, consulte "Considerações sobre Design" no Capítulo 11, "Melhorando o Desempenho Remoto".

Você Usa WebServices?

WebServices com estado representam um RPC ou um design de objeto distribuído. Com o design de estilo RPC, uma mesma operação lógica pode realizar várias chamadas. Esse tipo de design freqüentemente aumenta o vai e vem de chamadas e geralmente exige que o estado seja persistido entre várias chamadas.

Uma abordagem baseada em mensagem geralmente é preferível para serviços da Web. Com essa abordagem, a mensagem serve como o contrato de dados entre o cliente e o servidor. O cliente transfere a mensagem ao servidor. Ela contém informações suficientes para completar uma única unidade de trabalho. Geralmente isso não requer que qualquer estado seja persistido pelas chamadas, e como resultado, esse design pode ser facilmente escalado de forma horizontal entre vários servidores.

Para obter mais informações, consulte "Gestão de Estado" no Capítulo 10, "Melhorando o Desempenho dos Serviços da Web".

Você Usa Enterprise Services?

Se você planejar usar o *pool* de objetos do Enterprise Services, será necessário criar componentes sem estado. Isso é necessário porque os objetos precisam ser reciclados entre as várias solicitações de diferentes clientes. O armazenamento de estado para um cliente em particular fará com que o objeto não possa ser compartilhado entre os clientes.

Para obter mais informações, consulte "Gestão de Estado" no Capítulo 8, "Melhorando o Desempenho dos Enterprise Services".

Você se Certificou de que os Objetos a Serem Armazenados em Sessão São Serializáveis?

Para armazenar objetos numa sessão *out-of-process,* tais como o uso de um serviço para armazenar o estado ou o uso do SQL Server, os objetos devem ser serializáveis. Não é necessário serializar objetos para armazenar objetos numa sessão *in-process,* mas é recomendável levar isso em consideração caso precise mover seu estado de sessão para fora do processo.

Habilite uma classe a ser serializada usando o atributo ***Serializable.*** Certifique-se de usar o atributo ***NonSerialized*** para evitar serializações desnecessárias.

Você Depende do *View State*?

Se você usa ou planeja usar *View States* para manter o estado entre as chamadas, é recomendável prototipar e avaliar cuidadosamente o impacto do desempenho. Considere o tamanho total da página e os requisitos de largura de banda para atingir seus objetivos de tempo de resposta.

Grandes quantidades de dados persistentes dos server *controls,* como o ***DataGrid,*** aumentam significativamente o tamanho da página e aumentam o tempo de resposta. Use o rastreamento *(tracing)* do ASP.NET para saber o tamanho exato do estado de exibição para cada controle do servidor ou para uma página completa.

Mais Informações

Para obter mais informações sobre como melhorar a eficiência do estado de exibição, consulte "Estado de Exibição" no Capítulo 6, "Melhorando o Desempenho do ASP.NET".

Você Sabe o Número de Sessões Simultâneas e os Dados Médios de Sessão por Usuário?

Saber o número de sessões simultâneas e a média de dados da sessão por usuário permite decidir a forma de armazenamento da sessão. Se a quantidade total de dados de sessão responde por uma porção significativa da memória alocada no processo de trabalho do ASP.NET, é recomendável considerar o armazenamento fora do processo *(out-of-process).*

O uso do armazenamento de estado fora do processo aumenta as idas e vindas na rede e os custos de serialização, portanto ele precisa ser bem avaliado. O armazenamento de vários objetos customizados no armazenamento de sessão ou o armazenamento de vários valores pequenos aumenta a sobrecarga. Considere a agregação dos valores em um único tipo antes de adicioná-los ao armazenamento de sessão.

Estruturas e Algoritmos de Dados

Os principais problemas de estrutura de dados que precisam ser considerados encontram-se destacados na Tabela 4.7.

