

NHibernate in Action

MEAP

Unedited Draft

Pierre Henri Kuaté
Tobin Harris
Christian Bauer
Gavin King



Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



MEAP Edição
Manning Early Access Program

Copyright 2008 Manning Publications

Para mais informações sobre este e outros títulos Manning ir para
www.manning.com

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Índice

Capítulo Um: Object / Relational Persistência na Net

Capítulo Dois: Olá NHibernate

Capítulo Três: aulas de redação e Mapeamento

Capítulo Quatro: Trabalhando com objetos persistentes

Capítulo Cinco: Transações, simultaneidade e cache

Capítulo Seis: conceitos de mapeamento avançada

Capítulo Sete: Recuperando objetos de forma eficiente

Capítulo Oito: Desenvolvimento de Aplicações NHibernate

Capítulo Nove: Writing Modelos de Domínio Real World

Capítulo Dez: Técnicas Avançadas de Persistent

Apêndice A: SQL Fundamentals

Apêndice B: Mais informações sobre atributos de mapeamento NHibernate

Apêndice C: Apresentando ActiveRecord e MonoRail

Apêndice D: Seguir em Frente

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



MEAP Edição
Manning Early Access Program

Copyright 2008 Manning Publications

Para mais informações sobre este e outros títulos Manning ir para
www.manning.com

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Object / Relational Persistence in. NET

Este capítulo aborda

. NET persistência e bancos de dados relacionais

Camadas. NET

Abordagens para implementar a persistência in. NET

Como NHibernate resolve persistência de objetos em bancos de dados
relacionais

Persistência recursos avançados

Desenvolvimento de software é uma disciplina em constante mutação, onde novas técnicas e tecnologias são constantemente emergentes. Como desenvolvedores de software, temos um enorme conjunto de ferramentas e práticas disponíveis, e escolher o que é certo muitas vezes pode fazer ou quebrar um projeto. Uma escolha que é pensada para ser particularmente crítica é a forma de gerenciar dados persistentes, ou mais simplesmente, como armazenar, carregar e salvar dados.

Existem infinitas opções disponíveis para nós. Podemos armazenar os dados em binário simples ou arquivos de texto em um disco. Podemos escolher diferentes formatos, incluindo CSV, XML, JSON, YAML, SOAP, ou mesmo inventar o nosso próprio formato. Alternativamente, podemos enviar nossos dados através da rede para outra aplicação ou serviço, como um relacional banco de dados, um servidor Active Directory, ou uma fila de mensagens. Poderíamos até mesmo necessidade de armazenar dados em vários lugares, ou combinar todas essas opções em um único aplicativo.

Como você pode começar a perceber, o gerenciamento de dados persistentes é considerado um tema espinhoso. De todas as opções, bancos de dados relacionais têm provado ser extremamente popular, mas existem ainda muitas opções, perguntas e opções com que nos defrontamos em nosso trabalho diário. Por exemplo, devemos usar DataSets, ou são mais DataReaders

adequada? Devemos usar procedimentos armazenados? Devemos código SQL nosso lado, ou deixar que nossas ferramentas dinamicamente gerá-lo para nós? Devemos fortemente o nosso tipo DataSets? Ou, se nós realmente construir uma mão-codificado domínio

modelo contendo classes? Se sim, como é que vamos carregar e salvar os dados de e para o banco de dados? Fazer usamos a geração de código? A lista de continua ...

Este tópico não é restrito apenas para. NET. Então toda comunidade de desenvolvimento vem debatendo este tema, muitas vezes ferozmente, por muitos anos.

Debate continua, mas uma abordagem que ganhou popularidade generalizada é mapeamento objeto / relacional, ou ORM. Ao longo dos anos, muitas bibliotecas e ferramentas surgiram para ajudar a implementar ORM desenvolvedor em suas

aplicações. Uma dessas ferramentas é NHibernate - um objeto sofisticado e maduro / ferramenta de mapeamento relacional para

. NET.

NHibernate é um porto. NET do popular "Hibernate" Java biblioteca. NHibernate tem como objetivo ser um completo solução para o problema de gerenciamento de dados persistentes quando se trabalha com bancos de dados relacionais e modelo de domínio

classes. Esforça-se para realizar o árduo trabalho de mediação entre a aplicação eo banco de dados, deixando o desenvolvedor livre para se concentrar no problema em questão. Este livro abrange tanto básicos e avançados Uso NHibernate. Ele também recomenda as melhores práticas para o desenvolvimento de novas aplicações usando

[Referências](http://www.manning-sandbox.com/forum.jspa?forumID=295)

Antes que possamos começar com o NHibernate, que será útil para você entender o que a persistência é e as diversas maneiras que podem ser implementados usando a estrutura. NET. Este capítulo irá ajudá-lo a entender por que ferramentas como o NHibernate são realmente necessários.

Primeiro, vamos definir o conceito de persistência no contexto de aplicativos. NET. Em seguida, demonstrar como um clássico de aplicação. NET é implementada, usando as ferramentas de persistência padrão disponível no quadro. NET. Você vai descobrir algumas dificuldades comuns encontrados quando se utiliza bancos de dados relacionais com object-oriented frameworks como. NET, e como populares abordagens persistência tentar resolver estes problemas. Coletivamente, estes problemas são referidos como os incompatibilidade paradigma entre orientada a objetos e design de banco de dados. Nós em seguida, ir para introduzir a abordagem adoptada pela NHibernate e muitas de suas vantagens. Depois disso, nós vamos cavar para alguns desafios persistência complexas que a tornam uma ferramenta como o NHibernate essencial. Finalmente, definimos

o Mapeamento Objeto / Relacional é e porque você deve usá-lo. Até o final deste capítulo, você deve ter um idéia das grandes pressões que você pode encontrar com o NHibernate?

Não. Se você quiser tentar NHibernate imediatamente, pule para o capítulo 2, onde no salto e começar a programar uma (Pequeno) aplicação NHibernate. Você será capaz de entender o capítulo 2, sem ler este capítulo, mas recomendamos que você leia este capítulo se você é novo à persistência no. NET. Dessa forma, você vai compreender as vantagens de NHibernate e saber quando usá-lo. Você também vai aprender algumas conceitos importantes como unidade de trabalho. Então, se você estiver interessado por essa discussão, você poderia muito bem continuar com o capítulo 1, ter uma idéia ampla de persistência em. NET e depois seguir em frente.

1.1 O que é persistência?

A persistência é uma das preocupações fundamentais no desenvolvimento de aplicações. Se você tem alguma experiência em desenvolvimento de software, você já lidou com isso. Na verdade, quase todas as aplicações requerem dados persistentes. Usamos persistência para permitir que nossos dados sejam armazenados, mesmo quando os programas que o utilizam não estão funcionando.

Para ilustrar isso, digamos que você deseja criar um aplicativo que permite aos usuários armazenar sua empresa números de telefone e detalhes de contato, e recuperá-los sempre que necessário. A menos que você quer que o usuário deixar o programa em execução o tempo todo, você logo percebe que sua aplicação terá de alguma forma, salvar o contatos em algum lugar. Você está agora confrontado com uma decisão persistência, você vai precisar trabalhar para fora que persistência mecanismo você deseja usar. Você tem a opção de persistir os dados em muitos lugares, o mais simples de ser um arquivo de texto. Mais frequentemente do que não, você pode escolher um banco de dados relacional, porque é amplamente comum. Bancos de dados relacionais oferece algumas características grandes para armazenar e recuperar de forma confiável de dados.

Você provavelmente já trabalhou com um banco de dados relacional como o Microsoft SQL Server, MySQL ou Oracle. Se não for o caso, dê uma olhada no apêndice A. A maioria de nós usar bancos de dados relacionais a cada dia, eles têm ampla aceitação e são considerados uma solução robusta e madura para gerenciamento de dados moderna desafios.

Um sistema de gerenciamento de banco de dados relacional (RDBMS) não é específico para. NET, e um banco de dados relacional não é necessariamente específica para qualquer aplicação. Assim, podemos ter várias aplicações acessando um único banco de dados, alguns escritos em. NET, alguns escritos em Java ou etc Rubi tecnologia relacional fornece uma maneira de compartilhamento de dados entre muitas aplicações diferentes. Na verdade, mesmo diferentes componentes em um único aplicativo pode independentemente acessar um banco de dados relacional (um motor de comunicação e uma componente de log, por exemplo). Essencialmente, a tecnologia relacional é um denominador comum de muitos sistemas independentes e tecnologia Plataforma de gerenciamento de banco de dados relacional é muito comum. A apresentação de toda a empresa comum de negócios <http://www.manning-sandbox.com/forum.jspa?forumID=295>

objetos. Ou mais simplesmente, um negócio geralmente é necessário para armazenar informações sobre várias coisas, como

Clientes, contas e produtos etc (os objetos de negócios), e banco de dados relacional é geralmente a escolhida lugar central, onde ambos são definidos e armazenados. Isso torna o banco de dados relacional muito importante peça no cenário de TI.

Sistemas de gerenciamento de banco de dados relacional SQL têm interfaces baseadas em programação de aplicações, daí nós produtos de hoje chamada banco de dados relacional SQL sistemas de gestão de dados ou, quando estamos falando de sistemas particulares, SQL bancos de dados.

1.1.2 Noções básicas sobre SQL

Como acontece com qualquer desenvolvimento de banco de dados .NET, uma sólida compreensão das bases de dados relacionais e SQL é um

pré-requisito ao usar NHibernate. Você precisa usar o seu conhecimento de SQL para ajustar o desempenho do sua aplicação NHibernate. NHibernate irá automatizar muitas tarefas repetitivas de codificação, mas o seu conhecimento de tecnologia de persistência deve se estender além NHibernate-se se você quiser aproveitar toda a potência do modernos bancos de dados SQL. Lembre-se que o objetivo subjacente é de gestão, robusta e eficiente de dados persistentes.

Se você acha que pode precisar de melhorar suas habilidades SQL, em seguida, pegar uma cópia do excelente livro SQL

Sintonia por Dan Tow [Tow 2003] e SQL Cookbook por Anthony Molinaro [Mol 2005]. Joe também tem Celko alguns excelentes livros sobre técnicas avançadas SQL. Para um fundo mais teórico, considere a leitura de um Introdução aos Sistemas de Banco de Dados [Data 2004].

1.1.3 Usando SQL em Aplicações .NET

.NET nos oferece muitas ferramentas e opções quando se trata de fazer as nossas aplicações trabalhar com bases de dados SQL.

Poderíamos magra sobre o Visual Studio IDE, tirando partido da sua capacidade de arrastar e soltar onde, em uma série de cliques do mouse, podemos criar conexões de banco de dados, executar consultas e visualizar os dados editáveis na tela. Nós

acredito que este é ótimo para aplicações simples, mas a abordagem não escala bem para maior, mais complexo aplicações.

Alternativamente, podemos usar objetos SqlCommand e escrever manualmente e executar o nosso SQL para construir DataSets. Isto pode rapidamente tornar-se um pouco entediante, e aquilo que realmente quer fazer é trabalhar em um pouco maior nível de abstração de modo que podemos nos concentrar em resolver problemas de negócio ao invés de se preocupar com acesso a dados preocupações. Se você está interessado em aprender mais sobre a ampla gama de abordagens experimentadas e testadas para os dados acesso, então considere a leitura de Martin Fowlers "Patterns of Enterprise Application Architecture", onde muitos técnicas são catalogados explicado em profundidade.

De todas as opções, a abordagem que queremos ter é a de escrever classes reais - ou entidades empresariais - que podem ser carregados e salvos de e para o banco de dados. Ao contrário de DataSets, essas classes não são projetados para espelhar a estrutura de um banco de dados relacional (como linhas e colunas). Em vez disso, estão preocupados com a solução do problema em questão. Juntas, essas aulas normalmente representam o orientada a objeto modelo de domínio.

1.1.4 Persistência em aplicações orientadas objeto

Em uma aplicação orientada a objetos, a persistência permite que um objeto sobreviver ao processo ou aplicativo que criou

-lo. O estado do objeto pode ser armazenado em disco e um objeto com o mesmo estado re-criado em algum ponto no futuro.

Esta aplicação não se limita a única objetos-todo de objetos gráficos interligadas podem ser feitas persistentes e mais tarde re-criado. A maioria dos objetos não são realmente persistentes; uma transitório objeto é um que tem uma vida útil limitada, que é limitada pela vida do processo que instanciado-lo. Um exemplo simples é um controle da web Por favor, postar comentários ou correções para o fórum on-line em Autor <http://www.manning-sandbox.com/forum.jspa?forumID=295>

objeto que só existe na memória por uma fração de segundo antes que ela seja processada a tela e liberados do memória. Quase todas as aplicações. .NET contém uma mistura de objetos persistentes e transientes, e faz boa sentido ter um subsistema que gerencia as persistentes.

Modernos bancos de dados relacionais fornecem uma representação estruturada de dados persistentes, permitindo a classificação, busca e agrupamento de dados. Sistemas de gerenciamento de banco de dados são responsáveis pela gestão coisas como simultaneidade e integridade de dados, eles são responsáveis pelo compartilhamento de dados entre vários usuários e vários aplicativos. Um sistema de gerenciamento de banco de dados também fornece dados de nível de segurança. Quando discutimos a persistência em *Armazenamento, organização, recuperação e de dados*. Neste livro, estamos pensando em todas essas coisas.

Simultaneidade e integridade de dados

Compartilhamento de dados

Em particular, estamos pensando desses problemas no contexto de uma aplicação orientada a objeto que utiliza um modelo de domínio. Uma aplicação com um modelo de domínio não trabalha diretamente com a representação tabular de as entidades de negócios (ou seja, utilizando DataSets), o aplicativo tem o seu próprio, modelo orientado a objetos do negócio

entidades. Se o banco de dados tabelas ITEM e BID, a aplicação. .NET iria definir classes de itens e Bid ao invés de usar DataTables para estes.

Então, em vez de trabalhar directamente com as linhas e colunas de uma DataTable, a lógica do negócio interage com este modelo de domínio orientada a objeto e sua realização em tempo de execução como um gráfico de objetos interligados. O

lógica de negócio nunca é executado no banco de dados (como um procedimento armazenado SQL), é implementado em. .NET. Este

permite que a lógica do negócio para fazer uso de sofisticados conceitos orientados a objetos como herança e polimorfismo. Por exemplo, poderíamos usar padrões de projeto bem conhecidas, tais como Estratégia, Mediador, e Composite [GOF 1995], que dependem de chamadas de métodos polimórficos. Agora uma ressalva: Não .NET todos. aplicativos são projetados dessa maneira, nem devem ser. Aplicações simples pode ser muito melhor sem uma modelo de domínio. SQL eo ADO.NET são perfeitamente utilizável para lidar com pura dados tabulares, e os DataSet faz operações CRUD ainda mais fácil. Trabalhando com uma representação tabular de dados persistentes é simples e bem compreendida.

No entanto, no caso de aplicações com lógica de negócios não trivial, o modelo de domínio ajuda a melhorar reutilização de código e de manutenção de forma significativa. Nós nos concentramos em aplicações com um modelo de domínio neste livro, uma vez

NHibernate e ORM, em geral, são mais relevantes para este tipo de aplicação.

Será útil para compreender como esse modelo de domínio se encaixa no "quadro maior" de um software inteiro do sistema. Para explicar isso, damos um passo atrás e olhar para algo chamado de arquitetura em camadas.

1.1.5 Persistência e da arquitetura em camadas

Muitos, se não a maioria, os sistemas atuais são projetados com uma arquitetura em camadas, e NHibernate funciona muito bem

com esse design. No entanto, o que exatamente é uma arquitetura em camadas?

Uma arquitetura em camadas divide um sistema em que vários "grupos", onde cada grupo contém o código de endereçamento um área de problema particular. Esses grupos são chamados de camadas. Por exemplo, uma camada de interface de usuário pode conter todos os o código do aplicativo para a construção de páginas web e processamento de entrada do usuário. Um dos principais benefícios para as camadas

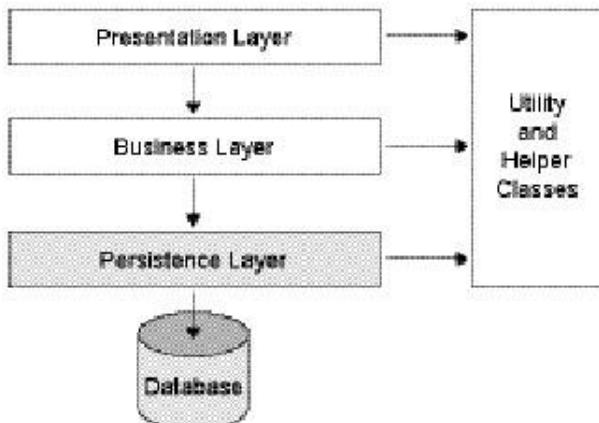
abordagem é que as mudanças a uma camada muitas vezes pode ser feita sem interrupções significativas para as outras camadas, assim

abrir para baixo de tornar os sistemas mais fáceis de modificar e testar. Qualquer alteração em uma camada deve cumprir algumas regras básicas:

Para aplicações de negócios, existe um popular, comprovada, arquitetura de aplicações de alto nível que é composto por três

camadas: A camada de apresentação, a camada de lógica de negócio ea camada de persistência. Veja a figura 1.1.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



Arquitetura em camadas destacando a camada de persistência.

Vamos dar uma olhada nas camadas e elementos no diagrama:

A apresentação da camada de lógica interface de usuário é mais alto. Em uma aplicação web isso contém o código

responsável pela elaboração páginas ou telas, coletando a entrada do usuário e controle de navegação. A camada de negócios forma exata desta camada varia muito entre as aplicações. E geralmente aceite, no entanto, que a camada de negócios é responsável por implementar as regras de negócio ou sistema requisitos que seria compreendido por usuários como parte do domínio do problema. Na maioria das vezes, este camada de partes da representação das entidades de domínio de negócio com a camada de persistência.

Revisitamos

esta questão no capítulo 3.

A camada de persistência- camada de persistência é um grupo de classes e componentes responsável por salvar e recuperação de dados de aplicativos de e para uma ou mais lojas de dados. Esta camada define um mapeamento

entre as entidades empresariais de domínio e banco de dados. Não pode surpreendê-lo ao saber que NHibernate seriam utilizados principalmente nesta camada.

O banco de dados banco de dados existe fora do aplicativo. NET. É a representação real e persistente do estado do sistema. Se um banco de dados SQL é usada, o banco de dados inclui o esquema relacional e, possivelmente, procedimentos armazenados.

Auxiliar / utilitário classes-Cada aplicação tem um conjunto de classes de infra-estrutura auxiliar ou utilitário que apoio das outras camadas. Por exemplo, widgets UI, classes de mensagens, **Exceção** classes e exploração madeireira

utilitários. Estes elementos de infra-estrutura não são consideradas como uma camada, porque eles não obedecem ao

regras para a dependência intercalar em uma arquitetura em camadas.

Lembre-se que as camadas são particularmente úteis para quebrar aplicações grandes e complexas, e muitas vezes são um exagero para as aplicações extremamente simples. NET. Para tais programas simples, você pode optar por colocar todos os seus

código em um só lugar. Assim, em vez de separar nitidamente as regras de negócio e funções de acesso banco de dados em separado

camadas, você colocá-los todos no seu code-behind (ou classes Windows Forms). Ferramentas como o Visual Studio .NET tornam muito fácil e indolor para construir este tipo de aplicação simples. Entretanto, esteja ciente que esta abordagem pode levar rapidamente a uma base de código problemático, como a aplicação cresce, você tem que adicionar mais e

mais código para cada formulário ou página, e as coisas começam a tornar-se cada vez mais difícil trabalhar com ele. Além disso,

alterações feitas no banco de dados pode facilmente quebrar o aplicativo, e encontrar e corrigir as partes afetadas pode ser desafiador. Fazendo assim?

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Embora um arquitetura de três camadas é comum e vantajoso em muitos casos, nem todos os NET. aplicativos são projetados como essa, nem devem ser. Aplicações simples pode ser muito melhor sem objetos complexos. SQL ea API ADO.NET são perfeitamente utilizável para lidar com pura dados tabulares, eo DataSet ADO.NET faz operações básicas ainda mais fácil. Trabalhando com um tabular representação de dados persistentes é simples e bem compreendida.

Abordagens para 1,2 Persistence in. NET

Discutimos como, em qualquer aplicação de tamanho considerável, uma camada de persistência é necessária para lidar com a carga e poupança de dados. Existem muitas abordagens disponíveis para nós na construção desta camada de persistência, cada uma com seu vantagens e desvantagens. Três escolhas populares são:

Mão-de codificação

NHibernate (ou similar)

Apesar do fato recomendamos NHibernate, é sempre prudente considerar as alternativas. Como você vai aprender mais rápido, a construção de aplicações com NHibernate é bastante simples, mas isso não significa que ele é perfeito para cada projeto. Nas seções seguintes, vamos examinar e comparar essas estratégias em detalhes, discutindo implicações no acesso banco de dados e interface do usuário.

1.2.1 Escolha da camada de persistência

Em nossas aplicações, que muitas vezes deseja carregar, manipular e guardar itens de e para o banco de dados. Independentemente de que a persistência abordagem que usemos, em algum ponto objetos ADO.NET deve ser criado e comandos SQL deve ser executado. Seria tedioso e improdutivo para escrever todo este código SQL cada vez que nós temos que manipular alguns dados, por isso podemos usar uma camada de persistência para cuidar destes passos de baixo nível.

Essencialmente, a camada de persistência é o conjunto de classes e utilitários usados para tornar a vida mais fácil quando se trata de salvar e carregar dados. ADO.NET permite a execução de comandos SQL que realmente executar a persistência, mas a complexidade desse processo exige que envolva estes comandos por trás dos componentes que compreendem como nossas entidades deve ser mantida. Esses componentes podem também esconder as especificidades da base de dados, tornando nossa aplicação menos acoplado ao banco de dados, portanto, mais fácil de manter. Por exemplo, quando você usa um SQL identificador contendo espaços ou palavras-chave reservadas, você deve delimitar este identificador. Bases de dados como SQL Server use colchetes para que, enquanto MySQL use back-carrapatos. É possível para esconder esse detalhe e deixe a camada de persistência selecione o delimitador direito.

DataSet é uma abordagem de persistência muito diferente.

Visual Studio nos permite facilmente gerar nossa própria camada de persistência, que pode ser estendido com novos funcionalidade com poucos cliques. As classes geradas pelo Visual Studio sabe como acessar o banco de dados e pode ser usado para carregar e salvar as entidades contidas no DataSet.

Mais uma vez, uma pequena quantidade de trabalho é necessário para começar. No entanto, ainda temos que cair de volta para a mão-codificação quando mais é necessário o controle, que normalmente é inevitável, como descrito no ponto 1.3.

Mão-codificado camada de persistência

Mão codificação de uma camada de persistência pode finalmente envolvem muito trabalho, é comum criar primeiro um conjunto genérico de funções para lidar com conexões de banco de dados, execução de comandos SQL, etc Então, em cima desta camada sub-Por favor, postar comentários ou correções para o fórum on-line em Autor <http://www.manning-sandbox.com/forum.jspa?forumID=295>

então, temos que construir um outro conjunto de funções que salvar, carregar e encontrar nossas entidades de negócio. As coisas ficam muito mais envolvidos, se você precisa de introduzir o cache, a aplicação da regra de negócio ou a manipulação das relações entidade.

Obviamente, a mão de codificação sua camada de persistência dá-lhe o maior grau de flexibilidade e controle, você ter liberdade de design mais moderno e pode facilmente explorar as características de banco de dados especializados. Por outro lado, pode ser um empreendimento enorme, e muitas vezes é bastante enfadonho e repetitivo de trabalho, mesmo quando se usa geração de código.

Camada de persistência usando NHibernate
NHibernate fornece todos os recursos necessários para construir rapidamente uma camada de persistência avançado em código. É capaz de carregar e salvar gráficos inteira de objetos interligados, mantendo as relações entre eles.

No contexto do nosso aplicativo de leilão, podemos facilmente salvar um **Item e sua Bids** pela implementação de um método como:

```
public void Salvar (item Item) {  
    OpenNHibernateSession ();  
    Session.save (item);  
    CloseNHibernateSession ();  
}
```

Aqui, a sessão é um objeto fornecido por NHibernate. Não se preocupe em entender o código ainda. Por agora, nós só queremos que você veja como é simples a camada de persistência é com NHibernate. Vamos começar a usar o NHibernate em Capítulo 2, onde você vai descobrir que é muito simples de executar operações de persistência. Tudo o que você precisa fazer é escrever suas entidades e explicar para NHibernate como persistir eles.

1.2.2 Implementar as entidades

Aquele que escolheu uma abordagem camada de persistência, podemos nos concentrar na construção de nossos objetos de negócio, ou entidades, que o aplicação irá manipular. Basicamente, são classes que representam os elementos do mundo real que o nosso aplicativo deve manipular. Para um aplicativo de leilão, **Usuário**, **Item** e **Oferta** seriam exemplos comuns. Vamos agora discutir como podemos implementar nossas entidades de negócio em cada uma das três abordagens.

Entidades em um DataSet

Um DataSet representa um conjunto de tabelas de banco de dados, e por sua vez, estas tabelas contêm os dados das entidades.

Portanto, um DataSet armazena dados sobre objetos de negócios de uma forma semelhante a um banco de dados. A digitado gerada

DataSet podem ser usados para facilitar a manipulação de dados, e também é possível inserir uma lógica de negócios e regras.

Enquanto nós queremos manipular os dados,. NET e IDEs fornecem a maioria dos recursos necessários para trabalhar com uma

DataSet. No entanto, assim como nós pensamos sobre objetos de negócios como objetos no sentido de design orientado a objetos,

que dificilmente podem ser satisfeitas por um DataSet (digitado ou não). Afinal, os objetos de negócios representam no mundo real

elementos, e estes elementos têm dados e comportamento. Eles podem estar ligados por relações avançadas como herança, mas isso não é possível com o DataSet. Este nível de liberdade na concepção das entidades só pode ser

Mãos codificando entidades

Voltando ao exemplo de um aplicativo de leilão, foram considerados tendo as entidades: **Usuário**, **Item** e **Oferta**.

Além dos dados que contêm, nós também esperar que haja relações entre eles. Por exemplo, um

Item tem uma coleção de propostas e uma **Oferta** refere-se a um item, em classes C #, isso pode ser expresso através de um

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

coleção como `item.Bids` e uma propriedade como `bid.Item`. Como você pode ver, a visão orientada a objetos é diferente da visão relacional de coisas: Em vez de ter chaves primárias e estrangeiras, temos associações. Você encontrará que o design orientado a objetos dá-nos alguns outros conceitos de modelagem poderosa como herança e polimorfismo.

Mão-codificado entidades estão livres de qualquer restrição, pois eles são mesmo livres da forma como eles são persistentes no banco de dados. Eles podem evoluir (quase) de forma independente e ser compartilhado por aplicações muito diferentes, o que é muito vantagem importante quando se trabalha em um ambiente complexo.

No entanto, eles são difíceis e tedioso para o código, pense sobre o trabalho manual que seria necessário para suporte a persistência de entidades herdando de outras entidades. É comum o uso de geração de código ou de base classes (como `DataSet`) para adicionar recursos com um mínimo de esforço. Estas características podem estar relacionadas com a persistência, transferência ou apresentação de informações. No entanto, sem um quadro útil, esses recursos podem ser muito demorado de implementar.

Entidades e NHibernate

NHibernate é não-invasivos; é comum usá-lo com mão-codificado (ou gerados) entidades. Você deve fornecer informações de mapeamento que diz como estas entidades devem ser carregados e salvos. Portanto, você obtém o melhor dos dois mundos: Object-oriented entidades facilmente mantidas em um banco de dados relacional. Há muitas diferenças fundamentais entre objetos e dados relacionais. Tentando usá-los juntos revela a incompatibilidade de paradigma (também chamado de objeto / relacional diferença de impedância). Vamos explorar esse descompasso no ponto 1.3. Até o final deste capítulo, você terá uma idéia clara dos problemas causados pelo paradigma incompatibilidade e como NHibernate resolve esses problemas.

Uma vez que as entidades são implementados, devemos pensar em como eles serão apresentados ao usuário final.

1.2.3 entidades Exibindo na interface do usuário

Usando NHibernate implica usar entidades, e usando entidades tem algumas consequências sobre a maneira como o usuário Interface (UI) é escrito. Para o usuário final, a interface do usuário é um dos elementos mais importantes. Se é um Web aplicativo (usando `ASP.NET`) Ou um aplicativo do Windows, ele deve satisfazer as necessidades do usuário. A profunda discussão em torno da implementação de uma interface do usuário não está no escopo deste livro, mas a maneira como a camada de persistência é implementado tem um efeito direto sobre a maneira como a interface do usuário será implementada.

Neste livro, vamos nos referir à interface do usuário como o camada de apresentação. .NET fornece controles para exibir da informação. A simplicidade dessa operação depende de como a informação é armazenada.

DataSet baseado camada de apresentação

A Microsoft fez o esforço de adicionar suporte para a ligação de dados com `DataSet` mais in. Controles NET. É fácil para vincular um `DataSet` a um controle para que sua informação é apresentada e qualquer alterado (feito pelo usuário) reverberou no `DataSet`.

Usando `DataSets` é provavelmente a forma mais produtiva para implementar uma camada de apresentação. Você pode perder algum controle sobre como a informação é apresentada, mas é bom o suficiente na maioria dos casos. A situação é um pouco mais complicada com a mão-coded entidades.

Camada de apresentação e entidades

A dificuldade em dados, informações vinculativas constantes da mão-codificado entidades reside no fato de que não são tão muitas possibilidades de implementá-las. Um `DataSet` é feito de tabelas, colunas e linhas, mas uma mão-codificado entidade, uma classe, sem qualquer restrição, contém campos e métodos, sem qualquer forma padronizada para acesso e exibi-los. .NET controles podem ser ligados a dados a entidades, neste caso, eles só vão acessar as propriedades públicas.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Este é o suficiente em casos simples, caso contrário, temos de cair de volta para a mão do código do processo de carregamento de entidades na UI e copiar de volta as alterações persistem.

Embora esta abordagem não é tão fácil como quando se usa DataSets, ainda é bastante simples. E você está livre para apresentar suas entidades exatamente como você deseja. Mas existem algumas outras situações onde não é assim tão fácil. Um dos eles é relatórios; ferramentas como o Crystal Reports fornecer um suporte limitado para entidades.

Existem muitas bibliotecas (como o ObjectViews projeto open source), que simplificam muito os dados vinculação das entidades. Sugerimos que você dê uma olhada nessas bibliotecas. E não se esqueça que você é sempre livre para fallback para DataSets quando se lida com casos extremos como o de relatórios complexos onde são muito mais fáceis de manipular. Vamos discutir essa questão no capítulo 9.

Usando persistência capaz informações afeta a maneira como a interface do usuário é projetado. Basicamente, os dados devem ser carregados ao abrir uma interface de usuário e salvos ao fechar a interface do usuário. NHibernate propõe alguns padrões para lidar com esse

2.4 Implementar operações CRUD

Agora, todas as camadas estão no lugar, podemos trabalhar sobre a realização de ações.

Quando se trabalha com informações persistentes, estamos preocupados com a persistência e recuperar essa informação.

CRUD significa Create, Read, Update, Delete. Essas são operações primitivas executadas, mesmo no mais simples aplicação. Na maioria das vezes, essas operações são desencadeadas por eventos gerados na camada de apresentação. Para exemplo, o usuário pode clicar em um botão para ver um item. A camada de persistência será usado para carregar este item que será vinculado a um formulário exibindo seus dados.

Não importa qual a abordagem que você usa, essas operações primitivas são bem compreendidos e simples de implementar. Operações que são mais complexos são abordados na próxima seção.

Operações de CRUD com DataSets

Nós já sabemos que uma boa parte da camada de persistência pode ser gerada ao usar DataSets. Este camada de persistência contém classes para executar operações CRUD. E 2.0 NET Visual Studio 2005 e. Vêm com classes mais poderosa chamada adaptadores de tabela.