Tabela 4.7: Problemas de Estruturas de Dados

Problemas	Implicações
Escolha de uma coleção sem avaliar suas necessidades (tamanho, adição, exclusão, atualização)	Eficiência reduzida; código extremamente complexo.
Uso da coleção incorreta para uma determinada tarefa	Eficiência reduzida; código extremamente complexo.
Conversão de tipo em excesso	Passagem de tipos de valor a uma referência, causando sobrecarga devido ao *boxing* e *unboxing*, resultando em baixo desempenho.
Buscas ineficazes	Varredura completa de todo o conteúdo da estrutura de dados, resultando em baixo desempenho
Sem medição do custo das estruturas de dados e/ou algoritmos na sua situação real	Gargalos não detectados devido ao código ineficiente.

Para obter mais informações, consulte "Estruturas de Dados e Algoritmos" no Capítulo 3, "Diretrizes de Design para Desempenho do Aplicativo".

Considere as seguintes perguntas para avaliar seu design da estrutura de dados e algoritmo:

- **Você usa as estruturas de dados apropriadas?**
- **Você precisa de coleções customizadas?**
- **Você precisa estender com *IEnumerable* as suas coleções customizadas?**

Você Usa as Estruturas de Dados Apropriadas?

A escolha incorreta da estrutura de dados para sua tarefa pode prejudicar o desempenho, porque as estruturas de dados específicas são projetadas e otimizadas para tarefas particulares. Por exemplo, se você precisar armazenar e transferir valores através de uma fronteira física, ao invés de usar uma coleção, é possível usar um *array* simples, que evita a sobrecarga devido ao *boxing*.

Defina claramente seus requisitos para cada estrutura de dados antes de escolhe-la. Por exemplo, é necessário classificar e pesquisar dados ou acessar elementos pelo índice?

Mais Informações

Para obter mais informações sobre como escolher a estrutura de dados apropriada, consulte "Guia para Coleções" no Capítulo 5, " Melhorando o Desempenho do Código Gerenciado" e "*Selecting a Collection Class*" no .NET Framework Developer's Guide, disponível no MSDN em *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html /cpconselectingcollectionclass.asp.*

Você Precisa de Coleções Customizadas?

Na maioria dos cenários as coleções fornecidas pelo .NET Framework são suficientes, embora, ocasionalmente, seja recomendável desenvolver uma coleção customizada. Investigue cuidadosamente as classes de Coleção fornecidas antes de desenvolver a sua própria. Os principais motivos para desenvolver a sua própria coleção customizada são:

- **É necessário empacotar a coleção por referência em vez de por valor, o que é o comportamento padrão das coleções fornecidas pelo .NET Framework.**
- **É necessário ter uma coleção fortemente tipada.**
- **É necessário personalizar o comportamento da serialização de uma coleção.**
- **É necessário melhorar o custo de enumeração.**

Você Precisa Estender com *IEnumerable* as suas Coleções Customizadas?

Se estiver desenvolvendo uma coleção customizada e precisar enumerá-la com freqüência percorrendo a coleção, é recomendável estender a interface *IEnumerable* para minimizar o custo da enumeração.

Para obter mais informações, consulte "Explicações sobre Coleções" no Capítulo 5, "Melhorando o Desempenho do Código Gerenciado".

Acesso a Dados

Os principais problemas de acesso a dados que precisam ser considerados encontram-se destacados na Tabela 4.8.

Tabela 4.8: Problemas de Acesso a Dados

Problemas	Implicações
Design do esquema é insatisfatório	Aumento no processamento do servidor do banco de dados; rendimento reduzido.
Falha ao paginar grandes conjuntos de resultados	Aumento no consumo da largura de banda da rede; tempos de resposta atrasados; aumento na carga do cliente e do servidor.
Exposição de hierarquias de objetos ineficiente quando outras mais simples serviriam	Aumento na sobrecarga da coleção de lixo; aumento no esforço de processamento necessário.
Consultas ineficientes ou busca *(fetch)* de muitos dados	Consultas ineficientes ou a coleta de todos os dados para exibir apenas uma parte deles é um custo desnecessário, em termos de desempenho e recursos do servidor.
Indices insatisfatórios ou estatísticas de índice obsoletas	Cria uma carga desnecessária no servidor do banco de dados.
Falha ao avaliar o custo de processamento em seu servidor de banco de dados e seu aplicativo	Falha em atingir os objetivos de desempenho e excesso das alocações do orçamento.

Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte o Capítulo 12, "Melhorando o Desempenho do ADO.NET". Considere o seguinte:

- **Como você transmite os dados entre as camadas?**
- **Você usa *stored procedures*?**
- **Você processa somente os dados necessários?**
- **Você precisa paginar através dos dados?**
- **Suas transações abrangem vários sistemas de armazenamentos de dados?**
- **Você manipula *BLOBs?***
- **Você está consolidando códigos repetidos para acesso a dados?**

Como Você Transmite os Dados entre as Camadas?

Revise sua abordagem para transferir dados entre as camadas de seu aplicativo. Além do desempenho bruto, as principais considerações são capacidades de uso, manutenção e programação. Considere os seguintes pontos:

- **Você considerou os requisitos do cliente?**

  Concentre-se nos requisitos do cliente e evite transmitir dados em um formato e forçar o cliente a convertê-los para outro. Se o cliente exigir os dados somente para fins de exibição, as coleções simples, como *array*s ou um objeto ***Arraylist,*** são adequadas pois oferecem suporte à ligação de dados *(data binding).*

- **Você transforma os dados?**

  Caso precise transformar dados, evite múltiplas transformações conforme os dados fluem pelo aplicativo.

- **Você pode agrupar dados logicamente?**

  Para agrupamentos lógicos, como os atributos que descrevem um funcionário, considere usar um tipo ***class*** ou ***struct*** customizado, que são eficientes para a serialização. Use o atributo ***NonSerializable*** em qualquer campo que não precise ser serializado.

- **A interoperabilidade entre plataformas é um objetivo do design?**

  Se esse for o caso, é recomendável usar XML, embora precise considerar os problemas de desempenho, incluindo requisitos de memória e o esforço de análise *(parsing)* significativo que é necessário para processar grandes *string*s XML.

- **Você usa objetos *DataSet?***

  Se o cliente precisar ser capaz de para exibir os dados de várias formas, de atualizar os dados no servidor usando a concorrência otimista e de manipular relações complexas entre vários conjuntos de dados, um ***DataSet*** é bastante adequado para atender a esses requisitos. ***DataSets*** são caros de criar e serializar e usam muita memória. Se precisar de um cache desconectado e uma funcionalidade rica com suporte do objeto ***DataSet,*** você pode considerar o ***DataSet*** fortemente tipado, que oferece acesso marginalmente mais rápido aos seus campos.

Você Usa *Stored Procedures*?

O uso de *stored procedure*s é preferível na maioria dos casos. Geralmente eles fornecem desempenho melhor em comparação com as instruções SQL dinâmicas. Do ponto de vista da segurança, é necessário considerar a possibilidade de injeção no SQL e a questão da autorização. Ambas as abordagens (*stored procedures* ou SQL dinâmicos), se forem mal escritos, estão suscetíveis à injeção SQL. A autorização do banco de dados é frequentemente mais fácil de gerenciar com as *stored procedure*s, pois é possível restringir as contas de serviço do aplicativo que podem executar *stored procedures* específicos e impedí-los de acessarem as tabelas diretamente.

Se você usa *stored procedures*, considere o seguinte:

- Tente evitar recompilações. Para obter mais informações sobre como são causadas as recompilações, consulte o artigo 243586 do Microsoft Knowledge Base, "*INF: Troubleshooting Stored Procedure Recompilation*", em *http://support.microsoft.com/default.aspx?scid=kb;en-us;243586*.
- Use a coleção ***Parameters;*** do contrário você ainda estará sujeito à injeção no SQL.
- Evite criar SQL dinâmico dentro da *stored procedure*.
- Evite misturar a lógica de negócio nas suas *stored procedures*.

Se você usa SQL dinâmico, considere o seguinte:

- Use a coleção ***Parameters*** para ajudar a impedir a injeção SQL.
- Coloque as instruções em lotes, se possível.
- Considere a capacidade de manutenção (por exemplo, atualização de arquivos de recursos versus *strings* de comandos no código).

Ao usar *stored procedures*, considere as seguintes diretrizes para maximizar o desempenho:

- Analise seu esquema de BD para ver se ele é adequado para executar as atualizações ou buscas necessárias. O seu esquema oferece suporte para sua unidade de trabalho? Você possui os índices apropriados? As suas consultas tiram proveito do design do esquema?
- Observe seus planos de execução e custos. A E/S lógica é com freqüência um excelente indicador do custo geral de consulta no servidor carregado.
- Quando for possível, use os parâmetros de saída em vez de devolver um conjunto de resultados que contém linhas únicas. Isso evita a sobrecarga de desempenho associada à criação do conjunto de resultados no servidor.
- Avalie o procedimento armazenado para garantir que não haja recompilações freqüentes para vários caminhos de código. Em vez de possuir várias instruções *if else* para sua *stored procedures*, considere dividí-las em várias *stored procedures* pequenas e chamá-las de uma mesma *stored procedures*.

Você Processa Somente os Dados Necessários?

Revise seu design para garantir que não recuperou mais dados (colunas ou linhas) do que o necessário. Identifique as oportunidades para paginar registros de modo a reduzir o tráfego na rede e a carga do servidor. Ao atualizar registros, certifique-se de atualizar apenas as alterações em vez do conjunto completo de dados.

Você Precisa Paginar Através dos Dados?

A paginação através dos dados requer a transmissão de dados do banco à camada de apresentação e a exibição ao usuário. A paginação por meio de um grande número de registros pode ser dispendiosa se enviar mais dados pelas conexões do que o necessário, o que pode adicionar custos de rede, memória e processamento nas camadas de apresentação e banco de dados. Considere as seguintes diretrizes para desenvolver uma solução para paginar pelos registros:

- Se os dados não forem muito grandes e precisarem ser servidos a vários clientes, considere enviá-los numa mesma iteração e armazene-os em cache no cliente. É possível paginar pelos dados sem efetuar ciclos completos ao servidor. Certifique-se de usar uma política de vencimento de dados apropriada.
- Se os dados a serem mostrados forem baseados na entrada do usuário e forem grandes, considere o envio somente das linhas mais relevantes ao cliente, para cada tamanho de página. Use a instrução *SELECT TOP* e o tipo de dado *TABLE* em suas consultas SQL para desenvolver esse tipo de solução.

- Se os dados a serem mostrados consistirem de um grande r*esult set* e forem os mesmos para todos os usuários, considere usar tabelas temporárias globais para criar e armazenar os dados em cache uma única vez, e então enviar as linhas relevantes para cada cliente, conforme a necessidade. Essa abordagem é útil se você precisar executar consultas de longa duração que se espalham sobre várias tabelas para criar o *result set*. Se você precisar coletar os dados de uma mesma tabela apenas, as vantagens da tabela temporária serão minimizadas.

Mais Informações

Para obter mais informações, consulte "Procedimento: Registrar Páginas em Aplicativos .NET" na seção “Procedimentos” deste guia.

Suas Transações Abrangem Vários Sistemas de Armazenamentos de Dados?

Caso você possua transações que se espalham sobre vários armazenamentos de dados, é recomendável considerar o uso de transações distribuídas fornecidas pelos Enterprise Services. Esses serviços usam o DTC para garantir as transações.

O DTC executa a comunicação entre as várias fontes de dados e garante que ou todos ou nenhum dado será “confirmado” *(commited).* Isso é conseguido mas com algum custo custo operacional. Caso você não possua transações que se espalhem sobre várias origens de dados, as transações manuais Transact-SQL (T-SQL) ou ADO.NET freqüentemente oferecem melhor desempenho. No entanto, é necessário balancear entre os benefícios de desempenho contra a facilidade de programação. As transações declarativas do Enterprise Services oferecem um modelo de programação simples baseado em componente.