Não só essas classes de suporte operações primitivas CRUD, mas também são extensíveis. Você pode adicionar métodos chamar procedimentos armazenados ou gerar os comandos SQL por apenas alguns cliques. Mas se você quiser implementar algo mais complexo, você deve entregar o código-it, e veremos, na próxima seção, que há algumas características muito úteis que não são fáceis de implementar. E a estrutura de um DataSet pode torná-los mais difícil de implementar.

Mão-codificado operações CRUD

A mão-codificado operação CRUD faz exatamente o que você quer porque você escreve o comando SQL para executar; Por outro lado, é um trabalho muito repetitivo e chato.

É possível implementar uma estrutura que gera estes comandos SQL. Depois de entender que carregamento de uma entidade implica a execução de um SELECT em sua linha de banco de dados, você pode automatizar CRUD primitiva operações. No entanto, muito mais trabalho é necessário para consultas complexas e manipular interligados entidades.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Operações de CRUD usando NHibernate

Assim que você fornecer para NHibernate as informações de mapeamento de suas entidades, você pode executar um CRUD operação por uma única chamada de método. Esta é uma característica fundamental de um mapeamento objeto / relacional ferramenta (ORM). Uma vez que tenha todas as informações que necessita, ele pode resolver o objeto / relacional diferença de impedância em cada operação.

NHibernate é projetado para executar com eficiência operações CRUD. Experiência e os testes ajudaram descobrir várias otimizações e melhores práticas. Por exemplo, ao manipular entidades, melhores desempenhos são alcançados pela persistência para atrasar o fim do transação. Neste ponto, uma única conexão é usada para salvar todas as entidades.

Agora que nós cobrimos todos os passos básicos e as operações de persistência, podemos descobrir alguns avançados características que ilustram claramente as vantagens do NHibernate.

1.3 Por que precisamos de NHibernate?

Até agora, temos falado sobre uma aplicação muito simples. No mundo real, no entanto, raramente lidar com simples aplicações. Um aplicativo corporativo tem muitas entidades com lógica de negócios complexos e objetivos do projeto: Produtividade, manutenção, desempenho e são essenciais.

Nesta secção, iremos percorrer algumas características indispensáveis para implementar uma aplicação bem-sucedida.

Primeiro, vamos dar alguns exemplos que ilustram as diferenças fundamentais entre objetos e banco de dados relacional. Você também terá uma idéia de como NHibernate ajuda a criar uma ponte entre essas representações. Então, nós se voltam para a camada de persistência para descobrir como lida com NHibernate entidades complexas e numerosas. Você irá

aprender os padrões e recursos que ele oferece para alcançar o melhor desempenho. Por fim, cobrimos complexo consultas; veremos que NHibernate pode ser usado para escrever um motor de busca poderoso e completo.

Vamos começar com as entidades e seu mapeamento para um banco de dados relacional.

1.3.1 A incompatibilidade paradigma

Enquanto um banco de dados é relacional, estamos usando linguagens orientadas a objeto. Não há nenhuma maneira direta a persistir um objeto como uma linha do banco de dados. E persistência não deve prejudicar a nossa capacidade de entidades de design, que corretamente representam o que estamos manipulando.

Chamamos incompatibilidade paradigma (Ou objeto / relacional diferença de impedância) as incompatibilidades fundamentais entre o design de objetos e bancos de dados relacionais. Vamos dar uma olhada em alguns dos problemas criados pela incompatibilidade de paradigma.

O problema de granularidade

Granularidade se refere ao tamanho relativo dos objetos que você está trabalhando. Quando estamos falando .NET objetos e tabelas do banco de dados, o problema granularidade, os objectos persistentes que podem ter vários tipos de granularidade para tabelas e colunas que são inherentemente limitado em granularidade.

Vamos dar um exemplo do leilão on-line descrito no ponto 1.1.4. Vamos dizer que deseja adicionar um endereço para um **Usuário** objeto, não como uma string, mas como outro objeto. Como devemos persistir este usuário em um mesa? Podemos acrescentar uma **ENDEREÇO** mesa, mas geralmente não é uma boa idéia (por razões de desempenho). Podemos criar um tipo definido pelo usuário coluna (UDT), mas esta opção não for amplamente apoiada e portátil. Outra opção é mesclar as informações de endereço para o utilizador, mas este não é um projeto orientado a objetos bom e não é muito re-utilizáveis.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Acontece que o problema granularidade não é especialmente difícil de resolver. Na verdade, nós provavelmente não mesmo a lista que, se não fosse o fato de que é visível em tantas abordagens como DataSet. Descrevemos a solução para este problema no capítulo 3, seção 3.6, "modelos refinada objeto."

Um problema muito mais difícil e interessante surge quando consideramos a herança, uma característica do objeto projeto orientado que é comumente usado.

O problema da herança e polimorfismo

Linguagens orientadas a objeto apoiam a noção de herança. Mas este não é o caso de bancos de dados relacionais. Digamos que, em nosso aplicativo de leilão, podemos ter muitos tipos de itens. Poderíamos criar subclasses como **Mobiliário** e **Livro**, Cada um com suas informações específicas. Então, como devemos persistir essa hierarquia de entidades em um banco de dados relacional? Você também deve perceber que um **Oferta** pode se referir a qualquer subclasse de Item. Deve ser possível executar consultas polimórficas como recuperar todas as propostas em livros. No capítulo 3, seção 3.7, "Mapeamento herança de classe ", discutimos como as soluções de mapeamento objeto / relacional como o NHibernate resolver o problema de persistir uma hierarquia de classe a uma tabela de banco de dados ou tabelas.

O problema da identidade

A identidade de uma linha de banco de dados é comumente expressa como a chave primária valor. Como você verá no capítulo 3, seção 3.5, "Entendendo a identidade do objeto," NET. identidade de objeto não é naturalmente equivalente à chave primária valor. Com bancos de dados relacional, é recomendado o uso de um chave substituta, que é uma coluna de chave primária com nenhum significado para o usuário; .NET, mas têm uma identidade intrínseca que é ou com base em sua memória local ou em um user-defined convenção (usando a implementação do **Equals ()** método).

Com base neste problema, como é que vamos para representar as associações? Vejamos isso a seguir.

Problemas relacionados às associações

No nosso modelo de objeto, associações representam os relacionamentos entre os objetos. Por exemplo, uma oferta tem um relacionamento com um item. Esta associação é feita usando referências de objeto. No mundo relacional, um associação é representada como uma coluna de chave estrangeira, com cópias dos valores de chave em várias tabelas. Há sutis diferenças entre as duas representações.

Referências a objetos são inherentemente direcional; a associação é de um objeto para o outro. Se um associação entre objetos devem ser navegável em ambas as direções, você deve definir a associação duas vezes, uma vez em cada uma das classes associadas.

Por outro lado, as associações estrangeiras chave não são, por natureza direcional. Na verdade, a navegação não tem nenhum significado para um modelo de dados relacional, porque você pode criar associações arbitrárias de dados com junções de tabela e projeção.

Discutiremos mapeamentos associação em grande detalhe nos capítulos 3 e 6.

Se você pensar sobre DataSet em todos estes problemas, você vai perceber como a sua estrutura rígida é. O informações em um DataSet é apresentado exatamente como no banco de dados. Para navegar de uma linha para outra, você deve manualmente resolver seu relacionamento, isto é usar um chave estrangeira para encontrar a linha se refere o relacionadas

mesa. Vamos passar a partir da representação das entidades de como podem ser eficientemente manipulada.

Quando um usuário trabalha em um aplicativo, ele realiza diferentes operações unitárias. Estas operações podem ser encaminhados como conversas (Ou transações comerciais ou operações de aplicação). Por exemplo, a colocação de um lance em um item

É uma conversa. Programadores experientes sabem o quanto difícil pode ser para se certificar de que muitas operações relacionadas

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

executadas pelo usuário são tratados como se fossem uma única transação comercial maior (a unidade). Você vai aprender no Nesta seção NHibernate que torna muito mais fácil de alcançar. Vamos dar outro exemplo para ilustrar melhor essa conceito.

Jogadores de meios populares permitem a taxa de músicas que você ouve e depois classificá-los com base na sua classificação. Este significa que suas avaliações são persistentes. Quando você abre uma lista de músicas, então você ouvir e avaliá-los um por um.

Quando é a persistência suposto ter lugar?

A primeira solução, que pode vir à sua mente, é persistir a classificação quando é digitado pelo usuário. Mas isso é muito ineficiente: O usuário pode alterá-la muitas vezes e essa persistência será feita separadamente para cada canção. No entanto, esta abordagem é, certamente, mais seguro se você espera que esse aplicativo falhar a qualquer momento.

O que podemos fazer é deixar a taxa de usuário de todas as músicas da lista, e quando ele fecha-lo, então nós persiste todas as classificações.

O processo de avaliação dessas músicas pode ser chamado de uma conversa.

Vamos ver como funciona e quais os seus benefícios são.

O padrão de unidade de trabalho

Ao trabalhar com um banco de dados relacional, tendemos a pensar de comandos: guardar ou carregar. Mas um aplicativo pode realizar operações envolvendo muitas entidades. Quando essas entidades são carregados ou salvos depende da contexto.

Por exemplo, se você quiser carregar o último item criado por um usuário, você deve primeiro salvar este usuário (e seu coleção de itens), então você pode executar uma consulta recuperar esse item. Se você se esqueça de salvar o usuário, você vai começar

O padrão de Mapa de identidade

Um padrão chamado Mapa de identidade é usado por NHibernate para se certificar de que o usuário deste item é o mesmo objeto como o usuário que você tinha antes de colocar este item (desde que você está trabalhando na mesma transação). Você vai aprender mais sobre o conceito de identidade no capítulo 3, seção 3.5, "Identidade de objeto Entendimento".

Agora, imagine que você está envolvido em uma complexa conversa. Rastreamento manual de entidades para salvar ou excluir e certificando-se que você carregar cada entidade apenas uma vez pode ser um pesadelo.

A Unidade de Trabalho é um padrão seguido por NHibernate para resolver este problema e facilitar a implementação de conversas. Note-se que, cobrimos conversas no capítulo 5 e nós implementá-las no capítulo 10.

Você pode criar entidades e associá-los ao NHibernate, e então ele se mantém informado de todas as cargas e pode salvar qualquer mudança somente quando necessário. No final da transação, ele descobre e aplica todas as alterações no sua ordem correta.

Persistência transparente e carregamento lento

NHibernate porque mantém o controle de todas as entidades, que pode simplificar muito a sua aplicação e aumentar a sua desempenho. Aqui estão dois exemplos simples:

Ao trabalhar em um item do nosso aplicativo de leilão, um usuário pode adicionar, modificar ou excluir suas propostas. Seria doloroso para acompanhar essas mudanças manualmente, um por um. Em vez disso, podemos usar o recurso de chamada NHibernate persistência transparente: Você pode pedir NHibernate para salvar todas as alterações na cobrança de lances quando o item é persistiu. Ele irá determinar automaticamente quais as operações CRUD deve ser executado.

Agora, se você deseja modificar um usuário, você irá carregar, alterar e persisti-lo. Mas o que sobre a coleção de itens que este usuário tem? Se você carregar esses itens ou deixar a coleção não inicializada? Carregar os itens seria ineficiente, mas deixando a coleção não inicializada irá limitar a nossa capacidade de manipular o usuário.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

NHibernate suporta um recurso chamado carregamento lento para resolver este problema. Basicamente, quando o carregamento do usuário, você pode decidir entre o carregamento dos itens ou não. Se você optar por não fazê-lo, esta coleção será transparente inicializado quando você precisar dele.

Há muitas implicações em como usar esses recursos, vamos progressivamente cobri-los neste livro.

Caching

Entidades de monitoramento implica a manutenção das suas referências em algum lugar. NHibernate usa o que é chamado de cache. Este cache é indispensável para implementar o padrão de unidade de trabalho. No entanto, ele também pode tornar os aplicativos mais eficientes. Nós profundamente cobrir o conceito de caching no capítulo 5, secção 5.3 "teoria e prática Caching".

A cache é usada pelo mapa de identidade do NHibernate para evitar carregar uma entidade vezes muitos. E esse cache pode ser compartilhado por transações e até mesmo aplicativos.

Vamos dizer que construir um site para o nosso aplicativo de leilão. Nossos visitantes podem se interessar por alguns itens.

Sem cache, esses itens serão carregados a partir do banco de dados cada vez que um visitante quer vê-lo. Com poucos linhas, podemos pedir NHibernate para armazenar em cache esses itens e aproveitar o ganho de desempenho.

1.3.3 Consultas complexas e ADO.NET Entity Framework

Este é o último recurso, mas não menos relacionadas com a persistência. Na seção 1.2.5, falamos sobre CRUD operações. Nós descobrimos algumas características relacionadas com Create, Update e Delete (todos relacionados à Unidade de Trabalho padrão). Agora, nós estamos indo falar sobre as operações Retrieve, que é pesquisar e carregar informações.

Sabemos que podemos facilmente gerar o código para carregar uma entidade usando seu identificador (chave primária no contexto de banco de dados relacional). Mas em aplicações do mundo real, os usuários raramente lidar com identificadores, em vez disso eles usam alguns critérios para executar uma pesquisa e depois escolher a informação que querem.

Implementação de um mecanismo de consulta.

Se você estiver familiarizado com SQL, você sabe que você pode escrever consultas muito complexas usando o **SELECT ...**

DE ... ONDE ... construir. Mas se você trabalha com objetos de negócios, você teria, então, para transformar os resultados de suas consultas SQL em entidades. Nós já anunciamos os benefícios de trabalhar com entidades, por isso, poderia fazer mais sentido para aproveitar esses benefícios, mesmo quando consultar o banco de dados.

Baseado no fato de que NHibernate pode carregar e salvar entidades, podemos deduzir que ele sabe como cada entidade é mapeado para o banco de dados. Quando pedimos para uma entidade pelo seu identificador, NHibernate sabe como encontrá-lo. Assim, devemos ser capazes de expressar uma consulta usando nomes de entidades e propriedades, e depois NHibernate deve ser capaz de convertê-lo em uma consulta SQL correspondente entendido pelo banco de dados relacional.

NHibernate fornece duas APIs consulta: O primeiro é chamado de Hibernate Query Language. HQL é semelhante ao SQL, em muitos aspectos, mas também tem algumas útil orientada a objeto recursos. Note que você pode consultar NHibernate

usando SQL simples de idade, mas como você vai aprender, usando HQL oferece várias vantagens.

A API segunda consulta é chamada de Consulta por Critério API (QBC). Ele fornece um conjunto de tipo seguro classes para construir consultas no seu escolhido a linguagem. NET. Isto significa que se você estiver usando o Visual Studio, você vai se beneficiar de

o relatório de erros inline e Intellisense™!

Ofer^{Parcial} Para dar uma amostra do poder dessas APIs, vamos construir três consultas simples. Em primeiro lugar, aqui é algum HQL. Ele encontra todas as propostas para os itens, onde o nome vendedores começa com a letra K: Como você pode ver, é muito fácil de entender. Se você quiser escrever algumas SQL para fazer a mesma coisa, você precisa de algo um pouco mais detalhado, ao longo das linhas de:

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

select * B.
de B BID
inner join ITEM I em B. item_id = I. item_id
U USUÁRIO junção interna em I. author_id = U. USER_ID
onde U. NOME like '% K'

```

Para ilustrar o poder da consulta por Criteria API, vamos usar um exemplo derivado daquele mostrado na capítulo 8, seção 8.5.1. Isso mostra um método que nos permite localizar e carregar todos os usuários que são semelhantes a um

usuário exemplo, e que também tem um item lance semelhante a um item exemplo dado:

```

público IList <User> FindUsersWithSimilarBidItem (User u, ponto i) {
    = Exemplo exampleUser
    Example.Create (u) EnableLike (MatchMode.Anywhere);
    Exemplo exampleItem =
    Example.Create (i) EnableLike (MatchMode.Anywhere);
    getSession return () . CreateCriteria (typeof (User))
        . Add (exampleUser)
        . CreateCriteria ("Itens")
            . Add (exampleItem)
        . <User> List ();
}

```

Essencialmente, este método permite ao desenvolvedor passar objetos que representam o tipo de usuários, queremos NHibernate para localizar e carregar. Ela cria duas NHibernate **Exemplo** objetos e usa a API Criteria Query by para executar a consulta e recuperar uma lista de usuários. A noção de entidade exemplo (aqui, o usuário exemplo e exemplo

Item) é poderosa e elegante, como demonstrado abaixo:

```

Usuário u = new User ();
Inciso I = new Item ();
u.Name = "K";
i.State = ItemState.Active;
i.ApprovedBy = administratorUser;
Resultado <User> lista = FindUsersWithSimilarBidItem (u, i);

```

Aqui, estamos usando o nosso **FindUsersWithSimilarBidItem** método para recuperar nomes de usuários que contêm 'K' e que está vendendo um item lance ativo, o que também foi aprovado pelo administrador. Uma façanha para código tão pouco! Se você é novo para esta abordagem, você deve encontrá-lo um tanto inacreditável. Nem sequer tentar implementar esta consulta usando mão-codificado SQL.

Você pode aprender mais sobre as consultas nos capítulos 5 e 7. Se você não está totalmente satisfeito com essas APIs, você pode também querer ver novos desenvolvimentos futuros que permitem LINQ para ser usado com NHibernate.

ADO.NET Entity Framework

No momento da redação deste artigo, a Microsoft está trabalhando em sua Next-Generation Data Access tecnologia que introduz uma série de inovações interessantes e emocionantes. Você pode sentir que esta tecnologia em breve substituir NHibernate, mas isso é muito improvável. Vamos ver o porquê.

Talvez a característica mais excitante novo é um quadro de consulta poderosa codinome LINQ. LINQ estende seu idioma. favoritos NET para que você possa executar consultas contra vários tipos de fonte de dados, sem ter que incorporar seqüências de consulta em seu código. Portanto, ao consultar um banco de dados relacional, você pode fazer algo como isto:

```

// C # LINQ exemplo, olhar ma sem cordas!
Usuários de IEnumerable = u em Usuários
    onde u.Forename.StartsWith ("K")
        por ordem decrescente user.Forename
    selecione u;

```

Como você pode ver, as consultas são type-safe e permitir-lhe tirar proveito de muitos. Recursos de linguagem NET. Um aspecto chave do LINQ é que ela nos dá uma forma declarativa de trabalhar com dados, para que possamos expressar o que quer em termos simples, ao invés de digitar lotes de for-each loops. Além disso, podemos beneficiar de IDE útil

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

capacidades, tais como auto-realização e assistência parâmetro. Esta é uma grande vitória para todos. Porque LINQ é projetado para ser extensível, outras ferramentas, como NHibernate são capazes de integrar-se com esta tecnologia e beneficiar muito. No momento da publicação, há um trabalho muito promissor em andamento para um LINQ to Projeto NHibernate. No momento da escrita, Manning estão trabalhando na "LINQ em Ação" livro, soa muito interessante!

Microsoft também está trabalhando em um quadro atualmente denominado LINQ to Entities, que visa proporcionar desenvolvedores um framework de mapeamento objeto-relacional não completamente ao contrário NHibernate. Isto é como um bem passo em frente porque a Microsoft irá promover o DataSet com menos freqüência, e também começar a educar e promover os benefícios das ferramentas de mapeamento objeto-relacional. Outro projeto, chamado DataSet LINQ mais melhora muito a recursos de consulta de DataSet, mas não resolve muitas outras questões discutidas neste capítulo.

Todas estas tecnologias terão um pouco de tempo para amadurecer. Há ainda muitas perguntas não respondidas, como a elasticidade dessa estrutura será? Será que vai apoiar RDBMSs mais populares ou apenas o SQL Server? Será que vai ser fácil de trabalhar com esquemas de banco de dados legado? O fato é que nenhuma estrutura pode fornecer todos os recursos, por isso deve ser extensível para permitir a integrar seus recursos próprios.

Note que se você está projetos particulares exigem que você trabalhar com bancos de dados legados, você pode ler o Capítulo 10, seção 10.2 para aprender sobre o NHibernate características nos dá para trabalhar com estruturas mais exóticas de dados.

1.4 mapeamento objeto / relacional

Você já tem uma idéia de como NHibernate fornece objeto / relacional persistência. No entanto, você ainda pode ser incapaz de dizer o que mapeamento objeto / relacional (ORM) é. Vamos tentar responder a esta pergunta agora. Depois disso, nós irá discutir algumas razões não-técnicas a utilizar ORM.

O que é o ORM?

Tempo provou que bancos de dados relacionais fornecem um bom meio de armazenamento de dados, e que orientada a objeto programação é uma boa abordagem para a construção de aplicações complexas. Com mapeamento objeto / relacional, é possível criar uma camada de tradução que pode facilmente transformar objetos em dados relacionais e de volta. Como essa ponte vai manipular objetos, pode fornecer muitas das características que precisamos (como cache de transação, controle de concorrência). Tudo o que temos a fazer é fornecer informações sobre como mapear objetos para tabelas.

Resumidamente, mapeamento objeto / relacional é a persistência (e possivelmente transparente) automatizado de objetos em uma aplicação para as tabelas em um banco de dados relacional, usando metadata que descreve o mapeamento entre os objetos

eo banco de dados. ORM, em essência, obras de transformação de dados de uma representação para outra.

Não é um ORM Visio plug-in?

O ORM sigla também pode significar Object Role Modeling, e este termo foi inventado antes que mapeamento objeto / relacional tornou-se relevante. Ele descreve um método para análise de informação, utilizados em modelagem de banco de dados, e é sobretudo suportada pela Microsoft Visio, uma ferramenta de modelagem gráfica. Banco de dados especialistas usá-lo como um substituto ou como um complemento para os mais populares de modelagem entidade-relacionamento.

No entanto, se você falar com os desenvolvedores .NET sobre ORM, normalmente é no contexto do objeto / relacional mapeamento.

Aprendemos no ponto 1.3.1 que há muitos problemas para resolver ao usar ORM. Nós nos referimos a estes problemas como a incompatibilidade de paradigma. Vamos discutir, de um ponto não de vista técnico, por isso que devemos enfrentar este incompatibilidade e utilizar uma ferramenta ORM como NHibernate.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

1.4.1 Por que ORM?

A solução geral para esses problemas incompatibilidade pode exigir um dispêndio significativo de tempo e esforço. Em nossa experiência, o objetivo principal de até 30 por cento do código do aplicativo. .NET é escrito para lidar com a tediosa SQL / ADO.NET eo manual ponte do objeto / relacional incompatibilidade paradigma. Apesar de todo esse esforço, o resultado final ainda não se sente muito bem. Vimos projetos quase afundar devido à complexidade e inflexibilidade de suas camadas de abstração de banco de dados.

Incompatibilidade de modelagem

Um dos principais custos está na área de modelagem. Os modelos relacionais e objeto deve abranger tanto o mesmo entidades empresariais. No entanto, um purista orientada a objeto será modelo destas entidades de uma forma muito diferente do que um experiente modelador de dados relacionais. Você aprendeu alguns detalhes do problema no ponto 1.3.1. O de sempre solução para este problema é a dobrar e torcer o modelo de objeto até que ele corresponda a relacional subjacente tecnologia.

Isto pode ser feito com sucesso, mas somente à custa de perder algumas das vantagens da orientação a objetos. Tenha em mente que a modelagem relacional é sustentada pela teoria relacional. Orientação a objetos não tem tais definição matemática rigorosa ou corpo do trabalho teórico. Não há transformação elegante à espera de ser descoberto. (Acabar com. .NET e SQL e começar do zero não é considerado elegante.)

Produtividade e manutenção

O domínio de problema de incompatibilidade de modelagem não é o único problema resolvido por ORM. Uma ferramenta como o NHibernate torna você mais produtivo. Elimina grande parte do trabalho pesado (mais do que você esperaria) e permite que você concentrar no problema do negócio. Não importa qual a estratégia de desenvolvimento de aplicações você prefere-top-para baixo, começando com um modelo de domínio, ou de baixo para cima, começando com um banco de dados existente esquema NHibernate utilizado em conjunto com as ferramentas apropriadas irá reduzir significativamente o tempo de desenvolvimento.

Menos linhas de código torna o sistema mais compreensível, uma vez que enfatiza a lógica do negócio ao invés de encanamento. Mais importante, um sistema com menos código é mais fácil de refatorar. NHibernate melhora substancialmente manutenção. Não só porque reduz o número de linhas de código, mas também porque fornece um buffer entre o modelo de objeto ea representação relacional. Ele permite uma utilização mais elegante de orientação a objetos ao lado. .NET, e isola cada modelo de pequenas alterações para o outro.

Execução

A alegação comum é que a persistência mão-codificado pode sempre ser pelo menos tão rápido, e muitas vezes pode ser mais rápido do que persistência automatizada. Isso é verdade, no mesmo sentido que é verdade que o código de montagem sempre pode ser pelo menos tão rápido. .NET código, em outras palavras, é irrelevante.

A implicação tácita do pedido é que a persistência mão-codificado irá realizar pelo menos tão bem em um aplicação real. Mas essa implicação será verdadeiro somente se o esforço necessário para implementar pelo menos-como-rápido mão-codificado persistência é semelhante à quantidade de esforço envolvido na utilização de uma solução automatizada. O

questão realmente interessante é, o que acontece quando se considera restrições de tempo e orçamento?

A melhor maneira de abordar esta questão é definir meios para medir o desempenho e os limiares de aceitabilidade. Depois, você pode descobrir se o custo de desempenho de um ORM é inaceitável. A experiência tem provou que um bom ORM tem um impacto mínimo no desempenho. Ele pode até mesmo desempenho melhor do que clássico

.NET quando corretamente utilizado, devido a características como caching e lotes. NHibernate é baseado em uma madura arquitetura que permite que você para tirar proveito de muitas otimizações de desempenho com um mínimo de esforço.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Independência de banco de dados

NHibernate resumos sua aplicação fora do banco de dados SQL subjacente e dialeto SQL. O fato de que ele suporta um número de diferentes bancos de dados confere um certo nível de portabilidade em sua aplicação.

Você não deve necessariamente objectivo de escrever totalmente independente de banco de dados de aplicações, já que a capacidade de bases de dados diferentes e conseguir a portabilidade completa exigiria sacrificar um pouco da força dos mais plataformas poderosas. No entanto, um ORM pode ajudar a mitigar alguns dos riscos associados ao vendor lock-in. Além disso, a independência de banco de dados ajuda na cenários de desenvolvimento onde os desenvolvedores usar um local leve implantar banco de dados mas para a produção em um banco de dados diferente.

1.5 Resumo

Neste capítulo, discutimos o conceito de persistência do objeto e da importância de NHibernate como um técnica de implementação. Persistência do objeto significa que objetos individuais podem sobreviver ao processo de candidatura;

eles podem ser salvos em um armazenamento de dados e ser re-criado em um momento posterior no tempo. Temos caminhado através da camada arquitetura de um aplicativo. NET e da implementação de persistência, explorando três abordagens.

Entendemos a produtividade do DataSet, mas também perceber o quanto limitada e rígida que é. Temos descobriu muitas características úteis que seria doloroso para a mão-código. Além disso, sabemos como NHibernate resolve o objeto / relacional incompatibilidade.

Esse descompasso entra em jogo quando o armazenamento de dados SQL é um gerenciamento baseado em banco de dados relacional system (RDBMS). Por exemplo, um gráfico de objetos ricamente digitado não pode simplesmente ser salvos em uma tabela do banco de dados, que deve ser desmontada e persistiu até colunas de dados portáteis tipos SQL.

Temos olhou para as APIs de consulta poderosa do NHibernate. Depois que você começou a usar eles, você vai nunca mais querer voltar para SQL. Você também vai descobrir que NHibernate é projetado para suportar esotérico mapeamento e para ser extensível.

Finalmente, nós aprendemos quais mapeamento objeto / relacional (ORM) é, e nós discutimos, a partir de um ponto de não-técnicos de vista, as vantagens de usar essa abordagem.

ORM não é uma bala de prata para todas as tarefas persistência; seu trabalho é aliviar o desenvolvedor de 95 por cento do objeto trabalho de persistência, como escrever instruções SQL complexas com mesa de muitas associações e copiar valores de Resultado ADO.NET conjuntos de objetos ou de objetos gráficos. A ORM middleware completo como o NHibernate fornece portabilidade de banco de dados, técnicas de otimização como certos caching, e outras funções viável que não são fáceis de mão-código em um tempo limitado com o SQL e ADO.NET.

É provável que uma solução melhor do que ORM vai existir algum dia. Nós (e muitos outros) pode ter que repensar tudo que sabemos sobre SQL, as normas API de persistência e integração de aplicações. A evolução dos sistemas de hoje em verdadeiros sistemas de banco de dados relacional com integração orientada a objetos transparente permanece pura especulação. No entanto, não podemos esperar, e não há sinal de que algum desses problemas vai melhorar em breve (a multibilionária indústria não é muito ágil). ORM é a melhor solução disponível atualmente, e é um poderoso para os desenvolvedores de frente para o objeto / relacional incompatibilidade todos os dias.

Nós esperamos que lhe deu um grande fundo nas razões por trás OR / M, as questões críticas que devem ser abordados, e as ferramentas e abordagens disponíveis para .NET para enfrentá-los. Nós explicamos que NHibernate é uma ferramenta fantástica OR / M que permite que você combine os benefícios de ambos orientação a objetos bancos de dados relacionais simultaneamente. O próximo passo é dar-lhe um olhar muito mais hands-on em NHibernate, assim você pode ver como ele pode ser usado em seus próprios projetos. É aí que vem dentro capítulo 2

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2

Olá NHibernate!

Este capítulo aborda

NHibernate em ação com um "Olá Mundo"

Como a arquitetura de um aplicativo NHibernate

Escrita e mapeamento de uma entidade

simples

Configurando o NHibernate

Implementação de operações primitivas CRUD

É bom para entender a necessidade de mapeamento objeto / relacional in. NET, mas provavelmente você está ansioso NHibernate para ver em ação. Vamos começar mostrando-lhe um exemplo simples que demonstra pouco do seu poder.

Como você está provavelmente ciente, é tradicional para um livro de programação para começar com um "Olá Mundo" exemplo.

Neste capítulo, seguimos essa tradição introduzindo NHibernate com um relativamente simples "Olá Mundo" programa. No entanto, simplesmente imprimir uma mensagem para uma janela do console não será o suficiente para realmente demonstrar

NHibernate. Em vez disso, nosso programa irá armazenar objetos recém-criados no banco de dados, atualizá-los, e realizar consultas para recuperá-los do banco de dados.

Este capítulo servirá de base para os capítulos subsequentes. Além do canônico "Olá Mundo" exemplo, nós introduzimos o núcleo NHibernate APIs e explicar como configurar o NHibernate em tempo de execução diversos ambientes, tais como aplicações ASP.NET e WinForms stand-alone aplicações.

2.1 "Olá Mundo" com NHibernate

Aplicações NHibernate definir classes persistentes que são "mapeados" para as tabelas do banco de dados. Nossa "Olá Mundo"

exemplo consiste de uma classe e um arquivo de mapeamento. Vamos ver o que uma classe simples e persistente parece, como o

mapeamento é especificado, e algumas das coisas que podemos fazer com as instâncias da classe persistente utilizando NHibernate.

2.1.1 Instalando o NHibernate

Antes de podermos começar a programar o nosso "Olá Mundo", primeiro precisamos instalar NHibernate. Depois, temos que

criar um novo Visual Studio solução para conter a nossa aplicação de exemplo.

Você pode fazer o download do instalador do NHibernate <http://www.nhibernate.org>. O projeto é Java Hibernate também sediou neste site web, assim você terá que localizar o arquivo correto para download. No momento da escrita, o versão mais recente do Nhibernate pode ser instalado por baixar e executar um arquivo chamado NHibernate1.2.1.GA.msi.

Depois de ter baixado e instalado NHibernate, você está pronto para criar uma nova solução e começar a usá-lo.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.1.2 Criar um projeto novo Visual Studio

Para a nossa aplicação exemplo, você deve criar um novo projeto em branco com o Visual Studio. Este é um simples aplicação, por isso a coisa mais fácil de criar é um "aplicativo de console C #". Nome de seu projeto HelloWorldNHibernate ou similar. Note que você também pode usar NHibernate com projetos VB.NET, mas neste livro que escolhemos para usar exemplos c #.

Nossa aplicação terá de fazer uso da biblioteca NHibernate, por isso nosso próximo passo é fazer referência a ela em nosso projeto. Para fazer isso, clique direito sobre o projeto e selecione "Add Reference ...". Você deve, então, clique no Procurar guia e navegue até a pasta onde NHibernate instalado. Por padrão, NHibernate vive na **C:\ Program Files \ NHibernate \ bin \ net2.0 ** pasta. Depois de ter encontrado as assembléias, selecione o "NHibernate.dll" e "NHibernate.Mapping.Attributes.dll". Clicar ok irá adicionar estas referências para o seu solução.

Ser padrão, o aplicativo de console deve ter adicionado um arquivo chamado Program.cs à sua solução. Em aplicações console, esta será a primeira coisa que é executado quando você executar o programa.

Precisamos nos certificar de que a referência NHibernate no topo dos arquivos com o Programa **utilização NHibernate** e usando **NHibernate.Cfg** declarações, como mostrado abaixo.