Você Manipula *BLOBs?*

Se você precisar ler ou gravar dados *BLOB,* como imagens, primeiramente é necessário considerar as opções de armazená-los diretamente no disco rígido, armazenando ou seu caminho físico ou a sua URL no banco de dados. Isso reduz a carga no banco de dados. Caso você leia ou escreva em campos *BLOBs,* uma das formas mais ineficientes é executar esta operação numa chamada única. Isso faz com que o *BLOB* seja transferido completamente pela rede e seja armazenado na memória. Isso pode causar congestionamento na rede e pressão na memória, particularmente quando existe uma carga considerável de usuários concorrentes.

Se você não precisar armazenar os dados do *BLOB* no banco de dados, considere as seguintes opções para reduzir o custo de desempenho:

- Use o particionamento *(chunking)* dos *Blobs* para reduzir a quantia de dados transferidos na rede. O particionamento envolve mais idas e vindas, mas coloca comparativamente menor carga no servidor e consome menos largura de banda da rede. É possível usar o ***DataReader.GetBytes*** para ler os dados em partes ou usar os comandos específicos do SQL Server, como *READTEXT* e *UPDATEDTEXT,* para executar tais tarefas de particionamento.
- Evite mover o *BLOB* repetidas vezes, pois o custo de movê-lo pode ser significativo em termos de recursos do servidor e da rede. Considere armazenar o *BLOB* em cache no cliente após uma operação de leitura.

Você Está Consolidando Códigos Repetidos para Acesso a Dados?

Caso você possua várias classes efetuando o acesso aos dados, é recomendável pensar em consolidar esta funcionalidade que se repete numa classe de ajuda. Desenvolvedores com nível variado de experiência e conhecimento em acesso a dados podem, inesperadamente, usar abordagens inconsistentes para acessar dados e, inadvertidamente, podem introduzir problemas de desempenho e escalabilidade.

Ao consolidar o código crítico de acesso a dados, é possível concentrar seus esforços de ajuste e usar uma única abordagem consistente para o gerenciamento de conexão com o banco de dados e o acesso aos dados.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre as práticas recomendadas de códigos de acesso a dados, consulte "*Data Access Application Block for .NET*" no MSDN no endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daab-rm.asp.*
- Para obter mais informações sobre as práticas recomendadas de acesso a dados, consulte *.NET Data Access Architecture Guide* no MSDN pelo endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/daag.asp*

Manipulação de Exceções

Os principais problemas de manipulação de exceções que precisam ser considerados encontram-se destacados na Tabela 4.9.

Tabela 4.9: Problemas de Manipulação de Exceções

Problemas	Implicações
Validações de código no cliente insatisfatórias	Idas e vindas aos servidores e chamadas caras.
Exceções como um método de controlar o fluxo regular do aplicativo	Custosas quando comparadas com o retorno do valor de uma enumeração ou de um booleano.
Disparo e captura de muitas exceções	Aumento da ineficiência.
Captura de exceções desnecessárias	Aumenta a sobrecarga e pode ocultar informações desnecessariamente.

Para avaliar a eficiência de sua abordagem na manipulação de exceções, revise as seguintes perguntas:

- **Você usa exceções para controlar o fluxo do aplicativo?**
- **As fronteiras de manipulação de exceções estão bem-definidas?**
- **Você usa códigos de erro?**
- **Você captura exceções somente quando necessário?**

Você Usa Exceções para Controlar o Fluxo do Aplicativo?

Não é recomendável usar exceções para controlar o fluxo do aplicativo porque o disparo de exceções pode ser caro. Algumas das alternativas incluem o seguinte:

- Altere a API de forma que indique o sucesso ou falha retornando um valor ***bool***, conforme exibido no seguinte código:

```
// BAD WAY
// ... search for Product
if ( dr.Read() ==0 ) // no record found, ask to create {
//this is an example of throwing an unnecessary exception because
//nothing has gone wrong and it is a perfectly acceptable situation
throw( new Exception("User Account Not found")); }
// GOOD WAY
// ... search for Product
if ( dr.Read() ==0 ){//no record found, ask to create return false;
```

- Refaça seu código para incluir a lógica de validação para evitar exceções em vez de disparar exceções.