```
utilização Sistema;
utilização System.Collections.Generic;
utilização System.Reflection;
utilização NHibernate;
utilização NHibernate.Cfg;

namespace HelloNHibernate
{
    Programa public class
    {
        static void Main ()
        {
            }
    }
}
```

Agora que temos a nossa solução criada, estamos prontos para começar a escrever a nossa primeira aplicação NHibernate.

2.1.3 Criando a classe Employee

O objetivo do nosso aplicativo de exemplo é armazenar uma **Empregado** registro em um banco de dados e depois recuperá-lo para mostrar. Nossa aplicação, portanto, precisa de uma classe simples e persistente, **Empregado**, O que representará uma pessoa que é empregado por uma empresa.

No Visual Studio, adicionar um novo arquivo de classe para sua aplicação, e nomeá-la Employee.cs quando solicitado. Então,

digite o seguinte código para a nossa entidade Funcionário. Isso é mostrado na listagem de 2,1.

```
Listagem 2.1 Employee.cs: Uma classe simples e persistente
namespace HelloWorldNHibernate
{
    classe Employee
    {
        id int público;
        nome de cadeia pública;
        gerente do empregado público;
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public string SayHello ()
{
    retorno string.Format (
        "!! Olá Mundo", disse {0}. ", Nome);
}
}

```

Nosso `Empregado` classe tem três campos: o campo identificador, o nome do empregado, e uma referência para gerente de empregado. O campo identificador permite que o aplicativo para acessar o banco de dados de identidade, o principal

valor de chave de um objeto persistente. Se duas instâncias de `Empregado` têm o mesmo valor identificador, eles representam

mesma linha no banco de dados. Nós escolhemos `int` para o tipo de identificador de nosso campo, mas esta não é uma exigência. NHibernate permite que praticamente qualquer coisa para o tipo de identificador, como você verá mais tarde.

Note que eu usei pública campos aqui em vez de propriedades. Isto é puramente para fazer o código de exemplo mais curtos, e nem sempre é considerada uma boa prática.

Instâncias do `Empregado` classe pode ser gerenciado (feita persistente) pelo NHibernate, mas eles não têm para ser. Desde que o objeto Employee não implementa qualquer classe NHibernate-específicas ou interfaces, podemos utilizar

-la como qualquer outra classe. NET.

```

Empregado fred = Employee new ();
fred.name = "Fred Bloggs";
Console.WriteLine (fred.SayHello ());

```

Este fragmento de código faz exatamente o que temos vindo a esperar do "Olá Mundo" aplicações: Ela imprime "Olá Mundo, disse Fred Bloggs "para o console Pode parecer que estamos tentando ser bonito aqui;. Na verdade, estamos demonstrando uma característica importante que distingue NHibernate de algumas soluções de persistência. Nosso classe persistente pode ser usado com ou sem NHibernate-sem requisitos especiais são necessários. Claro, você veio aqui para ver NHibernate-se, então vamos primeiro configurar o banco de dados e demonstrar usando NHibernate para salvar uma nova `Empregado` a ele.

2.1.4 Configurando o banco de dados

Precisamos ter um banco de dados criado para que NHibernate tem algum lugar para salvar entidades. Criação de um banco de dados para este programa deve levar apenas um minuto. NHibernate pode trabalhar com muitos bancos de dados, mas para este exemplo nós vamos usar o Microsoft SQL Server 2005.

Seu primeiro passo será abrir o Microsoft SQL Server Management Studio, conecte ao seu servidor de banco de dados e abrir uma nova janela de consulta. Digite o seguinte na janela de SQL para criar rapidamente uma nova base de dados.

```

CREATE DATABASE HelloNHibernate
GO

```

Executar esse SQL para criar o banco de dados. O próximo passo é mudar para o banco de dados, e criar uma tabela para armazenar os dados de nossos funcionários. Para fazer isso, exclua o SQL acima e substituí-lo com o seguinte.

```

USE HelloNHibernate
GO
CRIAR Employee TABLE (
    id int chave de identidade primária,

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
    nome varchar (50),
    int gerente)
GO
```

Executar esse código, e você tem criado agora um lugar para armazenar suas entidades de funcionários. Agora estamos prontos para ver
NHibernate in Action!

Note-se que, no capítulo 9, nós vamos mostrar-lhe como usar NHibernate para automaticamente criar as suas tabelas necessidades da sua aplicação apenas as informações nos arquivos de mapeamento. Há pouco mais de SQL que você não vai precisar escrever à mão!

2.1.3 Criando um empregado e salvar ao banco de dados

O código necessário para criar um Funcionário e salvá-lo ao banco de dados é mostrado abaixo. Este é composto por duas funções: `SaveEmployeeToDatabase` e `OpenSession`. Você pode digitar essas funções que em seu `Program.cs` arquivo, abaixo do `static void Main (...)` função na classe `Program`.

```
CreateEmployeeAndSaveToDatabase static void ()
{
    Empregado tobin = Employee new ();
    tobin.name = "Tobin Harris";

    usando (sessão ISession openSession = ())
    {
        usando (ITransaction transação session.BeginTransaction = ())
        {
            Session.save (Tobin);
            transaction.Commit ();
        }
        Console.WriteLine ("Saved Tobin ao banco de dados");
    }
}

openSession ISession estática ()
{
    Configuração c = new Configuration ();
    c.AddAssembly (Assembly.GetCallingAssembly ());
    ISessionFactory f = c.BuildSessionFactory ();
    f.OpenSession return ();
}
```

O `CreateEmployeeAndSaveToDatabase` função chama o NHibernate `Sessão` e `Transação` interfaces de. (Nós vamos chegar a isso `OpenSession ()` chamada em breve.) Nós não estamos completamente prontos para executar o código ainda, mas para lhe dar uma idéia do que iria acontecer, a execução do `CreateEmployeeAndSaveToDatabase` função irá resultar em algum SQL que está sendo executado nos bastidores pelo NHibernate:

```
inserir funcionários (gerente, nome)
valores (Tobin Harris ', null)
```

Espere-o `ID` coluna não está sendo inicializado aqui. Nós não definir o `id` campo de mensagem de qualquer lugar, assim como podemos esperar que para obter um valor? Na verdade, a `id` propriedade é especial: é um identificador de propriedade que detém um valor exclusivo gerado pelo banco de dados. Este valor gerado é atribuído ao `Empregado` por exemplo NHibernate durante a chamada para o `save ()` método.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Nós não vamos discutir o `OpenSession` função em profundidade, mas, essencialmente, ele configura e NHibernate retorna um `sessão` objeto que podemos usar para salvar, carregar e procurar objetos no nosso banco de dados (e muito mais!).

Não use esta função `openSession` em seus projetos de produção, como o seu vai aprender mais econômico abordagens ao longo deste livro.

Queremos "Olá Mundo" programa para imprimir a mensagem para o console. Agora que temos um `Empregado` em banco de dados, estamos prontos para demonstrar esse lado.

1.2.4 Carregando um funcionário do banco de dados

Ter escrito algum código para criar um Funcionário e salvá-los ao banco de dados, agora demonstrar algumas código que pode recuperar todos os `Funcionários` do banco de dados, em ordem alfabética. Digite este código em baixo da anterior `OpenSession()` função.

```
LoadEmployeesFromDatabase static void ()
{
    usando (nhibernateSession ISession openSession = ())
    {
        Consulta IQuery nhibernateSession.CreateQuery =
            "Empregado de como a ordem emp por emp.name asc";

        IList <Employee> foundEmployees = query.List <Employee> ();

        Console.WriteLine ("\n {0} empregados encontrado:", foundEmployees.Count);

        foreach (empregado Empregado em foundEmployees)
        {
            Console.WriteLine (employee.SayHello ());
        }
    }
}
```

A seqüência de caracteres literal "`Empregado de como a ordem emp por emp.name asc`" é uma consulta NHibernate, expressa na própria NHibernate Query Language orientada a objetos Hibernate (HQL). Esta consulta é internamente traduzidos para o SQL seguinte quando `query.List()` chama-se:

```
selecionar e.id, e.name, e.manager
e de Empregado
por fim e.name asc
```

Se você nunca usou uma ferramenta de ORM como NHibernate antes, você estava provavelmente esperando para ver o SQL declarações em algum lugar no código ou metadados. Eles não estão lá. Todos SQL é gerado em tempo de execução (na verdade em inicialização sempre que possível).

Até agora temos definida nossa entidade Funcionário, configure o banco de dados, e escrito algum código para criar um novo empregado. NHibernate mal entrou em cena, no entanto, tão próxima que irá dizer sobre a nossa NHibernate Employee entidade e como queremos que eles salvos no banco de dados.

1.2.5 Criando um arquivo de mapeamento

Para permitir que NHibernate para fazê-lo é mágica em qualquer um dos códigos obove, ele primeiro precisa de mais informações sobre como o `Empregado` classe deve ser feita persistente. Esta informação é normalmente fornecido em um mapeamento XML documento. O documento de mapeamento define, entre outras coisas, como as propriedades do `Empregado` mapa de classes de colunas da `Funcionários` mesa. Vamos olhar para o documento de mapeamento na listagem 2.2.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Para adicionar um documento de mapeamento para a sua solução, você pode simplesmente adicionar um novo documento XML e chamá-lo

Employee.hbm.xml. Em seguida, selecione o arquivo no Solution Explorer, e olhar para a propriedade denominada "Construir"

Ação "no painel de propriedades. Mudar isto para "Recurso Incorporado". Este é um passo importante e não deve ser desperdiçada, já que permite NHibernate para encontrar as informações de mapeamento.

Listagem 2.2?? Um simples Hibernate mapeamento XML

```
<? Xml version = "1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" auto-import="true">
<class name="HelloNHibernate.Employee, HelloNHibernate" lazy="false">
  <id name="id" access="field">
    <generator class="native" />
  </Id>
  <property name="nome" access="field" column="name"/>
  <many-to-one access="field" name="manager" column="manager" cascade="all"/>
</ Class>

</ Hibernate-mapping>
```

O documento de mapeamento diz que o NHibernate **Empregado** classe deve ser mantido para a **Funcionários** mesa, que o **id** mapas de campo para uma coluna chamada **id**, Que o **nome** mapas de campo para uma coluna chamada **nome**, E que o propriedade denominada **gerente** é uma associação com muitos-para-um multiplicidade que mapeia para uma coluna chamada

De gerente de. (Não se preocupe com os outros detalhes por enquanto.)

Como você pode ver, o documento XML não é difícil de entender. Você pode facilmente escrever e mantê-la por mão. No capítulo 3, discutimos uma forma de gerar o arquivo XML a partir de comentários embutido no código-fonte. Independentemente do método escolhido, NHibernate tem informação suficiente para gerar todas as completamente SQL declarações que seriam necessários para inserir, atualizar, excluir e recuperar instâncias da **Empregado** classe. Você não precisa mais escrever essas instruções SQL à mão.

NOTA

NHibernate tem padrões sensíveis que minimizam digitação e uma definição de tipo maduro documento que pode ser usado para auto-realização ou a validação em editores, incluindo o Visual Studio. Você pode até mesmo automaticamente gerar metadados com várias ferramentas.

Enquanto estamos no assunto de XML, agora seria um bom momento para mostrar-lhe como configurar NHibernate.

1.2.6 Configurando sua aplicação

Se você criou .NET que usam **DataSets** ou **DataReaders** para se conectar a um banco de dados, você pode também estar familiarizado com o conceito de armazenar uma **ConnectionString** em seu web.config ou arquivos app.config. Configurando o NHibernate é similar, nós apenas adicionar algumas informações de conexão no arquivo de configuração.

Comece clicando com o seu projeto HelloNHibernate no Solution Explorer e selecionando Add -> New Item ... Em seguida, selecione "Application Configuration File" a partir das opções. Hit ok, e isso vai adicionar um app.config arquivo para o projeto.

Então você deve copiar o XML a seguir em seu arquivo.

```
<? Xml version = "1.0" encoding = "utf-8"?>
<configuration>
<configSections>
<Nome da seção = "nhibernate"
  type = System.Configuration.NameValueSectionHandler ", Sistema
Version = 1.0.3300.0, Culture = neutral, PublicKeyToken = b77a5c561934e089 "
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        />
</ConfigSections>
<nhibernate>
<Add key = "hibernate.show_sql"
      value = "false" />
<Add key = "hibernate.connection.provider"
      value = "NHibernate.Connection.DriverConnectionProvider" />
<Add key = "hibernate.dialect"
      value = "NHibernate.Dialect.MsSql2000Dialect" />
<Add key = "hibernate.connection.driver_class"
      value = "NHibernate.Driver.SqlClientDriver" />
<Add key = "hibernate.connection.connection_string"
      value = "Data Source = 127.0.0.1; Initial Catalog = HelloNHibernate; Integrado
Security = SSPI; "/>
</nhibernate>

</Configuration>

```

Há bastante dele! Mas, lembre-se NHibernate é muito flexível e pode ser configurado de várias maneiras. Note que você pode precisar alterar o `hibernate.connection.connection_string` chave na parte inferior do XML para se conectar ao servidor de banco de dados no computador de desenvolvimento.

1.2.7 Atualizando um empregado

Antes de executar qualquer código, vamos adicionar mais uma função para demonstrar como NHibernate pode atualizar existentes

entidades. Vamos escrever um código para atualizar o nosso primeiro `Empregado` e, enquanto estamos no assunto, criar um novo `Empregado`
para ser o gerente do primeiro, como mostrado na listagem 2.3. Novamente, você deve digitar este abaixo as outras funções

~~listagem 2.3~~ Atualizando um empregado

```

UpdateTobinAndAssignPierreAsManager static void ()
{
    usando (sessão ISession openSession = ())
    {
        usando (ITransaction transação session.BeginTransaction = ())
        {
            IQuery session.CreateQuery q =
                "Do empregado quando name = 'Tobin Harris'";

            Empregado tobin = q.List <Employee> () [0];
            tobin.name = "Tobin David Harris";

            Empregado pierre = Employee new ();
            pierre.name = "Pierre Henri Kuate";

            tobin.manager = pierre;
            Session.flush ();
            transaction.Commit ();

            Console.WriteLine ("Atualizado Tobin e acrescentou Pierre");
        }
    }
}

```

Nos bastidores, NHibernate iria executar quatro instruções SQL dentro da mesma transação:

selecionar e.id, e.name, e.manager
e de Empregado
onde e.id = 1

inserir funcionários (gerente, nome)

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
values ('Pierre Henri Kuate', null)

declare @ newId int
select @ newId SCOPE_IDENTITY = ()
```

```
Os funcionários de atualização
set nome = 'Tobin David Harris, gerente = @ newId
onde id = 1
```

Observe como NHibernate detectou a modificação para o `nome` e `gerente` propriedades do primeiro trabalhador (Fred) e automaticamente atualizado banco de dados. Temos tido proveito de um recurso chamado NHibernate checagem suja automática: Esta funcionalidade poupa-nos o esforço de explicitamente pedir para atualizar o NHibernate banco de dados quando modificar o estado de um objeto. Da mesma forma, você pode ver que o novo `Empregado` (Bill) foi salvo quando ele foi associado com o primeiro `Empregado`. Este recurso é chamado salvar em cascata: Ele salva-nos a esforço de explicitamente tornando o novo objeto persistente chamando `Save()`. Contanto que ele é acessível por uma já objeto persistente (Bill). Notar também que a ordem das instruções SQL não é o mesmo que a ordem em que vamos definir os campos do objeto. NHibernate usa um algoritmo sofisticado para determinar uma ordenação eficiente que evita banco de dados de violações de restrição de chave estrangeira, mas ainda é suficientemente previsível para o usuário. Este recurso é chamado transacional write-behind.

1.2.8 - A execução do Programa

Antes de finalmente executar o nosso exemplo, precisamos escrever um código para executar todas essas funções na ordem certa.

Modificar o seu método de `Program.cs` principal para ficar assim:

```
static void Main ()
{
    CreateEmployeeAndSaveToDatabase ();
    UpdateTobinAndAssignPierreAsManager ();
    LoadEmployeesFromDatabase ();

    Console.WriteLine ("Pressione qualquer tecla para sair ...");
    Console.ReadKey ();
}
```

Se corremos "Olá Mundo", ela imprime

```
Salvo Tobin ao banco de dados
Atualizado Tobin e acrescentou Pierre
```

```
2 funcionários encontrados:
'Olá Mundo! ', Disse Pierre Henri Kuate.
'Olá Mundo! ', Disse Tobin David Harris.
Pressione qualquer tecla para sair ...
```

Isto é, tanto quanto nós vamos pegar o "Olá Mundo". Agora que finalmente temos um código sob nossa correia, vamos dar um passo atrás e apresentar uma visão geral das principais APIs do Hibernate.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.2 Entendendo a Arquitetura

As interfaces de programação são a primeira coisa que você tem que aprender sobre NHibernate, a fim de usá-lo no camada de persistência de sua aplicação. Um dos principais objetivos do projeto API é manter as interfaces entre componentes de software tão estreita quanto possível. Na prática, porém, ORM APIs não são especialmente pequenas. Não

se preocupe, você não tem que entender todas as interfaces NHibernate de uma vez.

Figura 2.1 ilustra o papel dos mais importantes interfaces NHibernate no negócio e persistência camadas. Nós mostramos a camada de negócios acima da camada de persistência, já que a camada de negócios funciona como um cliente do camada de persistência em uma aplicação tradicional em camadas. Note que algumas aplicações simples pode não limpa lógica de negócios separada da lógica de persistência, isto é bem-lo apenas simplifica o diagrama.

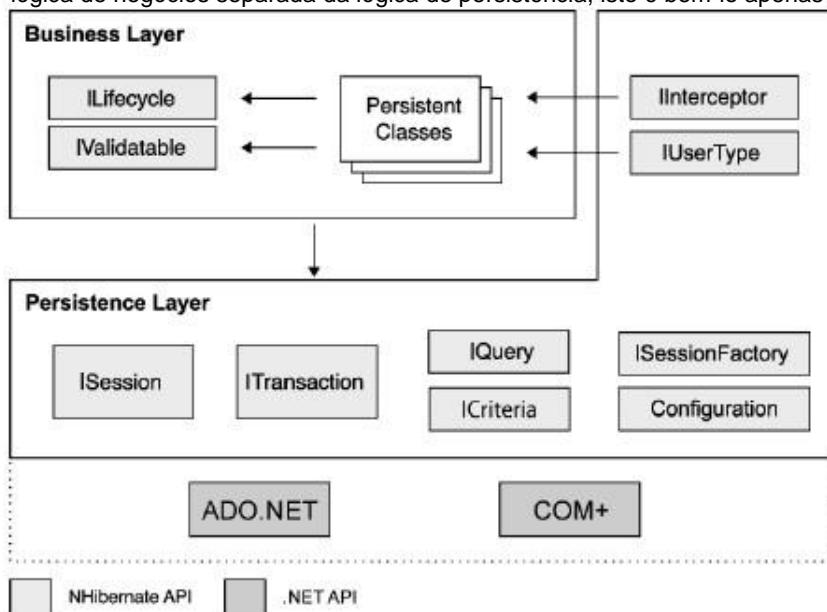


Figura 2.1 Visão geral de alto nível da API do NHibernate em uma arquitetura em camadas

As interfaces NHibernate mostrado na figura 2.1 podem ser de aproximadamente classificados como segue:

Interfaces chamado por aplicativos para realizar CRUD e operações básicas da consulta (isto é: Criar, Retrieve, Update e Delete). Estas interfaces são o ponto principal de dependência de aplicação negócio / lógica de controle sobre NHibernate. Eles incluem **ISession**, **ITransaction**, **IQuery** e **ICriteria**.

Interfaces chamado pelo código de infra-estrutura de aplicativos para configurar NHibernate, que é mais importante do

Configuração classe.

Interfaces de retorno de chamada que permitem a aplicação de reagir a eventos que ocorrem dentro NHibernate, como

IInterceptor, **ILifecycle** E **IValidatable**.

Interfaces que permitem a extensão da funcionalidade do NHibernate mapeamento poderosos, como **IUserType**, **IComposite UserType** E **IIdentifierGenerator**. Estas interfaces são implementadas por código da aplicação infra-estrutura (se necessário).

NHibernate faz uso de APIs existentes .NET, incluindo ADO.NET e seus **ITransaction** API. ADO.NET fornece um nível rudimentar de captação de funcionalidade comum para bancos de dados relacionais, permitindo que quase

qualquer banco de dados com um driver ADO.NET a ser suportado pelo NHibernate.

Nesta seção, nós não cobrem a semântica detalhada dos métodos NHibernate API, apenas o papel de cada um as interfaces primária. Nós vamos cobrir progressivamente métodos API nos próximos capítulos. Você pode encontrar uma completa

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

e sucinta descrição dessas interfaces na documentação do NHibernate referência. Vamos dar uma breve olhada cada interface, por sua vez.

2.2.1 As interfaces do núcleo

Os cinco principais interfaces estão listados abaixo e utilizado em praticamente todas as aplicações NHibernate. Com estes interfaces, você pode armazenar e recuperar objetos persistentes e operações de controle.

Interface de ISession

O **ISession** interface é a interface primária usada por aplicativos NHibernate, ele expõe NHibernates métodos para encontrar, salvar, atualizar e excluir objetos. Uma instância de **ISession** é leve e é barato para criar e destruir. Isto é importante porque a sua aplicação terá de criar e destruir sessões de todo o tempo, talvez em cada solicitação de página ASP.NET. NHibernate sessões não são Thread seguro e pelo projeto deve ser usado por apenas um segmento de cada vez. Isto é discutido em mais detalhes nos próximos capítulos.

A noção de um NHibernate sessão é algo entre conexão e transação. Pode ser mais fácil pensar em uma sessão como um cache ou coleção de objetos carregados relativas a uma única unidade de trabalho. NHibernate pode

detectar alterações aos objetos nesta unidade de trabalho. Às vezes chamamos a **ISession** um persistência gerente porque é também a interface para persistência operações relacionadas, tais como armazenar e recuperar objetos. Nota que uma sessão NHibernate tem nada a ver com uma sessão ASP.NET. Quando usamos a palavra sessão nesta livro, queremos dizer a sessão NHibernate.

Descrevemos a **ISession** interface em detalhe no capítulo 4, seção 4.2, "O gerente de persistência."

ISessionFactory interface de

O aplicativo obtém **ISession** instâncias de um **ISessionFactory**. Comparação com o **ISession** interface, este objeto é muito menos emocionante.

O **ISessionFactory** certamente não é leve! É destinado a ser compartilhado entre muitas aplicações threads. Existe normalmente uma única instância **ISessionFactory** para o conjunto de aplicativos criados durante inicialização do aplicativo, por exemplo. No entanto, se seu aplicativo acessa vários bancos de dados usando NHibernate, você precisará de um **SessionFactory** para cada banco de dados.

O **SessionFactory** caches gerado instruções SQL e outros metadados de mapeamento que NHibernate usa em tempo de execução. Ele também pode armazenar dados em cache que foi lido em uma unidade de trabalho, e que podem ser reutilizados em uma futura unidade de trabalho ou sessão. Isto é possível se você configurar mapeamentos de classe e coleção para usar o cache de segundo nível.

Interface de configuração

O **Configuração** objeto é usado para configurar o NHibernate. O aplicativo usa um **Configuração** exemplo, para especificar a localização de documentos, mapeamento e NHibernate para definir propriedades específicas antes de a criação do **ISessionFactory**.

Mesmo que o **Configuração** de interface desempenha um papel relativamente pequeno no total de um escopo Aplicação NHibernate, é o primeiro objeto que você vai encontrar quando você começar a usar o NHibernate. Seção 2.2 cobre o problema de configuração do NHibernate em algum detalhe.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Interface de ITransaction

O **ITransaction** interface é mostrado na figura 2.1, próxima à interface **ISession**. O **ITransaction** interface é uma API opcional. Aplicações NHibernate pode optar por não usar essa interface, em vez de gestão transações em seu código de infra-estrutura própria. A NHibernate **ITransaction** resumos código da aplicação de implementação da operação subjacente, que pode ser uma transação ADO.NET ou qualquer espécie de manual transação permitindo que o aplicativo para controlar os limites de transação através de uma API consistente. Isso ajuda a manter

Aplicações NHibernate portáteis entre diferentes tipos de ambientes de execução e contêineres.

Usamos o NHibernate **ITransaction** API ao longo deste livro. Transações e os **ITransaction** interface são explicadas no capítulo 5.

Interfaces de IQuery e ICriteria

O **IQuery** interface oferece maneiras poderosas de realizar consultas no banco de dados, ao mesmo tempo controlar a forma como a consulta é executada. É a interface básica usada para buscar dados usando NHibernate. Consultas

são escritos em HQL ou no dialeto SQL nativo de seu banco de dados. Um **IQuery** exemplo, é leve e não pode ser utilizado fora das **ISession** que o criou. Ele é usado para vincular os parâmetros de consulta, limitar o número de resultados

retornadas pela consulta e, finalmente, para executar a consulta.

O **ICriteria** interface é muito semelhante, que lhe permite criar e executar orientada a objetos critérios consultas.

Para ajudar a tornar o código do aplicativo menos detalhada, NHibernate fornece alguns métodos de atalho úteis no **ISession** interface que permite que você chamar uma consulta em uma linha de código. Por trás das cenas ele está usando o **IQuery**

interface. Nós não vamos usar esses atalhos no livro, em vez disso, vamos usar sempre o **IQuery** interface.

Nós descrevemos as características do **IQuery** interface no capítulo 7, onde você vai aprender como usá-lo na sua aplicações. Agora que nós introduzimos o principal APIs necessárias para escrever no mundo real NHibernate aplicações, a próxima seção apresenta alguns recursos mais avançados. Depois disso nós vamos mergulhar como NHibernate está configurado, e também como você pode configurar o log para ver o que está fazendo NHibernate por trás da cenas (uma ótima maneira de ver NHibernate em ação, se você desculpa o trocadilho!)

2.2.2 interfaces de Callback

Interfaces de callback permite que o aplicativo para receber uma notificação quando algo interessante acontece com um objeto, por exemplo, quando um objeto é carregado, salvo ou excluído. Aplicações NHibernate não precisa implementar esses callbacks, mas eles são úteis para a implementação de determinados tipos de funções genéricas, tais como criação de registros de auditoria.

O **ILifecycle** e **IVlatable** interfaces permitem um objeto persistente para reagir a eventos relacionados com seus próprios persistência do ciclo de vida. O ciclo de vida persistência é englobado pelas operações de um objeto CRUD (quando ele é criado, recuperado, atualizado ou excluído).

Nota: A equipe Hibernate foi fortemente influenciado por outras soluções ORM que callback similares interfaces. Mais tarde, eles perceberam que ter as classes persistentes implementar Hibernate interfaces específicas provavelmente não é uma boa idéia, porque isso polui o nosso classes persistentes com os não-portáteis de código. Uma vez que estes interfaces são obsoletos, não discuti-los neste livro.

O **IInterceptor** interface foi introduzido para permitir que o aplicativo processe sem callbacks forçando as classes persistentes para implementar NHibernate APIs específicas. Implementações do **IInterceptor** interface são passadas para as instâncias persistentes como parâmetros. Vamos discutir um exemplo no capítulo 8.

2.2.3 Tipos

Um elemento fundamental e muito poderoso da arquitetura é a noção do NHibernate de um **Tipo**. A NHibernate **Tipo** mapas de objetos tipo. NET para um tipo de coluna do banco de dados (na verdade, o tipo pode abranger várias colunas). Todos propriedades persistentes das classes persistentes, incluindo as associações, têm um tipo correspondente NHibernate. Este projeto faz NHibernate extremamente flexível e extensível que cada RDBMS tem um conjunto diferente de mapeamento para .NET.

Há uma vasta gama de built-in tipos, cobrindo todos. Primitivas NET e muitas classes CLR, incluindo tipos de **System.DateTime**, **System.Enum**, **byte**]E Classes serializáveis.

Melhor ainda, NHibernate suporta definidos pelo usuário tipos personalizados. As interfaces **IUserType**, **ICompositeUserType** e **IParameterizedType** são fornecidos para permitir que você adicione seus próprios tipos e **IUserCollectionType** para os tipos de sua coleção. Você pode usar esse recurso para permitir que comumente usados classes de aplicação, tais como **Endereço**, **Nome** Ou **MonetaryAmount** a ser tratado convenientemente e elegante. Tipos personalizados são considerados uma característica central do NHibernate, e você é encorajado a colocá-los para novas e usos criativos!

Explicamos tipos NHibernate e tipos definidos pelo usuário no capítulo 6, seção 6.1, "Entendendo o NHibernate tipo de sistema. "Nós vamos agora passar a lista de algumas das interfaces mais baixo nível. Você pode não precisar usar ou compreender todas essas, mas sabendo que existe pode dar-lhe uma maior flexibilidade quando se trata de projetar suas aplicações.

2.2.4 interfaces de Extensão

Grande parte da funcionalidade que fornece NHibernate é configurável, permitindo que você escolha entre certas built-in estratégias. Quando as estratégias built-in são insuficientes, NHibernate normalmente permitem que você ligar o seu implementação personalizada própria através da implementação de uma interface. Pontos de extensão são:

Geração de chave primária (**IIdentifierGenerator** interface)

SQL suporte dialeto (**Dialecto** classe abstrata)

Cache estratégias (**ICACHE** e **ICacheProvider** interfaces)

ADO.NET gerenciamento de conexão (**IConnectionProvider** interface)

Operação de gestão (**ITransactionFactory** e **ITransaction** interfaces)

Estratégias de ORM (**IClassPersister** hierarquia de interface)

Propriedade estratégias de acesso (**IPropertyAccessor** interface)

Criação de proxy (**IProxyFactory** interface)

Navios NHibernate com pelo menos uma implementação de cada uma das interfaces mencionadas, para que você geralmente não precisa

começar do zero se quiser estender a funcionalidade built-in. O código fonte está disponível para você usar como um exemplo para sua própria implementação.

Agora você deve ter a consciência de várias APIs e interfaces que NHibernate nos dá. Felizmente você não vai precisar deles todos! Na verdade, para aplicações simples que você pode precisar apenas a configuração e **ISession** interfaces, como mostrado no nosso "Olá Mundo" exemplo. No entanto, antes que você possa começar a usar o NHibernate na sua próprios aplicativos, você precisa ter algum entendimento de como NHibernate está configurado. É isso que nós discutiremos a seguir.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.3 configuração básica

NHibernate pode ser configurado para ser executado em praticamente qualquer aplicação .NET e ambiente de desenvolvimento. Geralmente, NHibernate é usado em duas e três camadas aplicações cliente / servidor, com NHibernate implantado apenas na servidor. A aplicação cliente é geralmente um navegador web, mas aplicações de cliente do Windows não são incomuns quer. Embora nós nos concentramos em aplicações multi-camadas web neste livro, nós cobriremos o Windows aplicações quando necessário.

A primeira coisa que você deve fazer é começar a NHibernate. Na prática, isso é muito fácil: Você tem que criar uma `ISessionFactory` exemplo, de um `Configuração`.

2.3.1 Criando um SessionFactory

A fim de criar uma `ISessionFactory` exemplo, você primeiro criar uma instância única de `Configuração` durante a inicialização do aplicativo e usá-lo para definir o acesso de dados e informações de mapeamento. Uma vez configurado, o `Configuração` instância é usado para criar o `SessionFactory`. Após o `SessionFactory` é criado, você pode descartar a `Configuração` classe.

Em nossas amostras anteriores, foi utilizado um `MySessionFactory` propriedade estática para criar `ISession` instâncias. Aqui está a sua implementação:

```
private static ISessionFactory sessionFactory = null;
public static ISessionFactory MySessionFactory
{
    obter
    {
        if (sessionFactory == null)
        {
            Cfg configuração Configuração = new ();
            cfg.Configure ();
            cfg.AddInputStream (
                HbmSerializer.Default.Serialize (typeof (Employee)));
            // OR: cfg.AddXmlFile ("Employee.hbm.xml");
            sessionFactory cfg.BuildSessionFactory = ();
        }
        retorno sessionFactory;
    }
}

# 1 feito apenas uma vez (no primeiro acesso)
# 2 Ao usar NHibernate.Mapping.Attributes
# 3 Quando usando um arquivo de mapeamento XML
```

A localização do arquivo de mapeamento, `Employee.hbm.xml`, É relativo ao diretório do aplicativo atual. Nesta exemplo, podemos também usar um arquivo XML para definir todas as outras opções de configuração (que pode ter sido definido antes por código da aplicação ou no arquivo de configuração do aplicativo).

Encadeamento de

método

Encadeamento de método é um estilo de programação suportado por muitas interfaces NHibernate (eles também são chamado interfaces fluentes). Este estilo é mais popular em Smalltalk que em .NET e é considerado por algumas pessoas a serem menos legível e mais difícil de debug do que o mais aceito estilo .NET. No entanto, é muito conveniente na maioria dos casos.

A maioria dos desenvolvedores .NET declarar métodos setter ou somador para ser do tipo `vazio`, o que significa que não retornam

valor. Em Smalltalk, que não tem `vazio` métodos setter tipo, ou somador geralmente voltam a receber objeto. Isto nos permitirá reescrever o exemplo de código anterior da seguinte forma:

Código a seguir é parte da barra lateral

```
ISessionFactory sessionFactory = new Configuration ()
    .Configure ()
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
.AddXmlFile ("Employee.hbm.xml")
.BuildSessionFactory ()
```

Repare que nós não precisamos declarar uma variável local para o **Configuração**. Usamos este estilo em alguns exemplos de código, mas se você não gostar, você não precisa usá-lo sozinho. Se você fazer usar esta codificação estilo, é melhor escrever cada invocação de método em uma linha diferente. Caso contrário, pode ser difícil percorrer o código em seu depurador.

Por convenção, arquivos de mapeamento XML NHibernate são nomeados com o **.Hbm.xml** extensão. Outra convenção é ter um arquivo de mapeamento por classe, ao invés de ter todos os seus mapeamentos listados em um arquivo (o que é possível mas considerada um estilo ruim). Nosso "Olá Mundo" exemplo tinha apenas uma classe persistente. Mas vamos supor que temos múltiplas classes persistentes, com um arquivo de mapeamento XML para cada um. Onde devemos colocar esses arquivos de mapeamento?

A documentação NHibernate recomenda que o arquivo de mapeamento para cada classe persistente é colocado no mesmo diretório que o arquivo de classe. Por exemplo, o arquivo de mapeamento para a **Empregado** classe seria colocado em um arquivo chamado **Employee.hbm.xml** no mesmo diretório que o arquivo **Employee.cs**. Se tivéssemos um outro classe persistente, seria definida em seu arquivo próprio mapeamento. Sugerimos que você siga esta prática e que você carregar vários arquivos de mapeamento chamando **AddXmlFile ()**.

É ainda possível incorporar arquivos de mapeamento XML dentro assembléias (sua ou compiled.dll. Exe). Você só tem que dizer ao compilador que cada um desses arquivos é uma **Recurso Incorporado**, A maioria das IDEs permitem que você

especificar essa opção. Depois, você pode usar o método **AddClass ()**, Passando um tipo da classe como parâmetro:

```
ISessionFactory sessionFactory = new Configuration ()
    .Configure ()
    .AddClass (typeof (Model.Item))
    .AddClass (typeof (Model.User))
    .AddClass (typeof (Model.Bid))
    .BuildSessionFactory ();
```

O **AddClass ()** método assume que o nome do arquivo de mapeamento termina com a **.Hbm.xml** extensão e é incorporado no mesmo assembly como o arquivo de classe mapeada.

Se você quiser adicionar todas as classes mapeadas (com. Atributos.NET) em uma montagem, você pode usar uma sobrecarga de **o método HbmSerializer.Serialize ()**, Ou se você quiser adicionar todos os arquivos de mapeamento incorporado em um montagem, você pode usar o método **AddAssembly ()**:

```
ISessionFactory sessionFactory = new Configuration ()
    .Configure ()
    .AddInputStream ( / . Atributos.NET
        HbmSerializer.Default.Serialize (typeof (Model.Item) da Assembléia.))
    .AddAssembly (typeof (Model.Item) da Assembléia.) // XML
    .BuildSessionFactory ();
```

Note que é propenso a erros de usar nomes de assembléias "(como **"NHibernate.Auction"**). É por isso que usamos um tipo de classe para recuperar directamente o conjunto que contém os arquivos de mapeamento integrado.

Por NHibernate diz que ele não sabe minha classe?

Um problema comum quando começar a utilizar NHibernate é garantir que todos os seus mapeamentos são enviados para

NHibernate, se você perder um, você receberá uma exceção. Ao construir a fábrica de sessão, que será um **MappingException** com um comentário que contenha "... Refere-se a uma classe mapeada: YourClass". Quando execução de uma consulta, que será um **QueryException** com um comentário como "Possivelmente uma ou inválido nome da classe mapeada foi usado na consulta".

Para resolver este problema, o primeiro passo é definir **log4net** para o nível INFO (você vai aprender como fazer isso em seção 3.3.2). Então, leia o log para certificar-se que NHibernate ler seus mapeamentos, você deve encontrar um mensagem como "Classe Mapping: Namespace.YourClass -> YourClass"). Se não for o caso, então verifique seu código de inicialização para se certificar de que você realmente enviar os mapeamentos.

Se você usar **AddAssembly ()**, certifique-se que eles são realmente incorporado em seu conjunto.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Por outro lado, você pode obter um `DuplicateMappingException` se você adicionar um mapeamento de muitos vezes. Por exemplo, evitar a adição de ambos os XML eo mapeamento atributos-based.

Nós demonstramos a criação de um único `SessionFactory`, Que é tudo o que a maioria das aplicações precisa. Se outro `ISessionFactory` instância é necessária, no caso de vários bancos de dados, por exemplo, repetir o processo. Cada `SessionFactory` é então disponível para um banco de dados e pronto para produzir `ISession` instâncias para trabalhar com esse banco de dados específico e um conjunto de mapeamentos de classes. Uma vez que temos a nossa `SessionFactory`, podemos ir para criar sessões, e começar a carregar e salvar objetos.

Claro, não há mais a configuração do NHibernate que apenas apontando para documentos de mapeamento. Você também necessidade de especificar como as conexões de banco de dados devem ser obtidos, junto com vários outros ajustes que afetam o comportamento do NHibernate em tempo de execução. A multidão de propriedades de configuração pode parecer esmagadora (a lista completa aparece na documentação NHibernate), mas não se preocupe, a maioria define padrão razoável valores, e apenas um punhado são comumente necessários.

Para especificar opções de configuração, você pode usar qualquer das seguintes técnicas:

Passar uma instância de `System.Collections.IDictionary` para `Configuration.setProperties()` ou uso `Configuration.setProperty()` para cada propriedade (ou manipular a coleção `Configuration.Properties` diretamente).

Defina todas as propriedades no arquivo de configuração do aplicativo (que é `App.config` ou `Web.config`).

Incluir `<property>` elementos em um arquivo XML chamado `hibernate.cfg.xml` no diretório atual.

A primeira opção é raramente usada, exceto para o teste rápido e protótipos, mas a maioria das aplicações precisa de um fixo arquivo de configuração. Tanto o arquivo de configuração do aplicativo e os `hibernate.cfg.xml` fornecer os arquivos mesma função: para configurar NHibernate. Qual arquivo você optar por usar depende da sua preferência sintaxe. `hibernate.cfg.xml` é o nome do arquivo escolhido pela convenção. Na verdade, você pode usar qualquer nome de arquivo (como `NHibernate.config`, Como `.Config` arquivos são automaticamente protegidos pelo ASP.NET quando implantado) e fornecer esse nome de arquivo para o método `Configure()`. É até possível misturar as duas opções e têm diferentes configurações para o desenvolvimento e implantação.

Uma opção alternativa é raramente usado para permitir que o aplicativo para fornecer um ADO.NET `IDbConnection` quando se abre uma NHibernate `ISession` a partir do `SessionFactory` (Por exemplo, chamando `sessionFactory.openSession(myConnection)`). Usando esta opção significa que você não tem que especificar todas as propriedades do banco de dados de conexão (as outras propriedades ainda são necessários). Nós não recomendamos essa abordagem para novas aplicações que podem ser configurados para usar a conexão do ambiente de banco de dados infra-estrutura.

De todas as opções de configuração, as configurações de conexão do banco de dados são os mais importantes porque, sem eles, NHibernate não vai saber como fazer corretamente falar com o banco de dados.

2.3.2 Configuração do acesso de banco de dados ADO.NET

Na maioria das vezes, a aplicação é responsável por obter conexões ADO.NET. NHibernate é parte do aplicação, por isso é responsável por obter essas conexões. Você diz NHibernate como chegar (ou criar novos) Conexões ADO.NET.

Figura 2.2 mostra como .NET interagir com ADO.NET. Sem NHibernate, a aplicação código geralmente recebe uma conexão ADO.NET do pool de conexão (que é configurado de forma transparente) e usa-lo para executar instruções SQL.

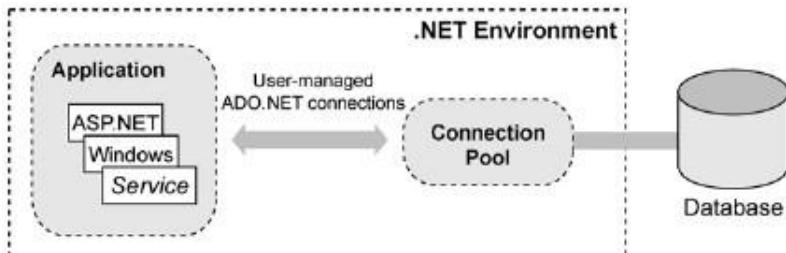


Figura 2.2 Acesso directo à conexões ADO.NET

Com NHibernate, o quadro muda: Ela age como um cliente do ADO.NET e seu pool de conexão, como mostrado na figura 2.3. O código de aplicativo usa o NHibernate `ISession` e `IQuery` APIs para operações de persistência e só tem de gerenciar as transações do banco de dados, de preferência usando o NHibernate `ITransaction` API.

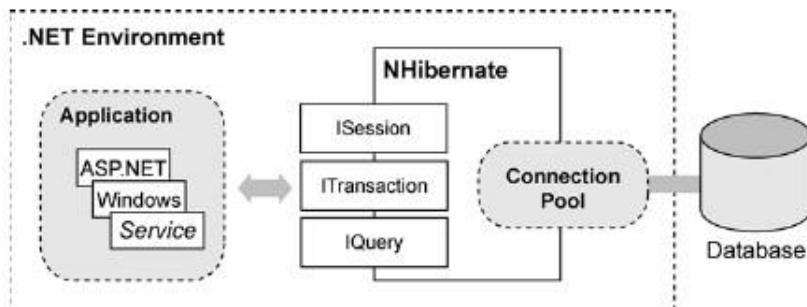


Figura 2.3 NHibernate gerenciar o acesso ao banco

Configuração do NHibernate usando hibernate.cfg.xml

Na listagem 2.1, usamos um arquivo chamado `hibernate.cfg.xml` para configurar NHibernate para acessar um Microsoft SQL Banco de dados do Server 2000.

Usando 2.1 listagem `hibernate.cfg.xml` para configurar Nhibernate

```
<? Xml version = "1.0"?>


```

Linhos Este código é especificar as seguintes informações:

O nome da classe. NET implementação do `IConnectionProvider` para NHibernate, aqui, estamos usando o padrão.

O nome da classe. NET implementação do `Dialecto` plataforma que permite que certos dependentes características. Apesar do esforço de padronização ANSI, SQL é implementado de forma diferente por vários bancos de dados fornecedores. Então, você deve especificar um `Dialecto`. NHibernate inclui suporte embutido para o SQL mais populares bancos de dados e novos dialetos pode ser definido facilmente.

Por favor, postar comentários ou correções para o fórum on-line em Autor <http://www.manning-sandbox.com/forum.jspa?forumID=295>

O nome da classe. NET implementação do ADO.NET **Motorista**. Note-se que, uma vez que NHibernate 1.2, quando se utiliza o nome parcial de um motorista que está no cache de assembly global (GAC), você tem que adicionar um `<qualifyAssembly>` elemento no arquivo de configuração do aplicativo para especificar o seu totalmente qualificado nome para que NHibernate pode carregá-lo com sucesso.

O `ConnectionString` é a corda usada para criar uma conexão com o banco, conforme definido pelo ADO.NET.

Note que estes nomes (exceto o `ConnectionString`) deve ser totalmente qualificado nomes de tipos, não são aqui, porque eles são implementados em `NHibernate.dll` biblioteca que é onde o quadro. NET procura não totalmente tipos qualificado quando NHibernate irá tentar carregá-los.

NHibernate partida

Como você começar a NHibernate com essas propriedades? Você declarou as propriedades em um arquivo chamado `hibernate.cfg.xml`, Então você precisa apenas colocar esse arquivo no diretório do aplicativo atual. Será automaticamente detectado e ler quando você cria um `Configuração` objeto e chamar seu `Configure()` método.

Vamos resumir as etapas de configuração que você aprendeu até agora (esta é uma boa hora para baixar e instalar NHibernate):

1. Se o provedor de dados ADO.NET de seu banco de dados ainda não está instalado, faça o download e instalá-lo, é geralmente disponíveis no site da base de dados fornecedores. Se você estiver usando SQL Server, então você pode pular este passo.
2. Adicionar `log4net.dll` como referência ao seu projeto. Isto é opcional, mas recomendado.
3. Decidir quais propriedades de acesso do banco de dados serão necessários NHibernate.
4. Deixe o `Configuração` saber sobre essas propriedades, colocando-os em um `hibernate.cfg.xml` arquivo no diretório atual.
5. Criar uma instância de `Configuração` em seu aplicativo, chamar o método `Configure()`, Carregar o classes mapeadas (com atributos .NET), utilizando `HbmSerializer.Default.Serialize()` e `AddInputStream()`, Carregar os arquivos de mapeamento XML usando `AddAssembly()`, `AddClass()` ou `AddXmlFile()`. Construir uma `ISessionFactory` exemplo, da `Configuração` chamando `BuildSessionFactory()`.
6. Lembre-se de fechar a instância do `ISessionFactory` (Utilizando `MySessionFactory.Close()`) quando você é feito usando NHibernate. Na maioria das vezes, você irá fazê-lo ao fechar o aplicativo.

Existem mais alguns passos quando utilizar serviços COM + Enterprise; vamos aprender mais sobre eles no capítulo 6. Não se preocupe, o código NHibernate pode ser facilmente integrado ao COM + como só poucos acréscimos são obrigatórios.

Agora você deve ter um sistema de NHibernate em execução. Criar e compilar uma classe persistente (o inicial `Empregado`, Por exemplo), adicione uma referência para `NHibernate.dll` e outras bibliotecas necessárias ao seu projeto, colocar um `hibernate.cfg.xml` arquivo no diretório do aplicativo atual, e construir uma `ISessionFactory` exemplo.

A próxima seção cobre as opções avançadas de configuração do NHibernate. Alguns deles são recomendados, como registro executado instruções SQL para depuração ou usando o arquivo de configuração XML conveniente em vez de propriedades simples. No entanto, você pode seguramente pular esta seção e voltar mais tarde depois de ter lido mais sobre classes persistentes no capítulo 3.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.4 Configurações avançadas de configuração

Quando você finalmente ter uma aplicação rodando NHibernate, vale bem a pena conhecer todos os NHibernate parâmetros de configuração. Esses parâmetros permitem otimizar o comportamento em tempo de execução do NHibernate, especialmente

por meio do ajuste da interação ADO.NET (por exemplo, usando ADO.NET atualizações em lote).

Não vou te aborrecer com esses detalhes agora, a melhor fonte de informações sobre as opções de configuração é a documentação de referência NHibernate. Na seção anterior, mostramos-lhe as opções que você precisa para começar iniciados.

No entanto, existe um parâmetro que preciso enfatizar, neste ponto. Você vai precisar dele sempre que continuamente você desenvolver software com NHibernate. A definição da propriedade `show_sql` ao valor `verdadeiro` permite o registro de todos os SQL gerados para o console. Você vai usá-lo para solução de problemas, ajuste de desempenho, e só para ver o que está

acontecendo. Vale a pena estar ciente de que sua camada ORM está fazendo, é por isso ORM não esconde de SQL desenvolvedores.

Até agora, temos assumido que você especificar parâmetros de configuração usando um `hibernate.cfg.xml` arquivo ou programaticamente usando a coleção `Configuration.Properties`. Você também pode especificar essas parâmetros usando o aplicativo arquivo de configuração (`web.config`, `app.config` etc).

2.4.1 Usando o arquivo de configuração do aplicativo

Você pode usar o arquivo de configuração do aplicativo (como demonstrado nas listagens 2.2 e 2.3) para configurar completamente um

`ISessionFactory` exemplo. Ela pode conter parâmetros de configuração usando um `<nhibernate>` seção, ou o mesmo conteúdo que `hibernate.cfg.xml` usando um `<hibernate-configuration>` seção. Muitos usuários preferem centralizar a configuração do NHibernate desta forma em vez de adicionar parâmetros para o `Configuração` no código do aplicativo.

Listagem 2.2 arquivo de configuração `app.config` usando `<nhibernate>`

```
<? Xml version = "1.0"?>
<configuration>
  <configSections>
    <Nome da seção = "nhibernate">
      type = System.Configuration.NameValueSectionHandler "
      Versão do sistema. = 1.0.5000.0, Culture = neutral,
      PublicKeyToken = b77a5c561934e089 "/>
    <Nome da seção = "log4net">
      type = "log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
  </ConfigSections>

  <nhibernate>
    <Adicionar
      key = "hibernate.connection.provider"
      value = "NHibernate.Connection.DriverConnectionProvider"
    />
    <Adicionar
      key = "hibernate.dialect"
      value = "NHibernate.Dialect.MsSql2000Dialect"
    />
    <Adicionar
      key = "hibernate.connection.driver_class"
      value = "NHibernate.Driver.SqlClientDriver"
    />
    <Adicionar
      key = "hibernate.connection.connection_string"
      value = "Catálogo inicial = nhibernate; Integrated Security = SSPI"
    />
  </ Nhibernate>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
<!-- Log4net definições de configuração aqui ... -->
</Configuration>
# 1 NHibernate declaração seção
# 2 Especificações de propriedade
Configurações Log4net # 3 deve estar lá
```

| 3

Seção NHibernate é declarado em # 1 como uma série de chave / valor entradas. Em # 2, a chave é o nome da propriedade para definir. Vamos aprender sobre log4net # 3 na próxima seção.

No entanto, é recomendado o uso de uma <Hibernate-Configuration section>:
Listagem 2.x arquivo de configuração app.config usando <hibernate-configuration>

```
<? Xml version = "1.0"?>
<configuration>
  <configSections>
    <Nome da seção = "hibernate-configuration" | 1
      type = "NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" /> | 1
        <Nome da seção = "log4net"
          type = "log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
    </ConfigSections>
```

```
xmlns="urn:nhibernate-configuration-2.2"> <hibernate-configuration
  <session-factory>
    <property name="connection.provider">
      NHibernate.Connection.DriverConnectionProvider
    </Property> | 2
    <property name="dialect">
      NHibernate.Dialect.MsSql2000Dialect
    </Property> | 2
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </Property> | 2
    <property name="connection.connection_string">
      Initial Catalog = nhibernate; Integrated Security = SSPI
    </Property> | 2
  </Session-factory>
</Hibernate-configuration>
```

| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2
| 2

```
<!-- Log4net definições de configuração aqui ... -->
</Configuration>
# 1 NHibernate declaração seção
# 2 Especificações de propriedade
Configurações Log4net # 3 deve estar lá
```

| 3

Agora, declara um # 1 <hibernate-configuration>, Como em [hibernate.cfg.xml](#), Que é baseado no esquema [nhibernate-configuration.xsd](#). Em # 2, o valor está dentro da tag <property>.

Desta forma é muito mais elegante e poderosa como você também pode especificar conjuntos / documentos de mapeamento e você pode configurar uma IDE como o Visual Studio para fornecer IntelliSense dentro do <Hibernate-configuration> seção: Basta copiar o arquivo de configuração do esquema ([nhibernate-configuration.xsd](#)) no sub-diretório \ Common7 \ Packages \ schemas \ xml \ diretório de instalação do Visual Studio. Você também pode copie o arquivo de esquema de mapeamento ([nhibernate-mapping.xsd](#)) Ter IntelliSense ao editar mapeamento arquivos. Você pode encontrar esses arquivos em código-fonte do NHibernate.

Note que você pode usar um <connectionStrings> elemento do arquivo de configuração para definir uma cadeia de ligação and then give its name to NHibernate using the property [hibernate.connection.connection_string_name](#).

Agora você pode inicializar usando NHibernate:

```
ISessionFactory sessionFactory = new Configuration()
  . Configure ()
  . BuildSessionFactory ();
```

Aguarde - how fez NHibernate saber onde o arquivo de configuração foi localizado?

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Quando `Configure()` foi chamado, NHibernate primeiro procurava para essas informações contidas no pedido arquivo de configuração, em seguida, em um arquivo chamado `hibernate.cfg.xml` no diretório atual. Se você quiser usar um nome de arquivo diferente ou ter o olhar NHibernate em um subdiretório, você deve passar um caminho para o `Configure()` método:

```
ISessionFactory sessionFactory = new Configuration()
    .Configure("NHibernate.config")
    .BuildSessionFactory();
```

Usando um arquivo de configuração XML é certamente mais confortável do que usando a configuração programática. O fato de que você pode ter os arquivos de classe mapeamento exteriorizada a partir da fonte do aplicativo (mesmo se seria somente em uma classe auxiliar de inicialização) é um grande benefício dessa abordagem. Você pode, por exemplo, usar diferentes conjuntos de arquivos de mapeamento (e diferentes opções de configuração), dependendo do seu banco de dados e ambiente (Desenvolvimento ou produção), e trocá-las por meio de programação.

Se você tem tanto um arquivo de configuração do aplicativo e `hibernate.cfg.xml` no diretório atual, o configurações do arquivo de configuração do aplicativo será usado.

Note que o `ISessionFactory` pode ser dado um nome. Este nome é especificado como um atributo da seguinte forma:
`<session-factory name="MySessionFactory">`. NHibernate usa esse nome para identificar este exemplo
aftercreation. You can use the static method:

`NHibernate.Impl.SessionFactoryObjectFactory.GetNamedInstance()` para recuperá-lo. Este recurso pode ser útil quando compartilhando um SessionFactory entre os componentes de baixo acoplamento. No entanto, é

raramente usado, porque, na maioria das vezes, é melhor se esconder atrás do NHibernate camada de persistência.

Agora que temos uma aplicação NHibernate funcional, vamos começar a encontrar erros de execução. Para facilitar o processo de depuração, precisamos log operações NHibernate.

2.4.2 Logging

NHibernate (e muitos outros implementações ORM) adia a execução de instruções SQL. Um **INSERIR** declaração não é geralmente executado quando o aplicativo chama `ISession.Save()`; Um **ATUALIZAÇÃO** não é imediatamente emitido quando o aplicativo chama `Item.AddBid()`. Em vez disso, as instruções SQL são normalmente emitido no final de uma transação. Esse comportamento é chamado write-behind, como mencionamos anteriormente.

Esse fato é evidência de que o rastreamento e depuração de código ORM é, por vezes não trivial. Em teoria, é possível, a aplicação para o tratamento de NHibernate como uma caixa preta e ignorar esse comportamento. Certamente, a Aplicação NHibernate não pode detectar esse write-behind (pelo menos, não sem recorrer a direta ADO.NET chamadas).

No entanto, quando você se encontrar solução de um problema difícil, você precisa ser capaz de ver exatamente o que está acontecendo dentro NHibernate. Desde NHibernate é open source, você pode facilmente entrar no NHibernate código. Ocasionalmente, isso ajuda muito! No entanto, especialmente em face de write-behind comportamento, depuração NHibernate pode obter rapidamente você perdeu. Você pode usar o log para obter uma visão de internos do NHibernate.

Já mencionamos o `show_sql` parâmetro de configuração, que geralmente é o primeiro porto de escala solução de problemas. Às vezes, o SQL por si só é insuficiente e, nesse caso, você deve cavar um pouco mais.

NHibernate registra todos os eventos interessantes usando o log4net biblioteca de código aberto. Para ver qualquer saída do log4net, você precisa adicionar algumas informações no arquivo de configuração do aplicativo. O exemplo da listagem 2,4 direciona todas as mensagens de log para o console:

Listagem de configuração básica de 2,4 log4net

```
<? Xml version = "1.0"?>
<configuration>
  <configSections>
    <Seção <
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        name = "log4net"
        type = "log4net.Config.Log4NetConfigurationSectionHandler, log4net"
    />
</ConfigSections>

<log4net>
<Nome appender = "ConsoleAppender"
      type = "log4net.Appender.ConsoleAppender, log4net">
    <layout log4net> type="log4net.Layout.PatternLayout,
      <param name="ConversionPattern" value="%m" />
    </ Layout>
  <Appender />
  <raiz>
    <priority value="WARN" />
    <appender-ref ref="ConsoleAppender" />
  </ Root>
</ Log4net>
</ Configuration>

```

Você pode facilmente mesclar esse arquivo na listagem 2.3. Com essa configuração, você não verá as mensagens de log em muitos tempo de execução. Substituindo o valor de prioridade de **WARN** para **INFO** ou **DEBUG** irá revelar o funcionamento interno do NHibernate. Verifique se você não fizer isso em um ambiente de produção escrita o log será muito mais lento que o acesso ao banco real. Nós não vamos dar mais detalhes sobre a configuração log4net. Sinta-se livre para ler seu documentação.

Nesta seção, falamos sobre a configuração de acesso de banco de dados. Esta configuração é inútil se NHibernate não sabe como manipular as nossas entidades. A próxima seção cobre mapeamento NHibernate.

2.5 Síntese

Neste capítulo, demos uma olhada de alto nível NHibernate e sua arquitetura depois de executar um simples "Olá Exemplo do mundo ". Você também viu como configurar NHibernate em vários ambientes e com vários técnicas.

As interfaces de configuração e SessionFactory são os pontos de entrada para NHibernate para aplicações execução em ambos os ambientes WinForms e ASP.NET.

Hibernate pode ser integrado em quase todos os ambientes .NET, seja ele uma aplicação Console, um ASP.NET aplicação, ou uma totalmente gerenciado três níveis de aplicação cliente / servidor. Os elementos mais importantes de um Configuração NHibernate são os recursos de banco de dados (configuração de conexão), as estratégias de operação e, é claro, os metadados de mapeamento baseados em XML.

Interfaces de configuração do NHibernate foram concebidos para cobrir como cenários de uso possíveis enquanto ainda está sendo fácil de entender. Normalmente, algumas modificações em seu arquivo de configuração. E uma linha de código são suficiente para obter NHibernate instalado e funcionando.

Nada disso é muito uso sem algum classes persistentes e seus documentos de mapeamento XML. Os próximos capítulo é dedicado à escrita e mapeamento de classes persistentes. Em breve você vai ser capaz de armazenar e recuperar objetos persistentes em um aplicativo real com um objeto não trivial / mapeamento relacional.

3

Escrito e Mapeamento aulas

Este capítulo aborda

- POCO básico para modelos de domínio rico
- O conceito de identidade de objeto e seu mapeamento
- Mapeamento de herança de classe
- Associações e mapeamentos de coleções

O "Olá Mundo" exemplo no capítulo 2 apresentamos o NHibernate, no entanto, não é muito útil para compreensão dos requisitos de aplicações do mundo real com modelos de dados complexos. Para o resto do livro, nós vamos usar um exemplo muito mais sofisticada aplicação de um leilão on-line do sistema de demonstrar NHibernate.

Neste capítulo, começamos nossa discussão sobre a aplicação, introduzindo um modelo de programação para classes persistentes. Projetar e implementar as classes persistentes é um processo multi-passo que examinaremos em detalhe.

Este capítulo é bastante desde que nós cobrimos muitos tópicos interligados. Primeiro, você vai aprender a identificar os objetos de negócios (Ou entidades) de um domínio do problema. Criamos um modelo conceitual dessas entidades e seus atributos, chamado de modelo de domínio. Nós implementamos este modelo de domínio em C #, criando uma classe persistente para cada entidade, e nós vamos passar algum tempo a explorar exatamente o que estas classes. NET deve ser parecida.

Nós, então, definir metadados de mapeamento para dizer NHibernate como essas classes e suas propriedades se relacionam com tabelas e colunas. Nós já cobriu a base desta etapa no capítulo 2. Neste capítulo, damos uma em profundidade apresentação das técnicas de mapeamento de grão fino classes, a identidade do objeto, herança e associações. Este capítulo, portanto, fornece o início de uma solução para os problemas primeiro genérico de ORM enumerados no capítulo 1, secção 1.3.1 Por exemplo, como é que vamos traçar nosso refinado objetos para nossas mesas simples, ou como é que vamos mapear hierarquias de herança para nossas mesas?

Vamos começar com a introdução do aplicativo de exemplo.

3.1 A aplicação CaveatEmptor

O CaveatEmptor aplicativo de leilão online demonstra técnicas ORM e funcionalidade NHibernate; você pode baixar o código fonte para o aplicativo inteiro trabalhando a partir do site web <http://caveatemptor.hibernate.org/>. A aplicação terá uma interface de usuário baseado em console. Nós não vamos pagar muita atenção à interface do usuário; vamos nos concentrar sobre o código de acesso a dados. No capítulo 8, discutimos a mudanças que seriam necessárias se fôssemos para executar toda a lógica de negócios e acesso a dados de um separado camada de negócios. E no capítulo 10, vamos discutir muitas soluções para problemas comuns que surgem quando NHibernate integração em aplicações Windows e Web.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Mas, vamos começar pelo começo. A fim de compreender as questões envolvidas no projeto ORM, vamos fingir a aplicação CaveatEmptor ainda não existe, e que estamos construindo a partir do zero. Nossa primeira tarefa seria análise.

3.1.1 Analisando o domínio de negócio

Um esforço de desenvolvimento de software começa com a análise do domínio do problema (assumindo que o código legado ou não banco de dados legado já existe).

Nesta fase, você, com a ajuda de especialistas domínio do problema, identificar os principais entidades que são relevantes para o sistema de software. Entidades são geralmente noções compreendida pelos usuários do sistema: **Pagamento, Cliente, Ordem, Item, Oferta** E assim por diante. Algumas entidades podem ser abstrações de coisas menos concreto que o usuário pensa sobre (por exemplo, **PricingAlgorithm**). Mas mesmo estes, normalmente, ser compreensível para o usuário. Todos essas entidades são encontrados na visão conceitual do negócio, que por vezes chamamos de uma modelo de negócio. Desenvolvedores de software orientado a objetos analisar o modelo de negócios e criar um modelo de objeto, ainda no nível conceitual (sem código C #). Esse modelo de objeto pode ser tão simples como uma imagem mental existente apenas no mente do desenvolvedor, ou pode ser tão elaborado como um diagrama de classes UML (como na figura 3.1) criado por um CASE (Computer-Aided Software Engineering) ferramenta como o Microsoft Visio, Enterprise Architect Sparx Systems ou UMLet.