Mais Informações

Para obter mais informações, consulte os seguintes recursos:

- Para obter mais informações sobre como usar exceções, consulte "*Writing Exceptional Code*" no MSDN pelo endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp07192001.asp.*
- Para saber as práticas recomendadas de gerenciamento, consulte *Exception Management Architecture Guide* no MSDN pelo endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp.*

Os Limites de Manipulação de Exceções estão Bem-Definidos?

É recomendável capturar, empacotar e diparar de novo as exceções em locais previsíveis. A manipulação de exceções deve ser implementada usando um conjunto comum de técnicas de manipulação de exceções em cada camada do aplicativo. Fronteiras bem definidas para a manipulação de exceções ajudam a evitar a redundância e a inconsistência na forma com que as exceções são capturadas e manipuladas e ajudam a manter um nível apropriado de abstração do erro. Evitar a redundância das manipulações de exceções melhora o desempenho do aplicativo e pode ajudar a simplificar as informações de instrumentação que o operador recebe do aplicativo.

É comum definir as fronteiras de gerenciamento de exceções em torno dos componentes que acessam recursos ou serviços externos, e em torno de fachadas *(façades)* que os sistemas externos ou as lógicas de interface de usuário podem acessar.

Você Usa Códigos de Erro?

Geralmente, é recomendável evitar usar códigos de retorno de método para indicar condições de erro. Em vez disso, use a manipulação de exceções estruturada. O uso de exceções é muito mais expressivo, resulta em código mais robusto e é menos propenso a abusos do que o uso de códigos de erro passados por valores no retorno.

O *Common Language Runtime* (CLR) usa exceções internamente mesmo em porções não gerenciadas da máquina virtual. No entanto, a sobrecarga de desempenho associada às exceções deve ser fatorada em sua decisão. É possível retornar um valor ***bool*** simples para informar o chamador sobre o resultado da chamada de função.

Você Captura Exceções Somente Quando Necessário?

A captura de exceções e o redisparo são caros e dificultam a depuração e identificação do código fonte exato que foi responsável pela exceção. Não capture exceções a menos que deseje especificamente registrar e colocar em *log* os detalhes da exceção, ou que possa tentar novamente uma operação com falha. Se você não fizer nada com a exceção, é provável que termine por redisparar esta mesma exceção adiante. Considere as seguintes diretrizes para capturar exceções:

- Não é recomendável capturar arbitrariamente as exceções a menos que você adicione algum valor. Deixe a exceção propagar-se da pilha de chamada até um código de tratamento que possa executar algum processamento apropriado.
- Não absorva qualquer exceção que não saiba como processar. Por exemplo, não absorva exceções em seu bloco *catch,* conforme mostrado no seguinte código.

```
catch (Exception e) {//Do nothing
```

Mais Informações

Para obter mais informações sobre as perguntas e problemas que surgiram desta seção, consulte "*Gestão de Exceções*" no Capítulo 5, "*Melhorando o Desempenho do Código Gerenciado*".

Considerações sobre Design das Classes

Use as seguintes perguntas para ajudar a revisar o design da classe:

- **A sua classe possui os dados sobre os quais ela age?**
- **As suas classes expõem interfaces?**
- **As suas classes contêm métodos virtuais?**
- **As suas classes contêm métodos que recebem parâmetros variáveis?**

A sua Classe Possui os Dados que Sobre os Quais Ela Age?

Revise os designs das suas classes para garantir que classes individuais agrupem dados e comportamentos relacionados apropriadamente. As classes devem ter a maioria dos dados que precisam para fins de processamento e não devem ser excessivamente dependentes de outras classes filho. Muita dependência em outras classes pode levar rapidamente a idas e vindas de chamadas de forma ineficientes.

As suas Classes Expõem Interfaces?

Geralmente, é recomendável usar uma abordagem baseada na interface implícita da classe por meio do empacotamento da funcionalidade e da exposição de uma única API (método) capaz de executar uma unidade de trabalho. Isso evita o custo do uso desnecessário de tabelas virtuais.