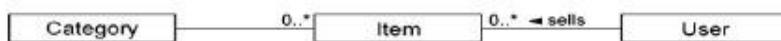


Figura 3.1 Um diagrama de classes de um modelo de objeto de leilão online típico

Este modelo simples contém entidades que você é obrigado a encontrar em qualquer sistema de leilão típico: **Categoria, Item** E **Usuário**. As entidades e seus relacionamentos (e talvez seus atributos) são representadas por este modelo do domínio do problema. Chamamos esse tipo de modelo de um modelo orientado a objetos de entidades de domínio do problema, abrangendo apenas as entidades que são de interesse para o usuário-a modelo de domínio. É uma visão abstrata do mundo real. Nós nos referimos a este modelo quando implementamos o nosso persistente. NET.

Vamos examinar o resultado da nossa análise do domínio do problema da aplicação CaveatEmptor.

3.1.2 O modelo de domínio CaveatEmptor

Os leilões do site CaveatEmptor muitos tipos diferentes de itens, desde equipamentos eletrônicos para passagens aéreas. Leilões de proceder de acordo com o "Inglês leilão" modelo: Os usuários continuam a dar lances em um item até o período de oferta para o item expira, eo maior lance ganha.

Em qualquer loja, os produtos são classificados por tipo e agrupados com produtos semelhantes em seções e nas prateleiras.

Claramente, o nosso catálogo do leilão requer algum tipo de hierarquia de categorias item. Um comprador pode navegar entre eles

categorias ou arbitrariamente busca por categoria e os atributos de item. Listas de itens aparecem no browser categoria e busca as telas de resultados. Selecionar um item de uma lista levará o comprador a uma visão de detalhes do item.

Um leilão é composto por uma seqüência de lances. Uma oferta especial é o lance vencedor. Detalhes do usuário incluir o nome,

endereço de login, endereço de e-mail, e informações de faturamento.

A teia de confiança é uma característica essencial de um site de leilão online. A web permite que os usuários de confiança para construir uma reputação de confiabilidade (ou falta de confiança). Os compradores podem criar comentários sobre os vendedores (e vice-versa), e os comentários são visíveis para todos os outros usuários.

A visão geral de alto nível do nosso modelo de domínio é mostrado na figura 3.2. Vamos discutir brevemente alguns interessantes características deste modelo.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

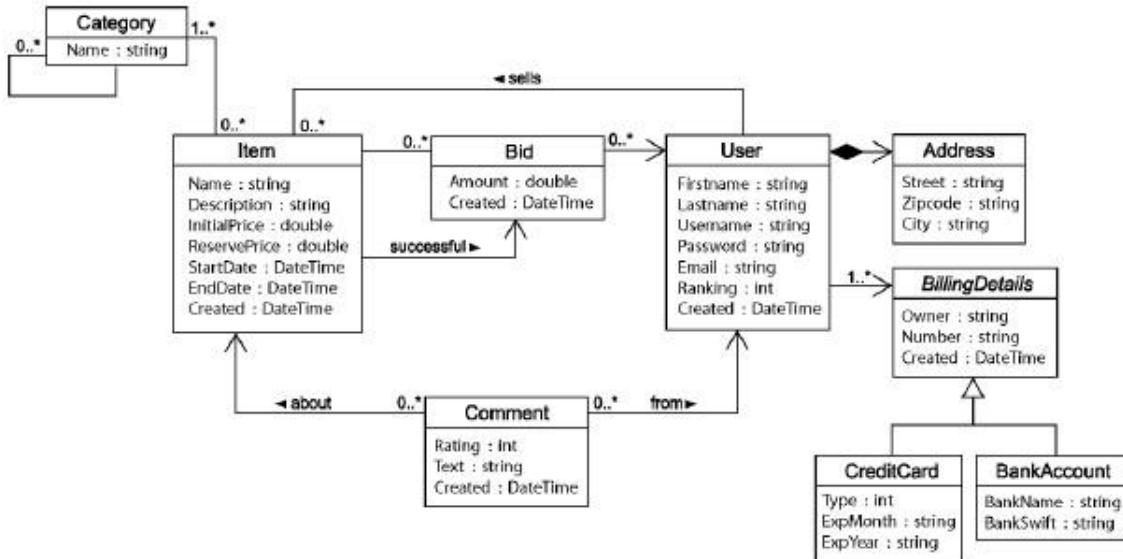


Figura 3.2 classes persistentes do modelo de objeto CaveatEmptor e suas relações

Cada item pode ser leiloado apenas uma vez, então não precisamos fazer **Item** distinta da **Leilão** entidades. Em vez disso, temos uma entidade único item chamado leilão **Item**. Assim, **Oferta** está associado diretamente com

Item. Os usuários podem escrever **Comentários** sobre outros usuários apenas no contexto de um leilão, daí a associação entre **Item** e **Comentário**. O **Endereço** informações de um **Usuário** é modelado como uma classe separada, mesmo embora o **Usuário** pode ter apenas um **Endereço**. Nós permitimos que o usuário tenha várias **BillingDetails**.

As estratégias de faturamento vários são representados como subclasses de uma classe abstrata (permitindo futura extensão).

A **Categoria** pode ser aninhado dentro de outro **Categoria**. Isto é expresso por um recursiva associação, a partir do **Categoria** a própria entidade. Note-se que um único **Categoria** podem ter múltiplas categorias infantil, mas ao mais uma categoria pai. Cada **Item** pertence a pelo menos um **Categoria**.

As entidades em um modelo de domínio deve encapsulam estado e comportamento. Por exemplo, o **Usuário** entidade deve definir o nome e endereço de um cliente ea lógica necessária para calcular os custos de transporte para itens (a este cliente particular). Nosso modelo de domínio é um rico modelo de objeto, com as associações complexas, interações e relações de herança. Uma discussão interessante e detalhado de técnicas orientadas a objetos para trabalhar com modelos de domínio pode ser encontrada em Padrões de Enterprise Architecture Aplicação [Fowler 2003] ou em Domain-Driven Design [Evans 2004].

No entanto, neste livro, não teremos muito a dizer sobre regras de negócio ou sobre o comportamento dos nossos modelo de domínio. Esta não é certamente porque consideramos esta uma preocupação sem importância, mas sim, essa preocupação é principalmente ortogonal ao problema da persistência. É o estado das entidades que é persistente. Então, nós concentrar nossa discussão sobre a melhor forma representam o estado no nosso modelo de domínio, não em como representar comportamento. Por exemplo, neste livro, não estamos interessados em como imposto para itens vendidos ou é calculado como o sistema pode aprovar uma nova conta de usuário. Estamos mais interessados em como a relação entre usuários e os itens que vendem é representada e fez persistente.

Pode-se usar ORM sem um modelo de domínio?

Ressaltamos que a persistência objeto com ORM completo é mais adequado para aplicações com base em um rico modelo de domínio. Se sua aplicação não implementa regras de negócios complexas ou interações complexas entre as entidades (ou se você tiver poucas entidades), você pode não precisar de um modelo de domínio. Muitos simples e alguns problemas não tão simples são perfeitamente adequados para a mesa de soluções orientadas, onde o pedido é concebido em torno do modelo de dados do banco de dados, em vez de em torno de um modelo de domínio orientada a objetos, muitas vezes

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

com a lógica executada no banco de dados (stored procedures). No entanto, o mais complexo e expressivo seu modelo de domínio, mais você vai se beneficiar do uso NHibernate, que brilha quando se lida com o complexidade do objeto / relacional persistência.

Agora que temos um modelo de domínio, o nosso próximo passo é implementar em C#. Vejamos algumas das coisas que precisa considerar.

3.2 Implementação do modelo de domínio

Diversas questões devem ser tratadas normalmente quando você implementar um modelo de domínio. Por exemplo, como você separar os interesses de negócio das preocupações transversais (como transações e até mesmo persistência)? Que tipo de persistência é necessária: Você precisa automatizado ou transparente persistência? Você tem que usar um modelo de programação específica para conseguir isso? Nesta seção, vamos examinar esses tipos de problemas e como abordá-los em uma aplicação NHibernate típico.

Vamos começar com uma questão que qualquer implementação deve lidar com: a separação de preocupações. O domínio implementação do modelo é geralmente um componente, organizador central, é reutilizado fortemente sempre que você implementar funcionalidade nova aplicação. Por esta razão, você deve estar preparado para ir para alguns comprimentos para garantir que outras preocupações que aspectos do negócio não vazam para a implementação do modelo de domínio.

3.2.1 Endereçamento vazamento das

preocupações

A implementação do modelo de domínio é uma parte tão importante da código que ele não deve depender de outros. NET APIs. Por exemplo, o código no modelo de domínio não deve executar Input / Output operações ou ligue para o banco de dados através da API ADO.NET. Isso permite que você reutilizar a implementação do modelo de domínio em praticamente qualquer lugar. A maioria importante, facilita a teste de unidade o modelo de domínio (no NUnit, por exemplo) fora de qualquer aplicativo servidor ou ambiente gerenciado outros.

Dizemos que o modelo de domínio deve ser "preocupado" apenas com a modelagem do domínio do negócio. No entanto, existem outras preocupações, como a persistência, gerenciamento de transações, e autorização. Você não deve colocar código que trata dessas preocupações transversais nas classes que implementam o modelo de domínio. Quando estes preocupações começam a aparecer nas aulas de modelo de domínio, nós chamamos isso de um exemplo de vazamento de preocupações.

DataSet não resolver este problema. Ele não pode ser considerado como um modelo de domínio, principalmente porque não é projetado para incluir regras de negócio.

Muita discussão tem ido para o tema da persistência, e ambos NHibernate e DataSets cuidar de essa preocupação. No entanto, NHibernate oferece algo que não DataSets: persistência transparente.

3.2.2 persistência transparente e automatizada

Um DataSet permite que você extraia as alterações executadas nele, a fim de persistir eles. NHibernate fornece uma característica diferente que é muito sofisticada e poderosa: Pode automaticamente persistem as alterações de forma que é transparente para o seu modelo de domínio.

Usamos transparente para significar uma separação completa das preocupações entre as classes persistentes da modelo de domínio e a lógica de persistência em si, onde as classes persistentes são inconscientes e não têm dependência para-o mecanismo de persistência.

Nosso Item classe, por exemplo, não terá qualquer dependência nível de código a qualquer API NHibernate. Além disso:

NHibernate não exige que qualquer base de classes especiais ou interfaces de ser herdado ou implementadas por classes persistentes. Nem são as classes especiais utilizados para implementar propriedades ou associações.

Assim,

persistência transparente melhora a legibilidade do código, como você verá em breve.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Classes persistentes podem ser reutilizados fora do contexto de persistência, em testes de unidade ou na interface do usuário

(UI) camadas, por exemplo. Testabilidade é um requisito básico para aplicações com modelos de domínio rico. Em um sistema com persistência transparente, os objetos não estão cientes de armazenamento de dados subjacente, eles precisam

nem mesmo estar cientes de que estão sendo mantido ou recuperado. Preocupações persistência são externalizados para um

genérico persistência gerente interface, no caso do NHibernate, o `ISession` e `IQuery` interfaces.

Persistência transparente promove um grau de portabilidade, sem interfaces especiais, as classes persistentes são dissociada de qualquer solução de persistência particular. Nossa lógica de negócio é totalmente reutilizável em qualquer outra aplicação

contexto. Poderíamos facilmente mudar para outro mecanismo de persistência transparente.

Por esta definição de persistência transparente, você vê que certas camadas não automatizada de persistência são transparentes (por exemplo, o padrão DAO), porque eles desacoplar o código de persistência relacionados com resumo programação de interfaces. Apenas simples. NET sem dependências estão expostas a lógica do negócio. Por outro lado, algumas camadas de persistência automatizada (como muitas soluções ORM) não são transparentes, porque eles requerem interfaces especiais ou modelos de programação intrusiva.

Nós consideramos a transparência, conforme necessário. Na verdade, persistência transparente deve ser um dos principais objetivos do qualquer solução ORM. No entanto, nenhuma solução de persistência automatizada é completamente transparente: Toda automatizada camada de persistência, incluindo NHibernate, impõe alguns exigências sobre as classes persistentes. Por exemplo, NHibernate requer que a coleta de valores de propriedades ser digitado em uma interface, como `IList` ou `IDictionary` (Ou a sua. NET 2.0 versões genéricas) e não para uma implementação real, tais como `ArrayList` (Esta é uma boa prática de qualquer maneira). (Discutimos as razões para este requisito no Apêndice B, "ORM estratégias de implementação. ")

Agora você sabe por que o mecanismo de persistência deve ter impacto mínimo sobre a forma como você implementar um modelo de domínio e que a persistência transparente e automatizada são obrigatórios. DataSet não é adequado aqui, então o que tipo de modelo de programação você deve usar? Você precisa de um modelo de programação especial em tudo? Em teoria, não; na prática, você deve adotar uma disciplina, modelo de programação consistente, que é bem aceito pela NET. da comunidade. Vamos discutir este modelo de programação e ver como ele trabalha com NHibernate.

3.2.3 Escrevendo POCOs

Desenvolvedores descobriram DataSets a ser natural para a representação de objetos de negócios em muitas situações. O

oposto de um modelo pesado como é o DataSet Plain Old CLR Objeto (POCO). É uma abordagem back-to-basics que consiste essencialmente de usar classes unbound na camada de negócios.¹

Ao usar NHibernate, as entidades são implementados como POCOs. Os requisitos poucos que NHibernate impõe suas entidades são também as melhores práticas para o modelo de programação POCO. Assim, a maioria são POCOs

NHibernate-compatível, sem quaisquer alterações. O modelo de programação vamos introduzir é uma mistura não-intrusiva

das melhores práticas de POCO e requisitos NHibernate. A POCO declara métodos de negócio, que definem comportamento, e propriedades, que representam o estado. Algumas propriedades representam associações POCOs outros.

Listagem 3.1 mostra uma classe POCO simples, é uma implementação do `Usuário` entidade do nosso modelo de domínio.¹

Listagem 3.1 implementação POCO do `Usuário` classe

¹O POCO termo foi derivado do termo POJO Java, o que significa Plain Old Java Object. Às vezes é também escrito como Plain Ordinário Java Objects, este termo foi cunhado em 2002 por Martin Fowler, Rebecca Parsons, e Josh Mackenzie. Como uma alternativa para POCO, também é comum usar o termo PONO que significa Plain Old .NET Object.

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        string username privado;
        Endereço endereço privado;
        Usuário pública () {
            Nome de usuário {public string
                get { nome return; }
                set {username = valor; }
            }
            Endereço de público {
                get {endereço de retorno; }
                set {endereço = valor; }
            }
        }
        pública CalcShipCosts MonetaryAmount (Endereço
            / ...
        )
    }

} # 1 Serializable classe
# 2 construtor da classe
# 3 Propriedades
# Método Business 4

```

Alguns comentários sobre o código acima:

NHibernate não requer classes persistentes ser serializável (como esta classe é, mostrado na # 1). No entanto, é comumente necessários, principalmente quando se usa. NET Remoting.

NHibernate requer um "default" construtor sem parâmetros para cada classe persistente (mostrado na # 2). O construtor pode ser não-públicos, mas deve ser pelo menos protegidos se runtime gerado proxies serão utilizados para de otimização de desempenho (ver capítulo 4). Note-se que. NET adiciona automaticamente um público sem parâmetros construtor para as classes se você não tiver escrito um no código.

As propriedades do POCO implementar os atributos de nossas entidades de negócio, como mostrado na # 3. Por exemplo, o

UsuárioUserName 's nome fornece acesso à chave privada **userName** variável de instância (o mesmo para **Endereço**).

NHibernate não exige que as propriedades ser declarado público, que pode facilmente usar as privadas também. Alguns propriedades fazer algo mais sofisticado do que simples acessar variáveis de instância (de validação, por exemplo). Trivial propriedades são comuns, no entanto.

No item 4, esta POCO também define um método de negócios que calcula o custo de envio de um item para um determinado

usuário (que deixou de fora a implementação deste método).

Agora que você entende o valor de usar POCO classes persistentes como o modelo de programação, vamos ver como você lida com as associações entre as classes.

3.2.4 Implementando associações POCO

Você usa propriedades para expressar as associações entre as classes POCO, e você usar métodos de acesso para navegar o objeto gráfico em tempo de execução. Vamos considerar as associações definidas pela **Categoria** classe. O primeiro associação é mostrado na figura 3.3.

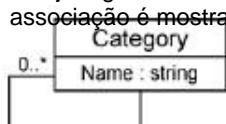


Figura 3.3 Diagrama do **Categoria** classe com uma associação

Tal como acontece com todos os nossos diagramas, que deixou de fora a associação atributos relacionados (**parentCategory** e **childCategories**) Porque seria a desordem ilustração. Esses atributos e os métodos que manipular seus valores são chamados código de andaimes.

Vamos implementar o código para o scaffolding um-para-muitos auto-associação de **Categoria**:

```

public class Categoria: ISerializable {
    private string name;

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

parentCategory Categoria privado;
privada ISET childCategories HashSet = new ();
Categoria pública () {}
...
}

```

Note que podemos usar .NET 2.0 genéricos aqui escrevendo `ChildCategories ISET <category>`. E não outra mudança seria necessária (mesmo no mapeamento).

Para permitir a navegação bidirecional da associação, que exigem dois atributos. O `parentCategory` atributo implementa o de valor único final da associação e é declarada como sendo do tipo `Categoria`. O muitos de valor final, implementado pelo `childCategories` atributo, deve ser do tipo coleção. Nós escolhemos um `ISET`, E inicializar a variável de instância para uma nova instância de `HashSet`.

NHibernate necessita de interfaces para coleta de atributos digitado. Você deve, por exemplo, use `ISET` mais que `HashSet`. Em tempo de execução, NHibernate envolve a instância de coleção com uma instância de um dos Próprias classes do NHibernate. (Esta classe especial não é visível para o código da aplicação). É uma boa prática programa para interface de coleção, ao invés de implementações concretas, pelo que esta restrição não deve incomodá-lo.

Utilizado biblioteca externa: lesi.Collections

Em Java, existe uma espécie de coleção chamada `Conjunto` que permite armazenar itens sem duplicação (isto é, você não pode adicionar o mesmo objeto muitas vezes). Mas .NET não fornece um equivalente para esta coleção. Assim, NHibernate utiliza uma biblioteca chamada `lesi.Collections` que inclui a interface `ISET` e muitos implementações (como `HashSet`). Seu comportamento é semelhante ao do `IList` então você deve ser capaz de facilmente usá-los. Vamos freqüentemente usam `Conjuntos` porque a sua semântica se encaixa com a exigência de nossas aulas.

Agora temos algumas variáveis de instância privadas, mas sem interface pública para permitir o acesso a partir do código de negócios ou administração de imóveis por NHibernate. Vamos adicionar algumas propriedades para o `Categoria` classe:

```

Nome {public string
    get {return nome;}
    set {nome = valor;}
}
pública ChildCategories ISET {
    get {childCategories return;}
    set {childCategories = valor;}
}
Categoria público ParentCategory {
    get {return parentCategory;}
    set {parentCategory = valor;}
}

```

Novamente, essas propriedades devem ser declaradas `público` somente se eles são parte da interface externa do classe persistente, a interface pública utilizado pela lógica da aplicação.

O procedimento básico para a adição de uma criança `Categoria` a um pai `Categoria` olha como este:

```

Categoria Categoria aParent = new ();
Categoria Categoria aChild = new ();
aChild.ParentCategory = aParent;
aParent.ChildCategories.Add (aChild);

```

Sempre que uma associação é criada entre um pai `Categoria` e uma criança `Categoria`, Duas ações são necessário:

O `parentCategory` da criança deve ser definida, efetivamente quebrar a associação entre a criança e seu pai antigo (não pode ser apenas um dos pais para qualquer criança).

A criança deve ser adicionado ao `childCategories` coleção do novo pai `Categoria`.

Relacionamentos gerenciados no NHibernate

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

NHibernate não "gerencia" as associações persistentes. Se você deseja manipular uma associação, você deve escrever exatamente o mesmo código que você escrever sem NHibernate. Se uma associação é bidirecional, ambos os lados da relação deve ser considerada. De qualquer forma, isso é necessário se você quiser usar seus objetos sem NHibernate (para testes ou com a interface do usuário).

Se você tiver problemas em entender o comportamento de associações em NHibernate, pergunte a si mesmo: "O que eu faria sem NHibernate?" NHibernate não muda o usual. Semântica.NET.

É uma boa idéia para adicionar um método de conveniência para o **Categoria** classe que grupos essas operações, permitindo a reutilização e ajudando a garantir correção:

```
public void AddChildCategory(Categoria childCategory) {  
    if (childCategory.ParentCategory != null)  
        childCategory.ParentCategory.ChildCategories  
            .Remove(childCategory);  
    childCategory.ParentCategory = this;  
    childCategories.Add(childCategory);  
}
```

O **AddChildCategory ()** método não só reduz as linhas de código quando se tratar de **Categoria** objetos, mas também reforça a cardinalidade da associação. Erros que surgem de deixar de fora um dos dois ações necessárias sejam evitados. Este tipo de agrupamento de operações deve sempre ser fornecida para as associações, se possível.

Porque gostaríamos que o **AddChildCategory ()** ser o único método modificador externamente visível para as categorias infantil, fazemos a **ChildCategories** propriedade privada; podemos acrescentar mais métodos para o acesso para ChildCategories se necessário. NHibernate não se importa se são propriedades privadas ou públicas, de modo que possa se concentrar em projeto API bom.

Um tipo diferente de relação existe entre **Categoria** eo **Item**: A bidirecional many-to-many associação (Ver figura 3.4).

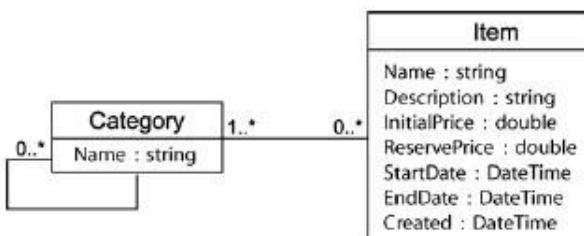


Figura 3.4 **Categoria** e os associados **Item**

No caso de uma associação muitos-para-muitos, ambos os lados são implementados com a coleta de atributos de valores.

Vamos adicionar os novos atributos e métodos para acessar o **Item** classe para o nosso **Categoria** classe, como mostrado na listagem 3.2.

Listagem 3.2 **Categoria** para **Item** código de andaimes

```
...  
itens ISET private = HashSet new();  
...  
Itens pública ISET {  
    get {itens return;}  
    set {itens = valor;}  
}
```

O código para o **Item** classe (a outra extremidade da associação muitos-para-muitos) é semelhante ao código para a **Categoria** classe. Nós adicionamos o **coleção** atributo, as propriedades padrão, e um método que simplifica gestão de relacionamento (você também pode adicioná-lo ao **Categoria** classe, veja a lista 3.3).

Listagem 3.3 **Item** para **Categoria** código de andaimes

```
class Item {pública  
    private string name;  
    descrição private string;
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    ...
    categorias privada ISET HashSet = new ();
    ...
    Categorias pública ISET () {
        get {categorias return;}
        set {categorias = valor;}
    }
    public void AddCategory (Categoria categoria) {
        category.Items.Add (this);
        categories.Add (categoria);
    }
}

```

O `AddCategory ()` do `Item` método é semelhante ao `AddChildCategory` método de conveniência do `Categoria` classe. É usado por um cliente para manipular a relação entre `Item` e um `Categoria`. Para o causa de legibilidade, não vamos mostrar métodos de conveniência em amostras futuro código e assumir que você vai adicionará-los de acordo com seu próprio gosto.

Agora você deve entender como criar classes para formar o seu modelo de domínio que pode ser mantido por NHibernate. Além disso, você deve ser capaz de criar associações entre essas classes, usando métodos de conveniência sempre que necessário para melhorar o modelo de domínio. O próximo passo é para enriquecer ainda mais o modelo de domínio, adicionando lógica de negócios. Vamos começar por olhar para como você pode, você pode adicionar lógica para suas propriedades.

3.2.5 Adicionando lógica para propriedades

Uma das razões que nós gostamos de usar as propriedades é que eles fornecem encapsulamento: A interna escondida implementação de uma propriedade pode ser alterada sem qualquer alteração à interface pública. Isso permite que você abstrair a estrutura de dados interna de uma instância de classe as variáveis: desde o design do banco de dados.

Por exemplo, se seu banco de dados armazena um nome do usuário como um único `NOME` coluna, mas o seu `Usuário` classe tem `firstname` e `lastname` propriedades, você pode adicionar o seguinte persistente `nome` propriedade para sua classe:

```

Usuário public class {
    firstname private string;
    lastname private string;
    ...
    Nome {public string
        get {return firstname + " " + sobrenome;}
        conjunto {
            string [] nomes = value.Split (" ");
            nome = nomes [0];
            sobrenome = nomes [1];
        }
    }
    ...
}

```

Mais tarde, você verá que um NHibernate tipo personalizado é provavelmente a melhor maneira de lidar com muitos desses tipos de situações. No entanto, isso ajuda a ter várias opções.

Propriedades também podem executar a validação. Por exemplo, no exemplo a seguir, o `FirstName` propriedade setter verifica se o nome é maiúscula:

```

Usuário public class {
    firstname private string;
    ...
    Apelido {public string
        get {return firstname;}
        conjunto {

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        if (!StringUtil.IsCapitalizedName (nome))
            throw new InvalidNameException (valor);
        nome = valor;
    }
    ...
}

```

No entanto, NHibernate irá usar mais tarde nossas propriedades para preencher o estado de um objeto ao carregar o objeto. Às vezes a gente preferiria que esta validação não ocorrer quando NHibernate é inicializar um recém-carregado objeto. Nesse caso, pode fazer sentido para dizer NHibernate para acessar diretamente a instância variáveis (que mais tarde irá ver que podemos fazê-lo pelo mapeamento da propriedade com `access = "field"` em NHibernate metadados), forçando NHibernate para ignorar a propriedade eo acesso a variável de instância diretamente. Outra questão a considerar é checagem suja. NHibernate detecta automaticamente as mudanças de estado de objeto, a fim de sincronizar o atualizada estado com o banco de dados. Geralmente é completamente seguro para retornar um objeto diferente do acessador get para o objeto passado por NHibernate para o acessador set. NHibernate irá comparar os objetos por valor não por identidade de objeto determinar se estado persistente da propriedade precisa ser atualizado. Por exemplo, o seguinte acessador get não vai resultar em atualizações desnecessárias SQL:

```

Apelido {public string
    get {return new String (nome);}
}

```

No entanto, há uma exceção muito importante. Coleções são comparadas por identidade!

Para uma propriedade mapeada como uma coleção persistente, você deve retornar exatamente a instância mesma coleção do assessor de obter o NHibernate passado para o acessador set. Se não o fizer, irá atualizar o NHibernate banco de dados, mesmo que nenhuma atualização é necessária, cada vez a sessão sincroniza estado guardadas na memória com o banco de dados. Este tipo de código deve quase sempre ser evitado em propriedades:

```

Nomes { IList pública
    get {return new ArrayList (nomes);}
    set {nomes = new string [value.Count]; value.CopyTo (nomes, 0);}
}

```

Você pode ver que NHibernate não restringir desnecessariamente o modelo de programação POCO. Você é livre para implementar qualquer lógica que você precisa em propriedades (desde que você mantenha a instância de coleção mesmo em ambos os se e definir os assessores). Note-se que as coleções não deve ter um setter em tudo.

Se for absolutamente necessário, você pode dizer NHibernate para usar uma estratégia de acesso diferentes para ler e definir o estado de uma propriedade (por exemplo, o acesso de campo direto exemplo), como você verá mais tarde. Esse tipo de transparência garante uma implementação do modelo independente e reutilizável domínio.

Neste momento temos um número definido um número de classes para o nosso modelo de domínio e criou algumas associações entre eles. Nós também adicionamos acrescentou alguns métodos de conveniência para tornar o trabalho mais fácil com o modelo, e acrescentou alguma lógica de negócio. Nosso próximo objetivo é ser capaz de carregar e salvar objetos no modelo de domínio para e a partir de um banco de dados relacional. Então, precisamos agora de olhar para a criação das peças necessárias para **3.3 Definindo metadados de mapeamento** nossos objetos, ou mais especificamente, o mapeamento objeto-relacional.

Ferramentas ORM requer um formato de metadados para a aplicação para especificar o mapeamento entre classes e tabelas, propriedades e colunas, associações e chaves estrangeiras,. NET e tipos SQL. Esta informação é chamada o mapeamento objeto / relacional metadados. Ele define a transformação entre os sistemas de tipo diferente de dados e representações relacionamento.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

É nosso trabalho como desenvolvedores para definir e manter esses metadados. Há duas maneiras diferentes de fazer isso:

atributos e metadados. Nesta seção, você vai aprender a escrever usando o mapeamento dessas duas maneiras e vamos compará-los para que você possa decidir qual deles usar. Vamos começar com o mapeamento você já está familiarizado com: O mapeamento usando arquivos XML.NET..

3.3.1 Mapeamento usando XML

NHibernate fornece um formato de mapeamento baseado no XML popular. Documentos de mapeamento por escrito e com

XML são leves, são legíveis, são facilmente mão-editável, são facilmente manipulados por controle de versão sistemas e editores de texto, e pode ser personalizado em tempo de implantação (ou até mesmo em tempo de execução, com programática

Geração de XML).

No entanto, é baseado em XML de metadados realmente uma abordagem viável? A reação certa contra o uso excessivo de

XML pode ser visto na comunidade de desenvolvedores. Cada quadro e serviço parece exigir a sua própria XML descritores.

Em nossa opinião, há três razões principais para essa reação:

Muitos formatos de metadados existentes não foram projetadas para ser legível e fácil de editar manualmente. Em particular,

uma das principais causas de dor é a falta de padrões sensíveis para atributos e valores de elementos, exigindo digitando significativamente mais do que deveria ser necessário.

Metadados baseados em soluções foram muitas vezes utilizados de forma inadequada. Metadados não é, por natureza, mais flexíveis ou

manutenção do que simples código C #.

Bons editores XML, especialmente em IDEs, não são tão comuns como bom. NET ambientes de codificação. Pior, e mais facilmente solucionáveis, um XML Schema Definition (XSD) muitas vezes não é fornecida, evitando auto-completa e com êxito. Outro problema são XSDs que são demasiado genéricas, onde cada declaração é envolto em uma "extensão" genérica do elemento "meta" (como a abordagem de chave / valor).

Não há nenhum começar em torno da necessidade de baseado em texto de metadados no ORM. No entanto, NHibernate foi projetado com

plena consciência dos problemas típicos de metadados. O formato de metadados é extremamente legível e define útil valores padrão. Quando alguns valores estão faltando, NHibernate usa reflexão sobre a classe mapeada para ajudar determinar os padrões. NHibernate vem com um XSD documentado e completo. Finalmente, o apoio para o IDE XML tem melhorado ultimamente, e IDEs modernas fornecer validação XML dinâmico e até mesmo um auto-complete característica. Se isso não é suficiente para você, no capítulo 9 demonstramos algumas ferramentas que podem ser usados para gerar

NHibernate XML mapeamentos.

Vamos olhar para a maneira que você pode usar metadados XML em NHibernate. Nós introduzimos o mapeamento da classe

Categoria na seção anterior, agora vamos dar mais detalhes sobre a estrutura do seu mapeamento XML documento que é reproduzido na listagem 3.4.

Listagem 3.4 NHibernate mapeamento XML dos Categoria classe

```
<? Xml version = "1.0"?>
<Mapeamento hibernate-
  xmlns = "urn: nhibernate-mapping-2.2" | 1
  auto-import = "true"> | 2

  Class < | 3
    name = "CaveatEmptor.Model.Category, CaveatEmptor"
    lazy = "false">

    <id name = "Id"> | 4
      <generator class = "native" />
    </ Id>

    Propriedade < | 5
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    name = "Nome"
    coluna = "nome" />

<Many-to-one
    name = "ParentCategory"
    cascade = "all" />
</Class>
</Hibernate-mapping>

# 1 declaração Mapping
# 2 XSD declaração (opcional)
# 3 mapeamento de classe Categoria
# 4 mapeamento Identificador
# 5 de mapeamento propriedade Name
# 6 A referência a um outro empregado

```

| 6

Como você pode ver, um documento de mapeamento XML pode ser dividido em muitas partes:

Mapeamentos são declarados dentro de uma `<hibernate-mapping>` elemento. Você pode incluir muitos como classe mapeamentos como você gosta, junto com algumas outras declarações especiais que nós vamos falar mais tarde no livro.

O XSD mapeamento NHibernate é declarada, a fim de fornecer validação sintática do XML e muitos editores XML usar isso para auto-realização. Mas não é recomendável usar a cópia online deste arquivo por razões de desempenho.

A classe `Categoria` (Na assembléia `CaveatEmptor.Model`) É mapeado para a mesa do mesmo nome (`Categoria`). Cada linha nessa tabela representa uma instância do tipo `Categoria`.

Nós não temos muito discutido o conceito de identidade de objeto. Este tema complexo é coberto na seção 3.3. Para entender esse mapeamento, é suficiente para saber que cada registro na `Categoria` tabela terá um valor de chave primária que corresponda à identidade do objeto da instância na memória. O `<id>` cartografia elemento é usado para definir os detalhes da identidade do objeto.

A propriedade `Nome` é mapeado para uma coluna de banco de dados com o mesmo nome (`Nome`). NHibernate irá usar

.NET reflexão para descobrir o tipo da propriedade e deduzir como mapeá-lo para a coluna de SQL, assumindo que eles têm tipos compatíveis. Note que é possível especificar explicitamente o mapeamento de dados

tipo NHibernate que deve usar. Nós tomamos um olhar mais atento sobre estes tipos no capítulo 7, seção 7.1, "Compreender o sistema de tipo NHibernate".

Nós usamos uma associação para ligar uma categoria para outra. Aqui, é uma many-to-one associação. No banco de dados, a tabela `Categoria` contém uma coluna `ParentCategory` que é uma chave estrangeira para outro linha na mesma tabela. Mapeamentos de associação são mais complexas, por isso vamos voltar a eles no capítulo 4,

seção 4.6.

Embora seja possível declarar mapeamentos para várias classes em um arquivo de mapeamento usando múltiplos `<class>` elementos, a prática recomendada (e a prática esperado por algumas ferramentas NHibernate) é usar um arquivo de mapeamento por classe persistente. A convenção é dar ao arquivo o mesmo nome que a classe mapeada, anexando uma `.hbm` sufixo: por exemplo, `Categoria.hbm.xml`.

Às vezes você pode querer usar. Atributos NET ao invés de arquivos XML para a definição de seus mapeamentos, e assim brevemente explicar como isso pode ser feito em seguida. Depois disso, nós vamos passar a olhar mais de perto a natureza do mapeamentos de classe e propriedade aqui descritos nesta seção.

3.3.2 Atributo de programação orientada

Uma maneira de definir os metadados de mapeamento é de usar. Atributos NET. Desde sua primeira versão,. NET fornece

apoio para a classe / membro atributos. Introduzimos, no capítulo 2, o `NHibernate.Mapping.Attributes` biblioteca. Ele utiliza atributos diretamente embutidas no código-fonte. NET para fornecer todas as informações do NHibernate

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

necessidades para as classes do mapa. Tudo o que temos a fazer é marcar o código fonte .NET de nossas classes persistentes com

personalizado atributos .NET como mostrado na listagem 3.5.

listagem 3.5. Mapeamento com NHibernate.Mapping.Attributes

usando `NHibernate.Mapping.Attributes`:

```
[Class (lazy = false)]
Categoria public class {
    ...
    [Id (Name = "Id")]
    [Generator (1, Class = "nativo")]
    Id longa pública {
        ...
    }
    ...
    [Propriedade]
    Nome {public string
        ...
    }
    ...
}
```

É muito fácil de usar esse mapeamento com NHibernate:

```
cfg.AddInputStream (
    NHibernate.Mapping.Attributes.HbmSerializer.Default.Serialize (
        typeof (Categoria)));
```

Aqui, `NHibernate.Mapping.Attributes` gera um fluxo XML a partir do mapeamento da classe `Categoria` e esse fluxo é enviado para NHibernate configuração. Também é possível escrever esta informação de mapeamento no exterior

Documentos XML.

Mapeamento XML ou atributos. NET?

We have introduced mapping using XML mapping files and using

`NHibernate.Mapping.Attributes`. Embora você possa usar os dois ao mesmo tempo, é mais comum (e homogêneo) para usar apenas uma técnica. Sua escolha é baseada na maneira como você desenvolve sua aplicação. Você pode ler mais detalhes sobre os processos de desenvolvimento no capítulo 8.

Por enquanto, você já percebeu isso. Atributos .NET são muito mais convenientes e reduzir o linhas de metadados de forma significativa. Eles também são type-safe, suporte auto-completar em seu IDE como você tipo (como qualquer outro tipo C #), e fazer a refatoração de classes e propriedades mais fácil.

`NHibernate.Mapping.Attributes` é normalmente usado quando se inicia um novo projeto. Indiscutivelmente, mapeamentos de atributo são menos configurável no momento da implantação. No entanto, nada está parando de mão-de editar o XML gerado antes da implantação, pelo que esta objeção não é, provavelmente, significativo.

Por outro lado, os documentos de mapeamento XML são externos, o que significa que eles podem evoluir independentemente do seu modelo de domínio, pois eles também são mais fáceis de manipular para mapeamento muito complexo e

eles podem conter algumas informações úteis (não diretamente relacionado com o mapeamento das classes). É comum o uso de arquivos de mapeamento XML quando as classes já existem e não estão sob nosso controle.

Note-se que, em alguns casos, é melhor escrever mapeamento XML, por exemplo, ao lidar com uma alta componente personalizado ou mapeamento coleção. Nestes casos, você pode usar o atributo `[RawXml]` para inserir esse XML no seu mapeamento de atributo.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Você deve agora agarrou a idéia básica de como ambos os mapeamentos de XML e trabalhar mapeamentos de atributo. Próximo vamos olhar mais de perto os tipos de mapeamentos com mais detalhes, começando com propriedade e mapeamentos de classe.

3.4 propriedade básica e mapeamentos de classe

Nesta seção, você vai aprender uma série de recursos e dicas para escrever mapeamentos melhor. NHibernate pode "adivinhar" algumas informações, a fim de fazer o seu mapeamentos mais curtos. Também é possível configurá-lo para acessar sua entidades de uma maneira específica.

Vamos começar com uma profunda revisão do mapeamento de propriedades simples.

3.4.1 Resumo mapeamento Propriedade

Um mapeamento de propriedades típicas NHibernate define um nome de propriedade, um nome de coluna do banco de dados, eo nome de um NHibernate tipo. Ele mapeia uma propriedade .NET para uma coluna da tabela. A declaração de base prevê muitas variações e configurações opcionais, por exemplo, muitas vezes é possível omitir o nome do tipo. Assim, se **Descrição** é uma propriedade da (.NET) tipo **Corda**, NHibernate irá usar o tipo de NHibernate **Corda** por padrão (discute-se a NHibernate tipo de sistema no capítulo 7). NHibernate usa reflexão para determinar o tipo .NET da propriedade. Assim, os seguintes mapeamentos são equivalentes, enquanto eles estão na propriedade **Descrição**:

```
[Propriedade (Name = "Descrição" da coluna, = "DESCRIÇÃO" Type = "String")]
[Propriedade (= A coluna "DESCRIÇÃO")]
Descrição public string { ... }
```

Estes mapeamento pode ser escrito usando XML; os seguintes mapeamentos são equivalentes:

```
<property name="Description" column="DESCRIPTION" type="String"/>
<property name="Description" column="DESCRIPTION"/>
```

Como você já sabe, você pode omitir o nome da coluna se é o mesmo que o nome da propriedade, ignorando a caixa. (Este é um dos padrões sensíveis mencionados anteriormente.)

Em alguns casos, talvez seja necessário dizer NHibernate mais sobre a coluna de banco de dados do que apenas o seu nome. Para isso, você pode usar o **<column>** elemento em vez do **coluna** atributo. O **<column>** elemento fornece mais flexibilidade, que tem mais atributos opcionais e podem aparecer mais de uma vez. A propriedade dois seguintes mapeamentos são equivalentes:

```
[Propriedade]
[Column (1, Nome = "Descrição")]
Descrição public string { ... }
```

Usando XML, você pode escrever:

```
<property name="Description" type="string">
  <column name="DESCRIPTION"/>
</Property>
```

Porque. Atributos .NET não são ordenados, às vezes você precisa especificar a sua posição. Aqui, **[Column]** vem depois **[Propriedade]**. Então sua posição é 1; a posição do **[Propriedade]** é 0 (o valor padrão).

NHibernate.Mapping.Attributes imita mapeamento XML por isso, se você pode escrever um, você pode deduzir como escrever o outro. A principal diferença é que você não precisa especificar nomes com mapeamentos de atributo porque NHibernate.Mapping.Attributes pode adivinhar-lo com base em onde o mapeamento de atributo está no código. A exceção é **[Id]**. Você deve especificar o nome do identificador, quando se tem um, porque é opcional.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O `<property>` elemento (e especialmente o `<column>` elemento) também define certos atributos que se aplicam principalmente a geração automática do banco de dados esquema. Se você não estiver usando o `hbm2ddl` ferramenta (ver secção 10.1.1) para gerar automaticamente o esquema de banco de dados, você pode seguramente omitir isso. No entanto, ainda é preferível para incluir pelo menos os `não-nulo` atributo, já que NHibernate irá então ser capaz de relatório de propriedade nula ilegal valores sem ir ao banco de dados:

```
<property name="InitialPrice" column="INITIAL_PRICE" not-null="true"/>
```

Detecção de valores nulos ilegal é útil principalmente para a prestação de exceções sensível no tempo de desenvolvimento. Não é destinada para validação de dados verdadeiros, que está fora do escopo do NHibernate.

Algumas propriedades não são mapeadas para uma coluna em tudo. Em particular, um derivados propriedade tem seu valor a partir de um SQL expressão.

3.4.2 Usando propriedades derivadas

O valor de uma propriedade derivados é calculado em tempo de execução de avaliação de uma expressão. Você define o expressão usando a `fórmula` atributo. Por exemplo, para um `ShoppingCart` classe podemos mapear uma `TotalIncludingTax` propriedade, e porque é uma fórmula não há nenhuma coluna para armazenar esse valor na banco de dados:

```
<Nome da propriedade = "TotalIncludingTax"
fórmula = "+ TOTAL TAX_RATE * TOTAL"
type = "Double" />
```

A fórmula dada SQL é avaliado cada vez que a entidade é recuperado do banco de dados. Assim, o banco de dados faz o cálculo em vez do objeto. NET. A propriedade não tem um `coluna` atributo (ou sub-elemento) e nunca aparece em um SQL INSERT ou UPDATE, apenas em SELECTs. Fórmulas podem fazer referência a colunas da tabela de banco de dados, chamar funções SQL, SQL e incluem subselects.

Neste exemplo, o mapeamento de uma propriedade derivada de `Item`, Usa um subselect correlacionado para calcular a média quantidade de todos os lances para um item:

```
Propriedade <
  name = "AverageBidAmount"
  fórmula = "(selecione o AVG (b.AMOUNT) do BID b
            onde b.ITEM_ID item_id =)"
  type = "Double" />
```

Observe que os nomes de coluna não qualificado (é este o caso, aqueles que não precedida por uma `b.`) Referem-se a colunas da tabela do classe à qual pertence a propriedade derivados.

Como mencionamos anteriormente, NHibernate não requer propriedades em entidades se você definir uma nova propriedade estratégia de acesso. A próxima seção irá explicar as várias estratégias e quando você deve usá-los na sua mapeamento.

3.4.3 acesso estratégias de Propriedade

O `acesso` atributo permite que você especifique como NHibernate deve acessar os valores da entidade. O padrão estratégia, `propriedade`, Usa a assessores da propriedade - sendo estes os getters e setters você declara na sua classes. Em seu arquivo de mapeamento XML, o mapeamento de uma propriedade getter e setter da classe para uma coluna é bastante simples:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
<property name="Description"/>
```

Quando as cargas NHibernate ou salva um objeto, ele irá sempre usar o getter e setter definida para acessar o dados no objeto.

No nosso "Olá Mundo" exemplo no capítulo 2, você pode se lembrar que usamos o campo estratégia de acesso no arquivo de mapeamento XML para o nosso Empregado entidade. O campo strategy é útil para momentos em que você não

definido getters e setters de propriedade para suas classes. Nos bastidores, ele usa a reflexão para acessar a instância campo de classe diretamente. Por exemplo, o seguinte "propriedade" Mapeamento não requer um par getter / setter no classe, porque ele está usando campo estratégia de acesso:

```
<property name="nome" access="field"/>
```

Usando o campo estratégia de acesso podem ser úteis às vezes, mas o acesso através de getters e setters é de propriedade

considerados melhores práticas pela comunidade NHibernate; eles dão-lhe um nível extra de abstração entre o .NET modelo de domínio eo modelo de dados, além do já previsto pelo NHibernate. Propriedades também são mais flexíveis do que os campos, por exemplo, definições de propriedade pode ser substituído por subclasses persistentes.

NHibernate dá-lhe mais flexibilidade ao trabalhar com propriedades. Por exemplo, se o seu setters propriedade contém lógica de negócios? Muitas vezes, nós só queremos esta lógica a ser executada quando o nosso código cliente define

a propriedade, não durante o tempo de carga. Se uma classe é mapeado através de um setter de propriedade, NHibernate irá executar o código como

ele carrega o objeto. Felizmente, existem maneiras de lidar com esta situação. NHibernate nos dá um acesso especial estratégia chamada nosetter.* estratégia. Usando isso no seu mapeamento diz NHibernate usar a propriedade getter na leitura dos dados do objeto, mas para usar o campo subjacente, enquanto a gravação de dados a ele.

Se você achar que precisa de ainda mais flexibilidade do que isso, você pode aprender sobre estratégias de acesso disponíveis em outras o NHibernate documentação on-line de referência. Como um provador, se você quisesse NHibernate não usar o getter se você usa o modo padrão de nomear os campos em C #, que é caso de camel prefixado por um sublinhado (como _firstName), você pode mapeá-lo assim, usando NHibernate.Mapping.Attributes:

```
_firstName private string;
```

```
[Propriedade (Access = "field.camelcase-sublinhado")]
Nome {public string
    get {return _firstName;}
}
```

O mapeamento XML equivalente é:

```
<property name="FirstName" access="field.camelcase-underscore"/>
```

O bom efeito colateral desse exemplo é que, ao escrever consultas HQL NHibernate, você usa a mais legível nome da propriedade ao invés de feios nomes de campo. Nos bastidores, sabe NHibernate para ignorar a propriedade e em vez usar o campo ao carregar e salvar objetos. Porque nós estamos usando um campo, a propriedade é efetivamente ignorou-it não tem sequer a existir no código! Como um extra útil, se você queria fazer isso para todas as propriedades em uma classe, você pode especificar essa estratégia de acesso em nível de classe usando <Hibernate-mapping padrão="..."> acesso or the property HbmSerializer.HbmDefaultAccess when using NHibernate.Mapping.Attributes.

Se ainda precisar de mais flexibilidade, você pode definir sua estratégia de acesso aos próprios personalizado propriedade por

implementar a interface NHibernate.Property.IPropertyAccessor e nomeá-la na acesso atributo (usando seu nome totalmente qualificado). Usando NHibernate.Mapping.Attributes, você tem a alternativa de usando:

```
[Propriedade (accessType = typeof (MyPropertyAccessor))]
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Esta instalação (adicionando "**Type**" no final de um elemento para fornecer um tipo .NET em vez de uma string) está disponível em muitos outros lugares.

Até agora, temos agora aprendido a construir as classes para o nosso modelo de domínio, e então como podemos definir metadados de mapeamento de dizer NHibernate como persistir essas classes e seus membros. Você pode ver que NHibernate nos dá uma riqueza de recursos e flexibilidade, mas essencialmente estamos falando bastante straight-capacidades frente ORM. Em seguida, vamos olhar para alguns dos mais profundos "sob o capô" aspectos do NHibernate, incluindo a capacidade de desabilitar seu otimizador para ajudar com a depuração, a capacidade de impor que os objetos são imutáveis, impedindo NHibernate de inserir e atualizar-los, e alguns outros truques úteis que ajudá-lo a lidar com outros cenários espinhosos.

3.4.4 O aproveitamento do otimizador de reflexão

Mencionamos que NHibernate pode usar a reflexão para obter e definir propriedades de uma entidade em tempo de execução. Reflexão pode ser lenta, de modo NHibernate vai um passo além e usa um otimizador para acelerar este processo. O otimizador é ativado por padrão, e vai trabalhar como você criar fábricas de sua sessão. Devido a isso, você sofre um pequeno custo inicial, mas é geralmente vale a pena.

Dependendo da versão do .NET você está usando, NHibernate tem diferentes abordagens para a otimização reflexão.

Under .NET 1.1, NHibernate usa um **CodeDom** provedor. Este provedor gera classes especiais em tempo de execução que sabem sobre as entidades e seu negócio, e que pode acessá-las sem o uso de reflexão. Um pequeno aviso é que ele só funciona para propriedades públicas, portanto, você deve usar o padrão de acesso estratégico de "propriedade" em sua arquivos de mapeamento para obter os melhores resultados. Outra restrição é que identificadores entre aspas SQL (seção 3.5.6) não são suportados.

NHibernate 1.2 apresenta um outro fornecedor que só funciona (e é usado por padrão) abaixo .NET 2.0. Este provedor injeta métodos dinâmicos para entidades e seu negócio na inicialização, e é mais poderoso porque não está restrito a propriedades públicas.

Raramente é necessário, mas você pode desativar esse otimizador reflexão, atualizando seu arquivo de configuração:

```
<property name="hibernate.use_reflection_optimizer"> false </ property>
```

Ou, em tempo de execução,
usando:

```
Environment.UseReflectionOptimizer = false;
```

Isto pode ser útil ao depurar o aplicativo, porque em tempo de execução classes geradas são mais difíceis de rastrear. Você deve definir essa propriedade antes de realmente criar uma instância de configuração. Assim, você não pode usar o **<hibernate-configuration>** seção em seu arquivo de configuração (**hibernate.cfg.xml**, **web.config** etc) porque este é lido depois o objeto de configuração é criado (durante a chamada para os objetos de configuração **Configure ()** método).

Você pode selecionar o **CodeDom** provedor usando:

```
<property name="hibernate.bytecode.provider"> CodeDOM </ property>
```

O valor **CodeDOM** pode ser substituído por **nulo** (Para desativar o otimizador) ou **lcg** (Em. .NET 2.0 ou posterior). Note-se que **CodeDOM** podem não funcionar corretamente com tipos genéricos.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Você deve definir essa propriedade na `<nhibernate>` seção de seu arquivo de configuração do aplicativo. Em tempo de execução, antes de construir a fábrica de sessão, você pode definir a propriedade `Environment.BytecodeProvider` para o valor retornado pelo método estático `Environment.BuildBytecodeProvider()` ou a uma instância de seu próprio provedor que implementa a interface `NHibernate.Bytecode.IBytecodeProvider`.

A capacidade próxima interessante veremos é a capacidade de controlar insere banco de dados e atualizações para as classes e seus membros. Este nível de controle vem em muito útil quando você deseja criar objetos imutáveis, ou quando você quiser desabilitar as atualizações em uma base per-propriedade.

3.4.5 Controle de inserção e atualizações

Para propriedades que mapeiam para colunas, você pode controlar se eles aparecem no `INSERIR` declaração, utilizando o `inserir` atributo e se eles aparecem no `ATUALIZAÇÃO` declaração, utilizando o `atualizar` atributo.

A propriedade a seguir nunca é gravada no banco de dados:

```
<Nome da propriedade = "Nome"
  coluna = "NAME"
  type = "String"
  insert = "false"
  update = "false" />
```

A propriedade `Nome` da entidade é, portanto, imutável, que pode ser lido a partir do banco de dados, mas não modificada em qualquer forma. Se a classe completa é imutável, definir o `mutable = "false"` no mapeamento da classe. Se você está familiarizado com classes imutáveis, eles são basicamente apenas uma classe em que você decidiu que nunca deve ser atualizada depois de terem sido criados. Um exemplo pode ser um registro de transação financeira.

Além disso, temos também a `dynamic-insert` e `dynamic-update` atributos que informam NHibernate se deseja incluir valores de propriedade inalterada durante as inserções SQL e atualizações,

```
<Nome da classe = "NHibernate.Auction.Model.User, NHibernate.Auction"
  dinâmica inserir= "true"
  dynamic-update = "true">
  ...
</ Class>
```

Estes são ambos de nível de classe configurações que são desativados por padrão, de modo que quando NHibernate gera `INSERT` e `UPDATE` SQL para um objeto, ele faz isso para todas as propriedades no objeto, independentemente se eles realmente mudou desde que o objeto foi carregado. Permitindo uma dessas configurações fará NHibernate para gerar algum SQL em tempo de execução, em vez de usar o cache SQL em tempo de inicialização. O custo de desempenho e memória de fazer isso é geralmente pequenas. Além disso, deixando de fora as colunas em uma inserção (e, especialmente, em uma atualização) pode ocasionalmente

3.4.6 Usando identificadores entre aspas

SQL

Por padrão, NHibernate não cita nomes de tabela e coluna no SQL gerado. Isso torna o SQL um pouco mais legível e também nos permite tirar proveito do fato de que bancos de dados SQL são caso mais insensível ao comparar os identificadores não cotadas. De tempos em tempos, especialmente em bancos de dados legados, você identificadores encontro com caracteres estranhos ou espaços em branco, ou você pode querer forçar de maiúsculas e minúsculas.

Se você citar uma tabela ou nome de coluna com backticks no documento de mapeamento, NHibernate sempre citar este identificador no SQL gerado. As forças seguinte declaração de propriedade para gerar NHibernate SQL com o nome da coluna citada `"Descrição do item"`. NHibernate também vai saber que o SQL da Microsoft Servidor precisa a variação `[Description Item]` e que requer MySQL `^` Item Descrição`.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
<Nome da propriedade = "Description"  
    coluna = "Descrição do item` "/>
```

Não há nenhuma maneira, além de citar todos os nomes de tabela e coluna em backticks, para forçar a usar NHibernate identificadores entre aspas em todos os lugares.

NHibernate nos dá mais controle quando o mapeamento entre o nosso modelo de domínio eo esquema do banco de dados, por também deixar-nos controlar as convenções de nomenclatura. Discutimos isso em seguida.

3.4.7 Convenções de nomenclatura

Equipes de desenvolvimento têm, frequentemente, convenções estritas para a tabela e coluna nomes em suas bases de dados. NHibernate oferece um recurso que lhe permite reforçar os padrões de nomenclatura automaticamente.

Suponha que todos os nomes de tabela em CaveatEmptor deve seguir o padrão **CE_ name** <table>. Um solução é especificar manualmente um **tabela** atributo em todas as <class> e elementos de coleta em nosso mapeamento arquivos. Esta abordagem é demorado e facilmente esquecidas. Em vez disso, podemos implementar NHibernate **INamingStrategy** interface, como na listagem 3.6.

Listagem 3.6 **INamingStrategy** implementação

```
CENamingStrategy public class: {INamingStrategy  
    ClassToTableName cadeia pública (className string) {  
        retorno TableName (  
            StringHelper.Unqualify (className) ToUpper () );  
    }  
    PropertyToColumnName public string (string propertyName) {  
        retorno propertyName.ToUpper ();  
    }  
    TableName cadeia pública (tableName string) {  
        return "CE_" + tableName;  
    }  
    ColumnName public string (string columnName) {  
        columnName retorno;  
    }  
    PropertyToTableName cadeia pública (className string,  
        cadeia propertyName) {  
        ClassToTableName retorno (className) + '_' +  
            PropertyToColumnName (propertyName);  
    }  
}
```

O **ClassToTableName ()** método é chamado somente se um <class> mapeamento não especificar uma explícita **tabela** nome. O **PropertyToColumnName ()** método é chamado se uma propriedade não tem explícita **coluna** nome. O **TableName ()** e **ColumnName ()** métodos são chamados quando um nome explícito é declarada.

Se permitir que os nossos **CENamingStrategy**, Esta declaração mapeamento de classe

```
name="BankAccount"> <class
```

irá resultar em **CE_BANKACCOUNT** como o nome da tabela. O **ClassToTableName ()** método foi chamado com o nome de classe totalmente qualificado como o argumento.

No entanto, se um nome de tabela é especificado

```
<class name="BankAccount" table="BANK_ACCOUNT">  
então CE_BANK_ACCOUNT será o nome da tabela. Neste caso, BANK_ACCOUNT foi passado para o  
TableName () método.
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

A melhor característica do **INamingStrategy** é o potencial para o comportamento dinâmico. Para ativar um determinado nomeação de estratégia, podemos passar uma instância do NHibernate **Configuração** em tempo de execução:

```
Cfg configuração Configuração = new ();
cfg.NamingStrategy = new CENamingStrategy ();
ISessionFactory sessionFactory =
    .cfg.configure () BuildSessionFactory ();
```

Isso vai nos permitir ter várias **ISessionFactory** casos com base nos documentos mesmo mapeamento, cada um usando uma diferente **INamingStrategy**. Isto é extremamente útil em uma instalação multi-cliente onde nomes de tabela única (mas o mesmo modelo de dados) são necessários para cada cliente.

No entanto, a melhor maneira de lidar com este tipo de exigência é a utilização do conceito de um SQL esquema (Um tipo de namespace).

3.4.8 esquemas SQL

Esquemas SQL são um recurso disponível em muitos bancos de dados, incluindo SQL Server 2005 e MySQL. Eles permitem que você organizar seus objetos de banco de dados em grupos significativos. Por exemplo, o exemplo AdventureWorks banco de dados que vem com o Microsoft SQL Server 2005 define cinco esquemas: Recursos Humanos, Pessoa, Compras, Produção e Vendas. Todos estes esquemas viverem em um único banco de dados, e cada um tem suas próprias tabelas, pontos de vista e outros objetos de banco.

Muitos bancos de dados são projetados com apenas um esquema, e, portanto, você pode especificar um esquema padrão utilizando o **hibernate.default_schema** opção de configuração, existem alguns benefícios de desempenho pequeno em fazê-lo.

Alternativamente, se o seu banco de dados é como o AdventureWorks um, e tem muitos esquemas, você pode especificar o esquema para um documento de mapeamento particular, ou mesmo uma determinada classe ou mapeamento da coleção:

```
<Class >
  name = "NHibernateInAction.HelloWorld.Message,
  NHibernateInAction.HelloWorld "
  schema = "HelloWorld">
  ...
</Class>
</Hibernate-mapping>
```

Pode até ser declarado para todo o documento:

```
<Mapeamento hibernate-
  schema = "HelloWorld">
  ..
</Hibernate-mapping>
```

Em seguida, vamos discutir outra coisa útil que você pode fazer com o **<hibernate-mapping>** elemento, que é especificar um namespace padrão para suas classes para reduzir a duplicação.

3.4.9 Declarando os nomes das classes

Neste capítulo, nós introduzimos a aplicação CaveatEmptor. Todas as classes persistentes da aplicação são declaradas no namespace **NHibernate.Auction.Model** e são compilados na **NHibernate.Auction** montagem. Se tornaria tedioso para especificar este nome totalmente qualificado de cada vez chamamos uma classe em nossos documentos de mapeamento.

Vamos reconsiderar nosso mapeamento para o **Usuário** class (o arquivo **User.hbm.xml**):

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<? Xml version = "1.0"?>
<Hibernate-mapping xmlns = "urn: nhibernate-mapping-2.2"
    xmlns: xsi = http://www.w3.org/2001/XMLSchema-instance
    xsi: schemaLocation = "urn: nhibernate-mapping-2.2
http://nhibernate.sourceforge.net/schemas/nhibernate-mapping.xsd ">
    Class <
        name = "NHibernate.Auction.Model.User, NHibernate.Auction">
        ...
    </ Class>
</ Hibernate-mapping>

```

Nós não queremos repetir o nome completo sempre que esta ou qualquer outra classe é chamada em uma associação, subclasse, ou mapeamento de componentes. Portanto, vamos especificar uma vez **namespace** e **uma montagem**:

```

<Mapeamento hibernate-
    namespace = "NHibernate.Auction.Model"
    assembly = "NHibernate.Auction">
    Class <
        name = "Usuário">
        ...
    </ Class>
</ Hibernate-mapping>

```

Agora todos os nomes de classes não qualificadas que aparecem neste documento mapeamento será prefixado com o declarado nome do pacote. Nós assumimos esta definição em todos os exemplos de mapeamento neste livro. No entanto, essa configuração é mais inútil quando utilizando o mapeamento de atributo porque podemos especificar o tipo. NET e NHibernate.Mapping.Attributes vai escrever seu nome totalmente qualificado.

Você também pode usar os métodos **SetDefaultNamespace ()** e **SetDefaultAssembly ()** do classe **Configuração** para alcançar o mesmo resultado para toda a aplicação.

Nota: Não vamos mais escrever as informações XSD como clutters estes exemplos.

Ambas as abordagens que temos descrito até agora, XML e. Atributos NET, assume que todas as informações de mapeamento

é conhecido no momento da implantação. Suponha que alguma informação não é conhecida antes da aplicação é iniciada. Pode

você programaticamente manipular os metadados de mapeamento em tempo de execução?

3.4.10 Manipulação de metadados em tempo de

execução

Às vezes é útil para uma aplicação para navegar, manipular ou construir novos mapeamentos em tempo de execução.

Você pode

seguramente pular esta seção e voltar a ele mais tarde, quando você precisar. . NET fornece XML APIs que permitem que direta

manipulação de tempo de execução de documentos XML. Portanto, você pode criar ou manipular um documento XML em tempo de execução, antes de oferecê-la aos **Configuração** objeto.

No entanto, NHibernate também expõe um metamodelo de configuração tempo. O metamodelo contém todos os informações declaradas em seus documentos de mapeamento XML. Manipulação programática direta deste metamodelo Às vezes é útil, especialmente para aplicações que permitem a extensão por código escrito pelo usuário.

Por exemplo, o código a seguir adiciona uma nova propriedade, **Lema**, Para o **Usuário** mapeamento de classe:

```
PersistentClass userMapping cfg.GetMapping((User));
```

| 1

```
Coluna coluna = coluna nova (novo StringType (), 0);
column.Name = "LEMA";
column.Nullable = false;
column.Unique = true;
userMapping.Table.AddColumn (coluna);
```

| 2
| 2
| 2
| 2
| 2

```
SimpleValue valor = new SimpleValue ();
```

| 3

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

value.Table = userMapping.Table;	3
value.AddColumn (coluna);	3
value.Type = new StringType ();	3
Propriedade Propriedade prop = new ();	4
prop.Value = valor;	4
prop.Name = "Motto";	4
userMapping.AddProperty (prop);	4
ISessionFactory sf = cfg.BuildSessionFactory ();	5
1. Obter informações de mapeamento para o usuário a partir da	
2. configuração	
3. Definir uma nova coluna para a tabela USER	
4. Enrole a coluna em um valor	
5. Definir uma nova propriedade da classe Usuário	
Construir uma fábrica nova sessão, usando o novo mapeamento	

A **PersistentClass** objeto representa o metamodelo para uma única classe persistente; que recuperá-lo do **Configuração.Coluna**, **SimpleValue** E **Propriedade** são todas as classes do metamodelo e NHibernate estão disponíveis no namespace **NHibernate.Mapping**; A classe **StringType** está no namespace **NHibernate.Type**. Tenha em mente que a adição de uma propriedade para uma existente mapeamento de classe persistente como mostrado aqui é fácil, mas programaticamente criar um novo mapeamento para uma classe previamente mapeados é um pouco mais envolvidas.

Uma vez que um **ISessionFactory** é criado, seus mapeamentos são imutáveis. De fato, o **ISessionFactory** utiliza um metamodelo diferentes internamente do que o utilizado no momento da configuração. Não há nenhuma maneira de voltar a o original **Configuração** a partir do **ISessionFactory** ou **ISession**. No entanto, o pedido pode **readtheISessionFactory**"SmetamodelbycallingGetClassMetadata ()ou **GetCollectionMetadata ()**. Por exemplo:

```
Usuário user = ...;
ClassMetadata meta = sessionFactory.GetClassMetadata (typeof (User));
string [] = meta.PropertyNames meta.GetPropertyNames ();
object [] = propertyValues meta.GetPropertyValues (usuário);
```

Este trecho de código recupera os nomes de propriedades persistentes do **Usuário** classe e os valores dessas propriedades de uma instância específica. Isso ajuda você a escrever código genérico. Por exemplo, você pode usar este recurso para rotular componentes UI ou melhorar a saída de log.