Use interfaces explícitas somente quando precisar oferecer suporte para várias versões ou quando precisar definir funcionalidades comuns aplicáveis a várias implementações de classe (ou seja, para polimorfismo).

As suas Classes Contêm Métodos Virtuais?

Revise a forma como você usa os membros virtuais em suas classes. Se não precisar estender sua classe, evite usá-los porque, para o .NET Framework 1.1, a chamada de um método virtual envolve uma busca na tabela virtual. Como resultado, os métodos virtuais não permitem que o compilador use a técnica de *inlining,* pois o destino final não é conhecido no momento do design.

Use os membros virtuais apenas para fornecer extensibilidade à sua classe. Se você derivar de uma classe que possua membros virtuais, é possível marcar os métodos da classe derivada com a palavra-chave ***sealed,*** que resulta na invocação do método como um método não-virtual. Isso interrompe a cadeia de substituições virtuais.

Considere os seguintes exemplos:

```
public class MyClass{
        protected virtual void SomeMethod() {... }
```

É possível substituir e selar o método numa classe derivada da seguinte forma.

```
public class DerivedClass : MyClass {
  protected override sealed void SomeMethod () { ... }
```

Esse código termina a cadeia de substituições virtuais e faz de ***DerivedClass.SomeMethod*** um candidato para *inlining*.

As suas Classes Contêm Métodos que Recebem Parâmetros Variáveis?

Métodos com um número variável de parâmetros resultam em caminhos de código digerentes para cada possível combinação de parâmetros. Se você possui objetos de alto desempenho, você deve usar vários métodos com sobrecarrega com parâmetros variáveis ao invés de ter um único método complexo que recebe um número variável de parâmetros.

Mais Informações

Para obter mais informações sobre métodos com números variáveis, consulte a seção "*Methods With Variable Numbers of Arguments*" de "*Method Usage Guidelines*" em *.NET Framework General Reference*, disponível no MSDN pelo endereço *http://msdn.microsoft.com/library /default.asp?url=/library/en-us/cpgenref/html/cpconmethodusageguidelines.asp.*

Resumo

As revisões de arquitetura e design para desempenho devem ser partes regulares do ciclo de vida de desenvolvimento de aplicativos. As características de desempenho de seus aplicativos são determinadas por sua arquitetura e design. Nenhuma quantia de refinamento e otimização pode compensar decisões de design insatisfatórias que impedem fundamentalmente que seu aplicativo atinja seus objetivos de desempenho e escalabilidade.

Este capítulo apresentou um processo e um conjunto de perguntas que devem ser usadas para ajudar a efetuar as revisões. Aplique estas diretrizes de revisão a designs novos e existentes.

Recursos Adicionais

Para obter mais informações, consulte os seguintes recursos:

- Para obter uma lista de verificação para imprimir, consulte "Lista de Verificação: Revisão de Arquitetura e Design para Desempenho e Escalabilidade" na seção "Listas de Verificação" deste guia.
- Para uma abordagem orientada por perguntas destinada à revisão de códigos e da implementação pela perspectiva de desempenho, consulte o Capítulo 13, "Revisão de Código: Desempenho do Aplicativo .NET".
- Para obter informações sobre como avaliar se a arquitetura de software atenderá aos objetivos de desempenho, consulte *PASA: An Architectural Approach to Fixing Software Performance Problems,* por Lloyd G. Williams e Connie U. Smith, no site *http://www.perfeng.com/papers/pasafix.pdf.*

- Para obter informações sobre padrões, consulte *Enterprise Solution Patterns Using Microsoft .NET,* no MSDN, pelo endereço *http://msdn.microsoft.com/library/default.asp? url=/library/en-us/dnpatterns/html/ESP.asp.*
- Para obter informações sobre a arquitetura de aplicativos, consulte *Application Architecture for .NET: Designing Applications and Services* no MSDN, pelo endereço *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp.*

O documento completo em inglês do **Melhorando a Performance e Escalabilidade de Aplicações .Net** pode ser encontrado no formato pdf em http://msdn.microsoft.com/patterns.

Microsoft®