Agora vamos voltar a um elemento de mapeamento especial que você viu na maioria dos nossos exemplos anteriores, a identificador mapeamento de propriedade. Começaremos discutindo a noção de identidade de objeto.

Compreender a identidade do objeto 3,5

É vital para compreender a diferença entre identidade de objeto e igualdade do objeto antes de discutirmos termos como identidade do banco de dados e como NHibernate gerencia identidade. Precisamos estes conceitos se quisermos concluir mapeamento nossa **CaveatEmptor** classes persistentes e suas associações com NHibernate.

3.5.1 Identidade versus igualdade

. NET desenvolvedores a entender a diferença entre. NET identidade e igualdade. Identidade do objeto, **Object.ReferenceEquals ()**, É uma noção definida pelo ambiente CLR. Duas referências a objetos são idênticos se eles apontam para o mesmo local de memória.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Por outro lado, a igualdade de objeto é uma noção definida por classes que implementam a `Equals()` método (ou o operador `==`). Às vezes também referida como equivalência. Equivalência significa que duas diferentes (não-objetos) idênticos têm o mesmo valor. Duas instâncias diferentes de `corda` são iguais se eles representam o mesmo seqüência de caracteres, mesmo que cada um tem sua própria localização no espaço de memória do virtual máquina. (Temos que admitir que isso não é inteiramente verdadeiro para `cordas`, mas essa é a idéia.)

Persistência complica esse quadro. Com o objeto / persistência relacional, um objeto persistente é um in-memory representação de uma determinada linha de uma tabela do banco de dados. Assim, junto com. Identidade NET (posição de memória) e igualdade do objeto, pegamos identidade do banco de dados (Localização no armazenamento de dados persistente). Agora temos três métodos para a identificação de objetos:

Objeto de identidade-Objects são idênticos se eles ocupam a mesma posição de memória. Isto pode ser verificado usando `Object.ReferenceEquals()`.

Objeto da igualdade-Objects são iguais se eles têm o mesmo valor, conforme definido pelo `Equals(object o)` método. Classes que não explicitamente substituir esse método herdar a implementação definida por `System.Object`, Que compara a identidade do objeto.

Banco de dados de identidade-Objects armazenadas em um banco de dados relacional são idênticos se eles representam a mesma linha

ou, equivalentemente, compartilhar a mesma mesa e valor de chave primária.

Você precisa entender como a identidade do banco de dados refere-se objeto de identidade em NHibernate.

3.5.2 identidade banco de dados com NHibernate

NHibernate expõe identidade banco de dados para a aplicação de duas maneiras:

O valor da propriedade identificador de uma instância persistente

O valor retornado por `ISession.GetIdentifier(object o)`

A propriedade identificador é especial: seu valor é o valor de chave primária da linha de banco de dados representada pelo instância persistente. Nós geralmente não apresentam a propriedade identificador no nosso domínio modelo é uma preocupação relacionada, não faz parte do nosso problema de negócios. Em nossos exemplos, a propriedade identificador é sempre chamado `id`.

Então, se `MyCategory` é uma instância de `Categoria`, Chamando `myCategory.Id` retorna o valor de chave primária a linha representada por `MyCategory` no banco de dados.

Você deve fazer a propriedade para o escopo identificador público ou privado? Bem, identificadores de banco de dados são

muitas vezes usada pelo aplicativo como um punho conveniente a uma instância particular, mesmo fora da camada de persistência.

Por exemplo, aplicações web, muitas vezes exibir os resultados de uma tela de busca para o usuário como uma lista de resumo

da informação. Quando o usuário seleciona um elemento particular, a aplicação pode precisar para recuperar o selecionado

objeto. É comum o uso de uma pesquisa por identificador para este fim-você provavelmente já usou identificadores Desta forma, mesmo em aplicações utilizando ADO.NET direta. É, portanto, geralmente apropriada para expor completamente a

identidade do banco de dados com uma propriedade de identificador público.

Por outro lado, nós geralmente não implementar um `conjunto` assessor para o identificador (neste caso, NHibernate usa. reflexo NET para modificar o campo identificador). E nós também costumam deixar NHibernate gerar o identificador valor. As exceções a esta regra são as classes com chaves naturais, onde o valor do identificador é atribuído pelo a aplicação antes que o objeto é feito persistente, em vez de ser gerado pelo NHibernate. (Discutimos chaves naturais na próxima seção). NHibernate não permite que você altere o valor do identificador de um persistente exemplo, após o primeiro atribuído. Lembre-se, parte da definição de uma chave primária é que seu valor deve nunca mudar. Vamos implementar uma propriedade de identificador para o `Categoria` classe e mapeá-lo com atributos NET.:

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

Categoria public class {
    id longo privado;
    ...
    [Id (Name = "Id" da coluna, = "category_id", Access = "nosetter.camelcase")]
    [Generator (I, Class = "nativo")]
    Id longa pública {
        get {return this.id;}
    }
    ...
}

```

O tipo de propriedade depende do tipo de chave primária da **CATEGORIA** tabela eo tipo de mapeamento NHibernate. Esta informação é determinada pelo **<id>** elemento no documento de mapeamento. Aqui está o mapeamento XML:

```

<class name="Category" table="CATEGORY">
    <id name="Id" column="CATEGORY_ID" access="nosetter.camelcase">
        <generator class="native"/>
    </Id>
    ...
</Class>

```

A propriedade identificador é mapeado para a coluna de chave primária **Category_id** da tabela **CATEGORIA**. O NHibernate tipo para essa propriedade é **longo**, Que mapeia a um **BIGINT** tipo de coluna, na maioria dos bancos de dados e que também tem sido escolhida para coincidir com o tipo do valor de identidade produzido pela **nativo** gerador de identificador. (Nós

discutir estratégias de geração de identificador na próxima seção) A estratégia utilizada para o acesso aqui.: **access = "nosetter.camelcase"** diz para NHibernate que não há acessador set e que ele deve usar a transformação camelCase para deduzir o nome do campo de identidade usando o nome da propriedade. Se possível, NHibernate irá usar um otimizador de reflexão para evitar custos de reflexão (explicado mais adiante neste capítulo).

Assim, além de operações para o teste. Identidade do objeto NET (**Object.ReferenceEquals (a, b)**) e igualdade do objeto (**a.equals (b)**), Você pode agora usar **a.Id == b.Id** para testar identidade de banco de dados.

Uma abordagem alternativa para tratamento de banco de dados de identidade é a de não implementar qualquer propriedade identificador, e deixe NHibernate gerenciar a identidade do banco de dados internamente. Neste caso, você omitir o **nome** atributo no mapeamento declaração:

```

<id column="CATEGORY_ID">
    <generator class="native"/>
</Id>

```

NHibernate irá agora gerir os valores identificador internamente. Você pode obter o valor do identificador de uma instância persistente da seguinte forma:

```
longo catid = session.GetIdentifier (long) (categoria);
```

Esta técnica tem uma desvantagem séria: você não pode mais usar NHibernate para manipular objetos soltos efetivamente (ver capítulo 4, seção 4.1.5, "Fora do âmbito de identidade"). Então, você deve sempre usar identificador propriedades em NHibernate. Se você não gosta deles sendo visível para o resto de sua aplicação, tornar a propriedade protegidos ou privados.

Usando identificadores de banco de dados em NHibernate é fácil e simples. Escolhendo uma boa chave primária (e estratégia de geração de chave) poderia ser mais difícil. Discutimos essas questões que vem.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.5.3 Escolhendo as chaves primárias

Você tem que dizer NHibernate sobre sua estratégia preferida geração de chave primária. Mas primeiro, vamos definir chave primária.

O chave candidata é uma coluna ou conjunto de colunas que identifica uma linha específica da tabela. A principais candidatos devem satisfazer as seguintes propriedades:

O valor ou valores nunca são nulos.

Cada linha tem um valor único ou valores.

O valor ou valores de uma linha específica nunca mudam.

Para uma determinada tabela, várias colunas ou combinações de colunas podem satisfazer essas propriedades. Se uma tabela tem identificação de apenas um atributo, é por definição, o chave primária. Se existem múltiplas chaves candidatas, você precisa para escolher entre elas (as chaves candidato não escolhido como a chave primária deve ser declarada como chaves únicas na banco de dados). Se houver não colunas exclusivas ou combinações únicas de colunas, e, portanto, nenhum candidato chaves, então a tabela é, por definição, não uma relação, tal como definido pelo modelo relacional (que permite linhas duplicadas), e você deve repensar seu modelo de dados.

Muitos modelos de dados SQL legado uso natural chaves primárias. Uma chave natural é uma chave com significado de negócios: uma atributo ou combinação de atributos que é único em virtude da semântica seus negócios. Exemplos de recursos naturais chaves pode ser um EUA Social Security Number ou Número do Arquivo Australian Tax. Distinguir chaves naturais é simples: se um atributo de chave candidata tem significado fora do contexto de banco de dados, é uma chave natural, ou não é gerado automaticamente.

A experiência tem mostrado que as chaves naturais quase sempre causam problemas a longo prazo. A principal boa chave deve ser único, constante e necessária (nunca nulo ou desconhecido). Atributos de entidade muito poucos satisfazer essas requisitos, e alguns que não são eficientemente indexáveis pelos bancos de dados SQL. Além disso, você deve fazer absolutamente certo de que uma definição chave candidata nunca poderia mudar durante toda a vida do banco de dados antes de promovê-lo para uma chave primária. Alterar a definição de uma chave primária e todas as chaves estrangeiras que se referem a é uma tarefa frustrante.

Por estas razões, nós recomendamos fortemente que novas aplicações usam identificadores sintética (também chamada chaves substitutas). Chaves substitutas não têm significado, eles são valores únicos de negócios gerados pelo banco de dados ou aplicação. Há uma série de abordagens bem conhecidas, para a geração de chave substituta.

NHibernate tem vários built-in estratégias de geração de identificador. Listamos as opções mais úteis na tabela 3.1.

nativo	O gerador de identidade nativa pega geradores outra identidade como seqüência de identidade, ou bilo dependendo das capacidades do banco de dados subjacente.
identidade	Este gerador suporta colunas de identidade em DB2, MySQL, MS SQL Server, Sybase, e Informix. O identificador retornado pelo banco de dados é convertido para o tipo de propriedade usando Convert.ChangeType. Qualquer tipo de propriedade integral é assim, apoiado.
seqüência	A seqüência em DB2, PostgreSQL, Oracle, SAP DB, McKoi, Firebird é usado. O identificador devolvido pelo banco de dados é convertido para o tipo de propriedade usando Convert.ChangeType. Qualquer tipo de propriedade integral é assim, apoiado.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

incremento	Na inicialização do NHibernate, este gerador lê o valor da coluna máxima de chave primária a mesa e aumenta o valor por um cada vez que uma nova linha é inserida. O identificador gerado pode ser de qualquer tipo integral. Este gerador é especialmente eficiente se o único servidor de aplicação NHibernate tem acesso exclusivo ao banco de dados, mas não deve ser usado em qualquer outro cenário (como em clusters).
hilo	Um algoritmo de alto / baixo é uma forma eficiente para gerar identificadores de qualquer tipo integral, dada uma tabela e coluna (por padrão hibernate_unique_key e next_hi, respectivamente) como fonte de valores ci. O algoritmo de alta / baixa gera identificadores que são únicos apenas para um banco de dados particular. Veja [Ambler 2002] para obter mais informações sobre o abordagem de alto / baixo para identificadores únicos. Não use este gerador com uma fornecida pelo usuário conexão.
uuid.hex	Este gerador usa System.Guid e seu ToString (formato) método para gerar identificadores do tipo string. O comprimento da string retornada depende da configuração formato. Esta estratégia de geração não é popular, desde CHAR chaves primárias consumir mais espaço de banco de dados do teclado numérico e são ligeiramente mais lento.
guid	Este gerador é usado quando o tipo do identificador é Guid. O identificador deve ter Guid.Empty como valor padrão. Ao salvar a entidade, este gerador irá atribuir-lhe um novo valor usando Guid.NewGuid () .
guid.comb	Este gerador é similar ao anterior. No entanto, ele usa outro algoritmo que torna quase tão rapidamente como quando se usa inteiros (especialmente ao salvar em um SQL Server banco de dados). Além disso, os valores gerados são ordenados, você pode usar uma parte destes valores como números de referência, por exemplo.

Você não está limitado a estes as estratégias integradas e, há alguns outros que você pode descobrir através da leitura Documentação de referência do NHibernate. Você também pode criar o seu gerador próprio identificador através da implementação de NHibernate **IIdentifierGenerator** interface. É até possível misturar geradores identificador classes persistentes em um modelo de domínio único, mas para o legado de dados não recomendamos o uso do mesmo gerador para todas as classes.

O especial **atribuído** estratégia do gerador identificador é mais útil para entidades com natural chaves primárias. Esta estratégia permite que o aplicativo atribuir valores identificador definindo a propriedade identificador antes de fazer a persistentes objeto chamando **Save()**. Esta estratégia tem alguns inconvenientes graves quando você está trabalhando com objetos destacados e persistência transitiva (ambos os conceitos são discutidos no próximo capítulo). Não use **atribuído** identificadores se você pode evitá-los, é muito mais fácil usar uma chave substituta primário gerado por uma das estratégias listadas na tabela 3.1.

Para dados legados, o quadro é mais complicado. Neste caso, estamos muitas vezes preso com chaves naturais especialmente chaves compostas (Teclas natural composto de múltiplas colunas da tabela). Porque um identificador composto pode ser mais difícil de trabalhar, só discuti-las no contexto do capítulo 9, seção 9.2, "Legacy esquemas e chaves compostas."

O próximo passo é adicionar propriedades de identificação para as classes da aplicação CaveatEmptor. Fazer todos classes persistentes têm sua identidade própria base de dados? Para responder a esta pergunta, devemos explorar a distinção entre entidades e tipos de valor em NHibernate. Estes conceitos são necessários para granulação fina modelagem de objetos.

3,6 refinada modelos de objetos

Um dos principais objetivos do projeto é o suporte para NHibernate de grão fino modelos de objetos, que foram isoladas como o requisito mais importante para um modelo de domínio rico. É uma razão que escolhemos POCOs.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Em termos brutos, de grão fino significa "classes mais tabelas." Por exemplo, um usuário pode ter tanto um endereço de cobrança e um endereço residencial. No banco de dados, poderíamos ter um único **USUÁRIO** tabela com as colunas

BILLING_STREET,BILLING_CITY E **BILLING_ZIPCODE** juntamente com **HOME_STREET,HOME_CITY**, e **HOME_ZIPCODE**. Há boas razões para utilizar este modelo um tanto desordenado relacional (Performance, por exemplo).

No nosso modelo de objeto, podemos usar a mesma abordagem, representando os dois endereços como seis cordas valorizado propriedades do **Usuário** classe. Mas seria muito melhor este modelo usando uma **Endereço** classe, onde **Usuário** tem o **BillingAddress** e **HomeAddress** propriedades.

Esse modelo de objeto atinge melhoria da coesão social e maior reutilização de código e é mais compreensível. No passado, as soluções ORM muitos não deram um bom suporte para esse tipo de mapeamento.

NHibernate enfatiza a utilidade de grão fino classes para implementar tipo de segurança e comportamento. Por exemplo, muitas pessoas modelo de um endereço de e-mail como uma cadeia de valor de propriedade de **Usuário**. Sugerimos que

uma abordagem mais sofisticada é a definição de um real **EmailAddress** classe que poderia acrescentar mais alto nível semântica e comportamento. Por exemplo, pode fornecer uma **SendEmail ()** método.

3.6.1 tipos de entidade e valor

Isso nos leva a uma distinção de importância central na ORM. . Em NET, todas as classes são em pé de igualdade: Todos objetos têm sua própria identidade e ciclo de vida, e todas as instâncias de classe são passados por referência. Apenas tipos são passados por valor.

Estamos defendendo um projeto em que há classes mais persistente do que tabelas. Uma linha representa vários objetos. Porque a identidade do banco de dados é implementado por valor de chave primária, alguns objetos persistentes não terá sua própria identidade. Com efeito, o mecanismo de persistência implementa passagem por valor-semântica para algumas classes. Um dos objetos representados na linha tem sua própria identidade, e outros que dependem.

NHibernate faz a seguinte distinção essencial:

Um objeto do tipo entidade tem sua identidade própria base de dados (valor de chave primária). Uma referência de objeto para um entidade é mantida como uma referência no banco de dados (um valor de chave estrangeira). Uma entidade tem o seu próprio ciclo de vida, que pode persistir independentemente de qualquer outra entidade. Um objeto do tipo valor tem sua identidade do banco de dados, que pertence a uma entidade, e seu estado persistente é incorporados na linha da tabela da entidade proprietária (exceto no caso de coleções, que são também tipos de valor considerado, como você verá no capítulo 6). Tipos de valor não têm identificadores ou identificador propriedades. A vida útil de uma instância de tipo de valor é limitado pelo tempo de vida da entidade proprietária.

Os tipos de valor mais óbvias são objetos simples, como **Cordas** e **Número inteiros**. NHibernate também permite tratar uma classe definida pelo usuário como um tipo de valor, como você verá a seguir. (Nós também voltar a este conceito importante no capítulo 6, seção 6.1, "Compreender o sistema de tipo NHibernate.")

3.6.2 Usando componentes

Até agora, as classes do nosso modelo de objeto foram todas as classes de entidade com seu ciclo de vida e identidade próprias. O

Usuário classe, no entanto, tem um tipo especial de associação com a **Endereço** classe, conforme mostrado na figura 3.5.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

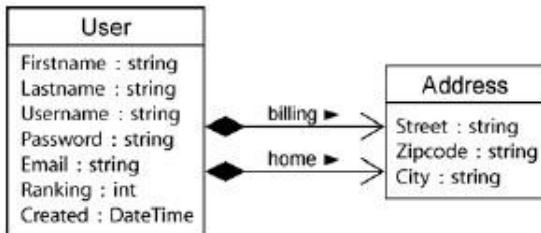


Figura 3.5 As relações entre Usuário e Endereço usando composição

Em termos de modelagem de objetos, essa associação é uma espécie de agregação de um "Parte de" relacionamento. Agregação

é uma forma forte de associação: Tem semântica adicional em relação ao ciclo de vida de objetos. No nosso caso, temos uma forma ainda mais forte, composição, onde o ciclo de vida de parte é dependente do ciclo de vida do inteiro.

Especialistas objeto de modelagem UML e designers vão alegar que não há diferença entre este composição e outros estilos mais fracos da associação quando se trata da implementação .NET. Mas no contexto das ORM, há uma grande diferença: uma classe composta muitas vezes é um tipo de valor candidato.

Nós agora mapa **Endereço** como um tipo de valor e **Usuário** como uma entidade. Isso afeta a implementação da nossa As classes POCO?

.NET em si não tem conceito de classe de composição, um ou atributo não pode ser marcado como um componente ou composição. A única diferença é o identificador do objeto: um componente não tem identidade, daí a persistente classe componente requer nenhuma propriedade identificador ou identificador de mapeamento. A composição entre **Usuário** e **Endereço** é uma noção em nível de metadados, só temos que dizer que o NHibernate **Endereço** é um tipo de valor na documento de mapeamento.

NHibernate usa o termo componente para uma classe definida pelo usuário que é mantido para a mesma tabela como a possuir entidade, como mostrado na listagem 3.7. (O uso da palavra componente aqui não tem nada a ver com a nível de arquitetura conceito, como em componente de software.)

Mapeamento de listagem 3.7 a **Usuário** classe com um componente **Endereço**

```

Class <
    name = "User"
    table = "USER">
<Id>
    name = "Id"
    coluna = "USER_ID"
    type = "Int64">
    <generator class="native"/>
</Id>
Propriedade <
    name = "Username"
    coluna = "username"
    type = "String" />
Componente <
    name = "HomeAddress"
    class = "Endereço">
    <Nome da propriedade = "Street" | 1
        type = "String"
        coluna = "HOME_STREET"
        not-null = "true" />
    <Nome da propriedade = "Cidade"
        type = "String"
        coluna = "HOME_CITY"
        not-null = "true" />
    <Nome da propriedade = "CEP"
        type = "Int16"
        coluna = "HOME_ZIPCODE"
        not-null = "true" />
</ Component>
Componente <
  
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

name = "BillingAddress"
class = "Endereço"
<Nome da propriedade = "Street"
  type = "String"
  coluna = "BILLING_STREET"
  not-null = "true" />
<Nome da propriedade = "Cidade"
  type = "String"
  coluna = "BILLING_CITY"
  not-null = "true" />
<Nome da propriedade = "CEP"
  type = "Int16"
  coluna = "BILLING_ZIPCODE"
  not-null = "true" />
</ Component>
...
</ Class>

```

1 Declare atributos persistentes
2 classe componente Reutilização

Em # 1, nós declaramos os atributos persistentes de **Endereço** dentro do **<component>** elemento. A propriedade do **Usuário** classe é chamado **HomeAddress**. Em # 2, reutilizar a classe mesmo componente para mapear uma outra propriedade de

deste tipo para a mesma tabela.

Figura 3.6 mostra como os atributos do **Endereço** classe são persistentes para a mesma tabela como a **Usuário** entidade.

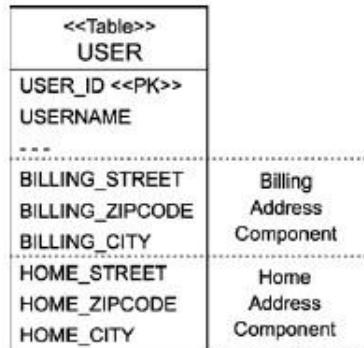


Figura 3.6 Tabela de atributos **Usuário** com **Endereço** componente

Componentes podem ser mais difíceis de mapa com NHibernate.Mapping.Attributes. Ao usar um componente em muitos classes com o mapeamento idênticos, é muito fácil de fazer (muito mais fácil do que com o mapeamento XML):

```

[Componente]
Endereço public class {
  [Propriedade (NotNull = true)]
  {public string rua ... }
  [Propriedade (NotNull = true)]
  Cidade {public string ... }
  [Propriedade (NotNull = true)]
  CEP pública de curto {... }
}
[Classe]
class User {
  ...
  [ComponentProperty]
  HomeAddress Endereço pública {... }
}
[Classe]
Casa da classe {
  ...
  [ComponentProperty]
}

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        Localização Endereço pública { ... }

    }

Mas a classe Usuário tem dois endereços, cada mapeados para colunas diferentes. Há muitas maneiras de mapeá-los
(Você pode descobri-los no apêndice B). Aqui está uma solução:

[Classe]
class User {
    ...
    [Componente (Name = "HomeAddress", ClassType = typeof (Endereço))]
    HomeAddressMapping classe protegida {
        [Propriedade (coluna = "HOME_STREET", NotNull = true)]
        {public string rua ...}
        [Propriedade (coluna = "HOME_CITY", NotNull = true)]
        Cidade {public string ...}
        [Propriedade (coluna = "HOME_ZIPCODE", NotNull = true)]
        CEP pública de curto { ... }
    }
    HomeAddress Endereço pública { ... }
    [Componente (Name = "BillingAddress", ClassType = typeof (Endereço))]
    BillingAddressMapping classe protegida {
        [Propriedade (coluna = "BILLING_STREET", NotNull = true)]
        {public string rua ...}
        [Propriedade (coluna = "BILLING_CITY", NotNull = true)]
        Cidade {public string ...}
        [Propriedade (coluna = "BILLING_ZIPCODE", NotNull = true)]
        CEP pública de curto { ... }
    }
    BillingAddress Endereço pública { ... }
}

```

Simulamos a hierarquia do mapeamento XML usando as classes **HomeAddressMapping** e **BillingAddressMapping** cuja única finalidade é fornecer o mapeamento. Note-se que NHibernate.Mapping.Attributes irá escolher automaticamente-los porque eles pertencem à classe **Usuário**. Este solução não é muito elegante. Felizmente, você não terá que lidar com este tipo de mapeamento com muita freqüência. Sempre que você pensar que o mapeamento XML seria mais fácil de usar do que atributos, você pode usar o **[RawXml]** atributo a integrar este XML dentro de seus atributos. Este é provavelmente o caso aqui, para que possamos incluir o XML mapeamento dos componentes em 3.7 lista como esta:

```

[Classe]
class User {
    ...
    [RawXml (Depois = typeof (ComponentAttribute) Conteúdo, = @ "
        componente> name=""HomeAddress"">
        ...
        </Component>")]
    HomeAddress Endereço pública { ... }
    ...
}

```

O **[RawXml]** atributo tem duas propriedades: **Depois** diz após o qual tipo de mapeamento do XML deve ser inserido; na maioria das vezes, é o tipo do atributo que é definido no XML. Esta propriedade é opcional, caso em que o XML é inserido no topo do mapeamento. A segunda propriedade é **Conteúdo**. É a cadeia contendo o XML para incluir.

Note que neste exemplo, nós modelamos a associação composição como unidirecional. Não podemos navegar de **Endereço** para **Usuário**. NHibernate suporta tanto unidirecional e bidirecional composições; No entanto, a composição unidirecional é muito mais comum. Aqui está um exemplo de um mapeamento bidirecional:

```

Componente <
    name = "HomeAddress"
    class = "Endereço">
    <parent name="Owner"/>
    <property name="Street" type="String" column="HOME_STREET"/>
    <property name="City" type="String" column="HOME_CITY"/>

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
<property name="Zipcode" type="short" column="HOME_ZIPCODE"/>
</Component>
```

O **<parent>** mapas da propriedade elemento de um tipo **Usuário** à entidade proprietária, neste exemplo, a propriedade é nomeado **Proprietário**. Em seguida, chamamos **Address.Owner** para navegar em outra direção.

Um componente NHibernate pode possuir outros componentes e até mesmo associações a outras entidades. Esta flexibilidade é a base de apoio do NHibernate para os modelos de grão fino objeto. (Discutiremos vários componentes mapeamentos no capítulo 6.)

No entanto, existem duas limitações importantes para classes mapeadas como componentes:

Referências compartilhadas não são possíveis. O componente **Endereço** não tem sua identidade própria base de dados

(Chave primária) e assim por um determinado **Endereço** objeto não pode ser referido por qualquer outro objeto que o

~~Exemplo anterior de Endereço~~ não pode representar uma referência nula para um **Endereço**. Em vez de uma abordagem elegante,

NHibernate representa os componentes nulo como valores nulos em todas as colunas mapeadas do componente. Este

significa que se você guardar um objeto componente com todos os valores de propriedade null, NHibernate irá retornar um valor nulo.

Finalmente, é também possível fazer um componente imutáveis, usando:
`<Component ... insert = "false", update = "false" />`

Supor para granulação fina classes não é o único ingrediente de um modelo de domínio rico. Herança de classe e polimorfismo está definindo características do objeto modelos orientados.

3.7 herança de classe Mapping

Uma estratégia simples para as classes de mapeamento para tabelas de banco de dados pode ser "uma tabela para cada classe." Esta abordagem

Parece simples, e funciona bem até encontrar herança.

Herança é a característica mais visível da incompatibilidade estrutural entre o orientada a objeto e relacionais mundos. Orientada a objeto modelo de ambos os sistemas "é um" e "tem um". SQL modelos baseados em fornecer apenas

"Tem um" relações entre as entidades.

Há três abordagens diferentes para representar uma hierarquia de herança. Estes foram catalogados por Scott Ambler [Ambler 2002] em seu amplamente lido papel "objetos de mapeamento para bancos de dados relacionais":

Tabela por classe concreta Descarte- polimorfismo e relações de herança completamente da modelo relacional

Tabela por hierarquia de classes-Habilitar polimorfismo por desnormalizar o modelo relacional e usando um coluna do tipo discriminador para manter informações de tipo

Tabela por subclasse-Representar "É um" (herança) relacionamentos como "tem um" (chave estrangeira)

~~relações~~ Esta seção tem uma de cima para baixo abordagem, ele assume que estamos começando com um modelo de domínio e tentando

derivar um esquema novo SQL. No entanto, as estratégias de mapeamento descritas são tão relevantes, se estamos trabalhando

baixo para cima, começando com as tabelas do banco de dados existente.

3.7.1 Tabela por classe concreta

Suponha que ficar com a abordagem mais simples: Nós poderíamos usar exatamente uma tabela para cada classe (não abstrata). Todos

propriedades de uma classe, incluindo propriedades herdadas, poderia ser mapeados para colunas dessa tabela, como mostrado na figura 3.7.

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

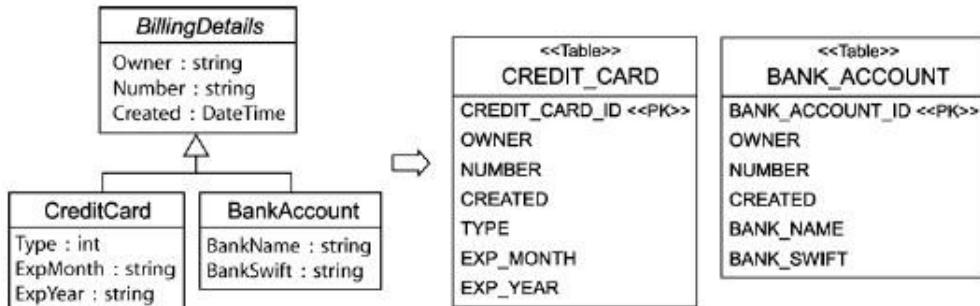


Figura 3.7 Mapping uma composição bidirecional

O principal problema com essa abordagem é que ele não suporta associações polimórficas muito bem. No banco de dados, as associações são geralmente representados como relacionamentos de chave estrangeira. Na figura 3.7, se o subclasse são todos mapeados para tabelas diferentes, uma associação polimórfica para sua classe base (abstrata **BillingDetails** em neste exemplo) não pode ser representada como uma relação simples chave estrangeira. Isso seria problemático para a nossa modelo de domínio, porque **BillingDetails** está associada com **Usuário**, Daí ambas as tabelas precisaria de um estrangeiro referência fundamental para a **USUÁRIO** mesa.

Consultas polimórficas (Consultas que retornam objetos de todas as classes que correspondem a interface da classe consultado) também são problemáticos. Uma consulta contra a classe base deve ser executado como SQL vários **SELEÇÃO**s, um para cada subclasse concreta. Poderemos ser capazes de usar um SQL **UNIÃO** para melhorar o desempenho, evitando múltiplas viagens de ida e volta ao banco de dados. Contudo, os sindicatos são um pouco nonportable e de outra maneira difícil de trabalhar.

NHibernate não suporta o uso de sindicatos no momento de escrever, e sempre use várias consultas SQL. Para uma consulta contra a **BillingDetails** classe (por exemplo, restringindo a uma determinada data de criação),

```
NHibernate<CREDIT_CARD> Propriedade SQL
de CREDIT_CARD
CRIADO onde =?
selecionar BANK_ACCOUNT_ID, proprietário, o número, CRIADO, BANK_NAME, ...
de BANK_ACCOUNT
CRIADO onde =?
```

Observe que uma consulta separada é necessária para cada subclasse concreta.

Por outro lado, as consultas contra as classes concretas são triviais e um bom desempenho:

```
selecionar CREDIT_CARD_ID, TYPE, EXP_MONTH, EXP_YEAR
de CREDIT_CARD onde CREATED =?
```

Note que aqui, e em outros lugares neste livro, mostramos que é SQL conceitualmente idêntico ao SQL executado pelo NHibernate. O SQL real pode parecer superficialmente diferentes.

Um outro problema conceitual com esta estratégia de mapeamento é que várias colunas diferentes de diferentes tabelas compartilham a mesma semântica. Isso faz com que a evolução do esquema mais complexo. Por exemplo, uma mudança para uma base classe resultados propriedade de tipo de alterações a várias colunas. Ele também torna muito mais difícil de implementar banco de dados de restrições de integridade que se aplicam a todas as subclases.

Esta estratégia de mapeamento não requer nenhuma declaração especial de mapeamento NHibernate: Basta criar um novo **<class>** declaração para cada classe concreta, especificando um diferente **tabela** atributo para cada um. Recomendamos esta abordagem (apenas) para o nível superior da sua hierarquia de classe, onde polimorfismo não é geralmente necessária.

3.7.2 Tabela por hierarquia de classe

Alternativamente, uma hierarquia de classe inteira pode ser mapeada para uma única tabela. Esta tabela deverá incluir colunas para todas as propriedades de todas as classes na hierarquia. A subclasse concreta representada por uma linha específica é identificada pelo valor de um tipo discriminador coluna. Essa abordagem é mostrada na figura 3.8.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

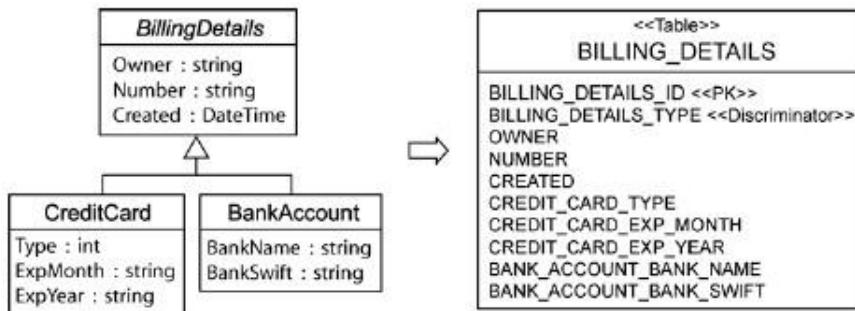


Figura 3.8 Tabela por hierarquia de classes de mapeamento

Esta estratégia de mapeamento é um vencedor em termos de desempenho e simplicidade. É o melhor desempenho forma de representar o polimorfismo, tanto consultas polimórficas e nonpolymorphic executar bem e é ainda fácil de implementar à mão. Ad hoc de relatórios é possível sem joins complexos ou sindicatos, e esquema evolução é simples.

Há um grande problema: Colunas de propriedades declaradas por subclasses deve ser declarado anulável. Se o seu subclasses cada definir várias propriedades não-nulo, a perda de NOT NULL restrições poderiam ser um problema sério do ponto de vista da integridade dos dados.

Em NHibernate, usamos o <subclass> elemento para indicar um mapeamento hierarquia table-per-classe, como no listagem 3.8.

Listagem 3.8 NHibernate <subclass> cartografia

```
<hibernate-mapping>
  Class <| 1
    name = "BillingDetails"
    table = "BILLING_DETAILS" discriminador valor = "BD">
      <Id <| 1
        name = "Id"
        coluna = "BILLING_DETAILS_ID"
        type = "Int64"
        <generator class="native"/>
      </Id> <| 2
      <Discriminador <| 2
        coluna = "BILLING_DETAILS_TYPE"
        type = "String" />
      <Propriedade <| 3
        name = "Nome"
        coluna = "PROPRIETÁRIO"
        type = "String" />
      ...
      <Subclasse <| 4
        name = "CreditCard"
        discriminator-value = "CC">
        <Propriedade <| 4
          name = "Tipo"
          coluna = "CREDIT_CARD_TYPE" />
        ...
      </Subclasse />
      ...
    </Class>
  </hibernate-mapping>
```

```
# 1 classe Root, mapeada para tabela
# 2 coluna discriminadora
# 3 mapeamentos de propriedades
# 4 subclasse CreditCard
```

1 A classe raiz **BillingDetails** da hierarquia de herança é mapeada para a tabela **BILLING_DETAILS**.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2 Temos que usar uma coluna especial para distinguir entre as classes persistentes: o discriminador. Esta não é uma propriedade da classe persistente, é usado internamente pelo NHibernate. O nome da coluna é **BILLING_DETAILS_TYPE**, e os valores serão strings, neste caso, "CC" ou "BA". NHibernate irá automaticamente e recuperar os valores discriminador.

3 Propriedades da classe base são mapeados como sempre, com um <property> elemento.

4 Cada subclasse tem suas próprias <subclass> elemento. Propriedades de uma subclasse são mapeados para colunas no **BILLING_DETAILS** mesa. Lembre-se que **não-nulo** restrições não são permitidos, porque um **CreditCard** instância não terá um **BankSwift** propriedade eo **BANK_ACCOUNT_BANK_SWIFT** campo deve ser nulo para nessa linha.

O <subclass> elemento pode, por sua vez contêm outros <subclass> elementos, até que toda a hierarquia é mapeada para a tabela. A <subclass> elemento não pode conter um <joined-subclass> elemento. (A <joined-subclass> elemento é usado na especificação da opção de mapeamento de terceiros: uma tabela por subclasse. Esta opção é discutido na próxima seção.) A estratégia de mapeamento não pode ser mudado mais neste ponto.

Aqui estão as classes mapeadas usando NHibernate.Mapping.Attributes:

```
[Classe (Tabela = "BILLING_DETAILS", DiscriminatorValue = "BD")]
pública BillingDetails classe {
    [Id (Name = "Id" da coluna, = "BILLING_DETAILS_ID")]
    [Generator (1, Class = "nativo")]
    [Discriminador (2 Coluna, = "BILLING_DETAILS_TYPE"
        TypeType = typeof (string))]
    Id longo público {...}
    [Propriedade (coluna = "Proprietário")]
    Nome {public string ...}
```

```
[Subclasse (DiscriminatorValue = "CC")]
CreditCard public class: {BillingDetails
    [Propriedade (coluna = "CREDIT_CARD_TYPE")]
    Tipo CreditCardType público {...}
}
...
}
```

Lembre-se que quando queremos especificar uma classe no mapeamento, podemos adicionar "**Type**" ao nome do elemento;

para o atributo **[Discriminador]**, Foi utilizado **TypeType**. Note que este atributo pode ser escrito antes de qualquer propriedade, uma vez que não está ligada a qualquer campo / propriedade da classe (se houver mais do que este atributo na propriedade, certifique-se que vem depois do **[Id]** e antes de os outros atributos).

NHibernate usaria o SQL seguinte ao consultar o **BillingDetails** classe:
selecionar BILLING_DETAILS_ID, BILLING_DETAILS_TYPE,
 PROPRIETÁRIO, ..., CREDIT_CARD_TYPE,
de BILLING_DETAILS
CRIADO onde =?

Para consultar o **CreditCard** subclasse, NHibernate usaria uma condição para o discriminador:

```
selecionar BILLING_DETAILS_ID,
    CREDIT_CARD_TYPE, CREDIT_CARD_EXP_MONTH, ...
de BILLING_DETAILS
onde BILLING_DETAILS_TYPE = "CC" e CREATED =?
```

Como poderia ser mais simples que isso?

Em vez de ter um campo discriminador, é possível usar uma fórmula de SQL arbitrários. Por exemplo:

```
<Tipo discriminador = "String"
    fórmula = "quando CREDIT_CARD_TYPE caso for nulo, end 'BD' else 'CC'"
/>
```

Aqui, usamos a coluna **CREDIT_CARD_TYPE** para avaliar o tipo.

Agora, vamos descobrir a alternativa para table-per-class-hierarchy.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.7.3 Tabela por subclasse

A terceira opção é para representar relações de herança como relacional associações estrangeiras chave. Cada subclasse

que declara propriedades persistentes, incluindo classes abstratas e até mesmo interfaces tem sua própria tabela.

Ao contrário da estratégia que usa uma tabela por classe concreta, a tabela contém colunas aqui apenas para cada não

herdado propriedade (cada propriedade declarado pelo próprio subclasse), juntamente com uma chave primária que é também uma política externa

base. Essa abordagem é mostrada na figura 3.9.

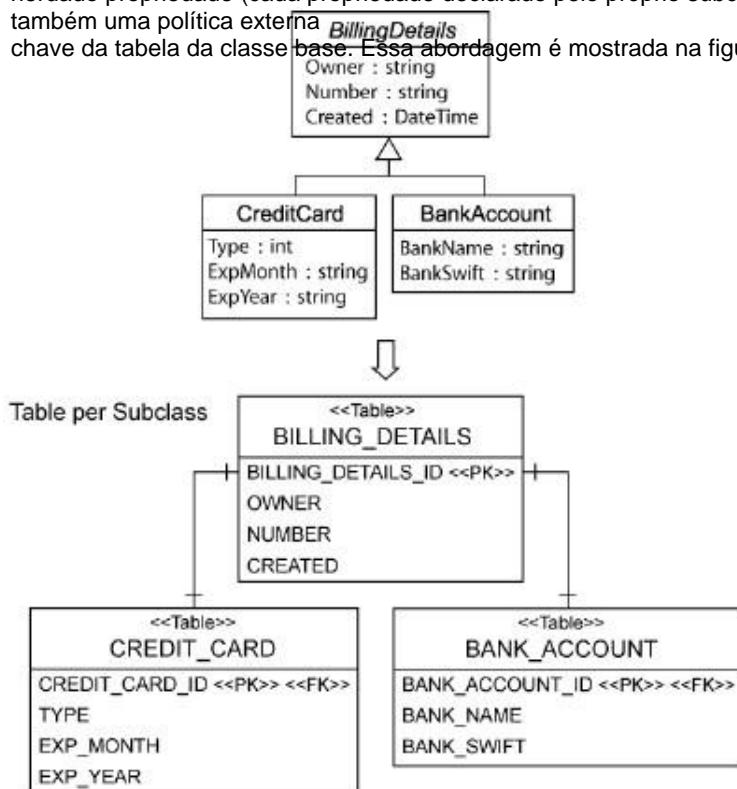


Figura 3.9 Tabela por subclasse de mapeamento

Se uma instância do **CreditCard** subclasse é feita persistente, os valores das propriedades declaradas pelo **BillingDetails** classe base são persistentes para uma nova linha do **BILLING_DETAILS** mesa. Apenas os valores de propriedades declaradas pela subclasse são persistentes para a nova linha do **CREDIT_CARD** mesa. As duas linhas são ligados entre si por seu valor de chave compartilhada primária. Mais tarde, a instância da subclasse podem ser obtidos a partir do

banco de dados juntando-se a tabela da subclasse com a tabela da classe base.

A principal vantagem desta estratégia é que o modelo relacional é totalmente normalizada. Evolução do esquema e definição de restrição de integridade são simples. A associação polimórfica para uma determinada subclasse pode ser representado como uma chave estrangeira apontando para a tabela dessa subclasse.

Em NHibernate, usamos o **<joined-subclass>** elemento para indicar um mapeamento table-per-subclass (ver listagem 3.9).

Listagem 3.9 NHibernate **<joined-subclass>** cartografia

```
<? Xml version = "1.0"?>
<hibernate-mapping>
    Class <
        name = "BillingDetails"
        table = "BILLING_DETAILS">
        <Id
            name = "Id">
    </Id>
</Class>
<joined-subclass
    name = "CreditCard"
    table = "CREDIT_CARD">
    <id
        name = "Id">
        <column
            name = "BILLING_DETAILS_ID">
    </column>
</id>
<many-to-one
    name = "BillingDetails"
    column = "BILLING_DETAILS_ID">
    <!-- This is the inverse side -->
</many-to-one>
<column
    name = "TYPE">
</column>
<column
    name = "EXP_MONTH">
</column>
<column
    name = "EXP_YEAR">
</column>
</joined-subclass>
<joined-subclass
    name = "BankAccount"
    table = "BANK_ACCOUNT">
    <id
        name = "Id">
        <column
            name = "BILLING_DETAILS_ID">
    </column>
</id>
<many-to-one
    name = "BillingDetails"
    column = "BILLING_DETAILS_ID">
    <!-- This is the inverse side -->
</many-to-one>
<column
    name = "BANK_NAME">
</column>
<column
    name = "BANK_SWIFT">
</column>
</joined-subclass>
</hibernate-mapping>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    coluna = "BILLING_DETAILS_ID"
    type = "Int64">
    <generator class="native"/>
</Id>
Propriedade <
    name = "Owner"
    coluna = "PROPRIETÁRIO"
    type = "String" />
...
<Subclasse juntou-
    name = "CreditCard"
    table = "CREDIT_CARD">
    <key column="CREDIT_CARD_ID">
Propriedade <
    name = "Tipo"
    coluna = "TIPO" />
...
</Joined-subclass>
...
</Class>
</Hibernate-mapping>

```

1 BillingDetails classe raiz, mapeada para tabela BILLING_DETAILS

2 elemento <joined-subclass>

Chave # 3 Primária / estrangeiros

1 Mais uma vez, a classe raiz **BillingDetails** é mapeada para a tabela **BILLING_DETAILS**. Note que não há discriminador é necessária com esta estratégia.

2 O novo <joined-subclass> elemento é usado para mapear uma subclasse para uma nova tabela (neste exemplo, **CREDIT_CARD**). Todas as propriedades declaradas na subclasse juntou serão mapeadas para esta tabela. Note que nós intencionalmente deixado de fora o exemplo de mapeamento para **BankAccount**, Que é semelhante ao **CreditCard**.

3 A chave primária é necessária para o **CREDIT_CARD** tabela, que também terá uma restrição de chave estrangeira para a chave primária da **BILLING_DETAILS** mesa. A **CreditCard** objeto de pesquisa exigirá uma junção de ambos os tabelas.

A <joined-subclass> elemento pode conter outros <joined-subclass> elementos, mas não um <subclass> elemento. NHibernate não suporta a mistura dessas duas estratégias de mapeamento.

NHibernate irá usar uma junção externa ao consultar o **BillingDetails** classe:

```

selecionar BD.BILLING_DETAILS_ID, BD.OWNER, BD.NUMER, BD.CREATED,
    CC.TYPE, ..., BA.BANK_SWIFT, ...
caso
    CC.CREDIT_CARD_ID quando não for nulo, um
        BA.BANK_ACCOUNT_ID quando não é nulo, em seguida, 2
        BD.BILLING_DETAILS_ID quando não é nulo, em seguida, 0
    final como TIPO
BILLING_DETAILS de BD
    left join em CC CREDIT_CARD
        BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
    left join BA BANK_ACCOUNT em
        BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
onde = BD.CREATED?

```

O SQL **caso** instrução utiliza a existência (ou inexistência) de linhas nas tabelas subclasse **CREDIT_CARD** e **BANK_ACCOUNT** para determinar a subclasse concreta de uma determinada linha do **BILLING_DETAILS** mesa.

Para restringir a consulta para a subclasse, NHibernate usa uma junção interna em vez disso:

```

selecionar BD.BILLING_DETAILS_ID, BD.OWNER, BD.CREATED, CC.TYPE, ...
CREDIT_CARD de CC
    junção interna BILLING_DETAILS BD em
        BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
onde = CC.CREATED?

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Como você pode ver, esta estratégia de mapeamento é mais difícil de implementar com a mão mesmo ad-hoc de relatórios será mais complexa. Esta é uma consideração importante se você pretende misturar código NHibernate com escrita à mão SQL / ADO.NET. (Para relatórios ad hoc, vistas banco de dados fornecem uma forma de compensar a complexidade da mesa-per-subclasse estratégia. A visão pode ser usada para transformar o modelo de mesa-per-subclass no muito mais simples de mesa por hierarquia-modelo.)

Além disso, embora esta estratégia de mapeamento é enganosamente simples, nossa experiência é que o desempenho pode ser inaceitável para a hierarquia de classes complexa. Consultas sempre exigir quer participar de um todo muitas tabelas ou muitas leituras seqüenciais. Nossa problema deve ser reformulado como como escolher um adequado combinação de mapeamento

9.7. Escolhendo uma estratégia

Onde usar Esta seção aplica-se a aplicações hierárquicas de classe. Um projeto modelo típico de domínio tem uma mistura de interfaces e classes abstratas.

Você pode aplicar todas as estratégias de mapeamento para classes abstratas e interfaces. Interfaces podem não ter estado, mas pode conter declarações de propriedade, para que possam ser tratados como classes abstratas. Você pode mapear uma interface usando

`<class>,<subclass>` Ou `<joined-subclass>`, E você pode mapear qualquer propriedade declarado ou herdadas utilização `<property>`. NHibernate não vai tentar instanciar uma classe abstrata, no entanto, mesmo se você consulta ou de -lo.

Aqui **está o quando** para grande maioria das associações polimórficas ou consultas, inclinar-se para a mesa-per-concrete-class estratégia. Se você precisar de associações polimórficas (uma associação a uma classe base, portanto, a todas as classes em a hierarquia, com resolução dinâmica da classe concreta em tempo de execução) ou consultas e subclasses declarar propriedades relativamente poucos (principalmente se a principal diferença entre subclasses está em sua comportamento), inclinar-se para o modelo table-per-class-hierarchy.

Se você precisar de associações polimórficas ou consultas, e subclasses declarar muitas propriedades (subclasses diferem principalmente pelos dados que possuem), inclinar-se para a abordagem table-per-subclass. Por padrão, escolha table-per-class-hierarchy para problemas simples. Para casos mais complexos (ou quando você está anuladas por um modelador de dados insistindo sobre a importância de restrições de nulidade), você deve considerar o table-per-subclass estratégia. Mas naquele momento, pergunte-se se não seria melhor remodelar como herança Delegação no modelo de objeto. Herança complexa muitas vezes é melhor evitar para todos os tipos de razões não relacionadas com persistência ou ORM. NHibernate funciona como um amortecedor entre o objeto e os modelos relacionais, mas que não significa que você pode ignorar completamente as preocupações persistência na elaboração de seu modelo de objeto.

Note-se que você também pode usar `<subclass>` e `<joined-subclass>` elementos de mapeamento em um separado arquivo de mapeamento (como um elemento de nível superior, em vez de `<class>`). Então você tem que declarar a classe que é estendida (Por exemplo, `<subclass name="CreditCard" extends="BillingDetails">`), E classe da base mapeamento deve ser carregado antes o arquivo de mapeamento da subclasse. Esta técnica permite estender uma classe hierarquia, sem modificar o arquivo de mapeamento da classe base. NHibernate.Mapping.Attributes usando, você pode mover-se a implementação de `CreditCard` em outro arquivo e mapeá-la assim:

```
[Subclasse (ExtendsType = typeof (BillingDetails), DiscriminatorValue = "CC")]
CreditCard public class: { BillingDetails
    ...
}
```

Agora você já viu os meandros do mapeamento de uma entidade de forma isolada. Na próxima seção, nos voltamos para o problema mapeamento de associações entre entidades, que é outra questão importante decorrente do objeto / relacional incompatibilidade de paradigma.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.8 associações Apresentando

Gerenciando as associações entre classes e as relações entre as tabelas é a alma do ORM. A maioria dos os difíceis problemas envolvidos na implementação de uma solução ORM se relacionam com associação de gestão.

O NHibernate modelo de associação é extremamente rico, mas não é isenta de riscos, especialmente para novos usuários. Em

Nesta seção, não vai tentar cobrir todas as combinações possíveis. O que vamos fazer é examinar determinados casos que são extremamente comuns. Voltamos ao assunto de mapeamentos associação no capítulo 7, para uma mais completa o tratamento.

Mas primeiro, há algo que precisamos para explicar na frente.

3.8.1 associações unidirecionais

Ao usar (digitado) DataSets, associações são representadas como na base de dados. Para ligar duas entidades, você tem que definir a chave estrangeira em uma entidade para a chave primária do outro. Não há a noção de coleção, assim você não pode adicionar uma entidade a uma coleção e obter a associação criada.

Transparente implementações POCO orientada persistência NHibernate como fornecer suporte para coleções. Mas é importante entender que, as associações NHibernate são inherentemente unidirecional. Como quanto NHibernate está em causa, a associação de **Oferta** para **Item** é uma associação diferentes do que o associação de **Item** para **Oferta**. Isso significa que **bid.Item item = e item.Bids.Add (bid)** são dois operações não relacionadas.

Para algumas pessoas, isso parece estranho, para outros, ele se sente completamente natural. Afinal de contas, as associações na nível de linguagem são sempre afirmações unidirecional e NHibernate para implementar persistência para a planície. NET objetos. Vamos apenas observar que esta decisão foi tomada porque os objetos NHibernate não estão vinculados a qualquer contexto. Em aplicações NHibernate, o comportamento de uma instância não-persistente é o mesmo que o comportamento de um instância persistente.

Porque as associações são tão importantes, precisamos de uma linguagem muito precisas para classificá-los.

3.8.2 Multiplicidade

Em descrever e classificar as associações, nós quase sempre usar a associação multiplicidade. Olhar para a figura 3.10.

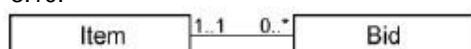


Figura 3.10 Relação entre **Item** e **Oferta**

Para nós, a multiplicidade é apenas dois bits de informação:

Pode haver mais de um **Oferta** para um determinado **Item**?

Pode haver mais de um **Item** para um determinado **Oferta**?

Depois de olhar para o modelo de objeto, concluímos que a associação de **Oferta** para **Item** é uma many-to-one associação. Lembrando que as associações são direcionais, teríamos também chamar a associação inversa da **Item** para **Oferta** umum-para-muitos associação.

(É claro, existem mais duas possibilidades: many-to-many e um-para-um; vamos voltar a esses possibilidades no capítulo 6.)

No contexto de persistência do objeto, não estamos interessados em saber se "muitos" realmente significa "dois" ou "Máximo de cinco" ou "sem restrições."

3.8.3 A associação mais simples possível

A associação de **Oferta** para **Item** é um exemplo do tipo mais simples possível de associação na ORM. O referência de objeto retornado por **bid.Item** é facilmente mapeada para uma coluna de chave estrangeira na **BID** mesa. Em primeiro lugar,

aqui está a implementação da classe C # da **Oferta** mapeados usando atributos NET.:
[Classe (Tabela = "BID")]

```
Licitacao public class {
    ...
    item Item privado;
    [ManyToOne (coluna = "item_id", NotNull = true)]
    do Item pública {
        get {artigo do retorno;}
        set {item = valor;}
    }
    ...
}
```

Em seguida, aqui está o mapeamento correspondente NHibernate para esta associação:

```
name = "Oferta"
table = "BID">
...
<Many-to-one
    name = "Item"
    coluna = "item_id"
    class = "Item"
    not-null = "true" />
</Class>
```

Este mapeamento é chamado de unidirecional many-to-one associação. A coluna **Item_id** no **BID** tabela é uma chave estrangeira para a chave primária da **ITEM** mesa.

Temos especificado explicitamente a classe **Item**, Que a associação se refere. Esta especificação é geralmente opcional, desde que NHibernate pode determinar esta reflexão usando.

Nós especificamos o **não-nulo** atributo porque não podemos ter uma oferta sem um item. O **não-nulo** atributo não afeta o comportamento de tempo de execução do NHibernate, neste caso, mas existe principalmente para controle automático definição de dados linguagem de geração (DDL) (ver capítulo 10).

Em alguns bancos de dados legados, pode acontecer que um many-to-one pontos associação a uma entidade inexistente. O propriedade **não encontrado** permite que você definir como NHibernate deve reagir a esta situação.

```
<Many-to-one ... não encontrado = "ignore | exceção" />
```

Utilização **não encontrado = "exceção"** (O valor padrão), NHibernate irá lançar uma exceção. E **não-found = "ignore"** NHibernate fará ignorar esta associação (deixando-a nula).

3.8.4 Fazendo a associação bidirecional

Até aqui tudo bem. Mas também precisamos ser capazes de facilmente buscar todas as propostas para um determinado item. Precisamos de uma associação bidirecional aqui, por isso temos de adicionar o código de andaimes para a **Item** classe:

```
class Item pública {
    ...
    lances ISET private = HashSet new ();
    Licitações públicas ISET {
        get {lances return;}
        set {lances = valor;}
    }
    AddBid public void (bid Bid) {
        bid.Item = this;
        bids.Add (bid);
    }
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
}
```

Você pode pensar do código em `AddBid ()` (Um método de conveniência), como implementar uma forte bidirecional associação no modelo de objeto.

Um mapeamento básico para este um-para-muitos associação ficaria assim:

```
[Set]
  [Key (1 Coluna, = "item_id")]
  [OneToMany (2, ClassType = typeof (Bid))]
Licitações públicas ISET { ... }
```

Aqui está o XML equivalente envolto em seu mapeamento da classe:

```
name = "Item"
table = "ITEM"
...
<set name="Bids">
  <key column="ITEM_ID"/>
  <one-to-many class="Bid"/>
</Set>
</Class>
```

O mapeamento da coluna definida pelo `<key>` elemento é uma coluna de chave estrangeira do associado **BID** mesa. Observe que especificar a coluna externa mesma chave neste mapeamento da coleção que especificado no mapeamento para a associação many-to-one. A estrutura da tabela para este mapeamento da associação é mostrada na figura 3.11.

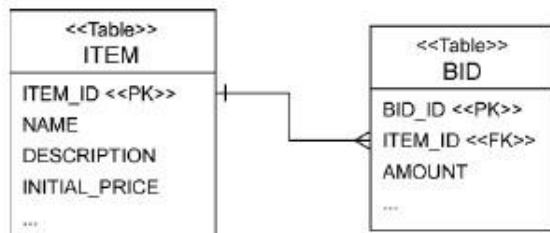


Figura 3.11 Tabela relacionamentos e chaves para um mapeamento one-to-many/many-to-one

Agora temos dois diferentes associações unidirecionais mapeados para a mesma chave estrangeira, o que representa um problema. Em tempo de execução, existem duas diferentes representações na memória do mesmo valor de chave estrangeira: o

`item` propriedade da **Oferta** e um elemento da **licitações** coleção mantida por um **Item**. Suponha que a nossa aplicação modifica a associação, por exemplo, a adição de um lance para um item neste fragmento do `AddBid ()` método:

```
bid.Item = this;
bids.Add (bid);
```

Este código é bom, mas nesta situação, NHibernate detecta duas mudanças diferentes ao em memória persistente instâncias. Do ponto de vista do banco de dados, apenas um valor deve ser atualizado para refletir essas alterações: a `Item_id` coluna do **BID** mesa. NHibernate não transparente detectar o fato de que as duas alterações referem-se para a coluna de mesmo banco de dados, uma vez que neste momento nós não fizemos nada para indicar que este é um bidirecional associação.

Precisamos de mais uma coisa em nosso mapeamento da associação de dizer NHibernate para tratar isso como uma bidirecional

associação: A `inverso` atributo diz NHibernate que a coleção é uma imagem espelho de muitos-para-um associação do outro lado:

```
Class <
  name = "Item"
  table = "ITEM"
  ...
<Set
  name = "lances"
  inverse = "true">
  <key column="ITEM_ID"/>
  <one-to-many class="Bid"/>
</Set>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
</ Class>
```

Sem a `inverso` atributo, NHibernate seria tentar executar duas instruções SQL diferentes, tanto a atualização a coluna externa mesma chave, quando manipular a associação entre as duas instâncias. Especificando `inverse = "true"`, Que dizer explicitamente que NHibernate final da associação deve sincronizar com a banco de dados. Neste exemplo, dizemos NHibernate que deve propagar as alterações feitas no `Oferta` final do associação ao banco de dados, ignorando as alterações feitas apenas para o `Licitações` coleção. Assim, se apenas uma chamada

`item.Bids.Add (bid)`, Não serão feitas alterações persistentes. Isto é consistente com o comportamento in. NET sem NHibernate: Se uma associação é bidirecional, você tem que criar o link em dois lados, não apenas um.

Agora temos um ambiente de trabalho associação many-to-one bidirecional (Que também poderia ser chamado de bidirecional um-para-muitos associação, é claro).

Um último está faltando. Nós exploramos a noção de persistência transitiva em muito maior detalhe na próximo capítulo. Por agora, iremos introduzir os conceitos de salvar em cascata e exclusão em cascata, que precisamos em

Para terminar o nosso mapeamento da associação.

Quando instanciar um novo `Oferta` e adicioná-lo a um `Item`, A oferta deve se tornar persistente imediatamente. Nós gostaria de evitar a necessidade de explicitamente fazer uma `Oferta` persistentes chamando `Save ()` no `ISession` interface.

Fazemos uma mexida final para o documento de mapeamento para permitir salvar em cascata:

```
Class <
  name = "Item"
  table = "ITEM">
  ...
  <Set
    name = "Licitações"
    inverse = "true"
    cascade = "save-update">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
  </ Set>
</ Class>
```

O `cascata` atributo diz NHibernate para fazer qualquer nova `Oferta` persistente instância (isto é, salve-o no banco de dados) se o `Oferta` é referenciado por um persistente `Item`.

O `cascata` atributo é direcional: Aplica-se a apenas uma das extremidades da associação. Poderíamos também especificar

`cascade = "save-update"` para o many-to-one associação declarada no mapeamento para `Oferta`, Mas isso não faria sentido neste caso, porque `Ofertas` são criados depois `Items`.

Terminamos? Não é bem assim. Ainda precisamos definir o ciclo de vida para ambas as entidades em nossa associação.

3.8.5 A relação pai / filho

Com o mapeamento anterior, a associação entre `Oferta` e `Item` é bastante frouxa. Gostaríamos de usar esse mapeamento em um sistema real, se ambas as entidades tiveram seus próprio ciclo de vida e foram criados e removidos em negócios não relacionados

processos. Certas associações são muito mais fortes do que isso, algumas entidades estão unidas para que seus ciclos de vida não são verdadeiramente independentes. No nosso exemplo, parece razoável que a supressão de um item implica

eliminação de todas as propostas para o item. Um caso particular referências lance apenas uma instância um item para a sua inteira

tempo de vida. Neste caso, ambos os salva em cascata e exclusões faz sentido.

Se habilitar em cascata excluir, a associação entre `Item` e `Oferta` é chamado de pai / filho relacionamento.

Em uma relação pai / filho, a entidade-mãe é responsável pelo ciclo de vida de seu filho entidades associadas.

Esta é a mesma semântica como uma composição (usando componentes NHibernate), mas neste caso apenas as entidades são

envolvidos; `Oferta` não é um tipo de valor. A vantagem de usar uma relação pai / filho é que a criança pode ser

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

carregados individualmente ou referenciados diretamente por outra entidade. A oferta, por exemplo, podem ser carregados e

manipulado sem recuperar o item possuir. Pode ser armazenado sem armazenar o item possuir, ao mesmo tempo. Além disso, a mesma referência **Oferta** exemplo, em uma segunda propriedade de **Item**, O único **SuccessfulBid** (Ver figura 3.2). Objetos do tipo valor não pode ser compartilhada.

Para remodelar a **Item** para **Oferta** associação como uma relação pai / filho, a única mudança que precisamos fazer é ao **cascata** atributo:

```
Class <
    name = "Item"
    table = "ITEM">
    ...
<Set
    name = "Licitações"
    inverse = "true"
    cascade = "all-delete-orphan">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
</Set>
</Class>
```

Usamos **cascade = "all-delete-orphan"** Para indicar o seguinte:

Qualquer recém instanciado **Oferta** se torna persistente se o **Oferta** é referenciado por um persistente **Item** (Como foi também o caso com **cascade = "save-update"**). Qualquer persistente **Oferta** deve ser eliminada se for referenciado por um **Item** quando o item é excluído.

Qualquer persistente **Oferta** deve ser eliminada se for removido do **Licitações** cobrança de um persistente **Item**. (NHibernate irá assumir que era apenas referenciado por este item e considerá-lo um órfão.)

Atingimos o seguinte com esse mapeamento: A **Oferta** é removido do banco de dados se ele é removido a coleção de **Ofertas** da **Item** (Ou ele é removido se o **Item** em si é removido).

A cascata de operações para entidades associadas é a implementação de NHibernate transitivo persistência. Olharmos mais de perto este conceito no capítulo 4, secção 4.3, "Usando a persistência transitiva em NHibernate."

Nós cobrimos apenas um subconjunto pequeno das opções disponíveis em associação NHibernate. No entanto, você já tem conhecimento suficiente para ser capaz de construir aplicativos inteiros. As opções restantes são raros ou são variações das associações que descrevemos.

Recomendamos manter a sua associação mapeamentos simples, usando NHibernate para consultas mais complexas tarefas.

3.9 Resumo

Neste capítulo, enfocamos o aspecto estrutural do objeto / incompatibilidade paradigma relacional e discutidos os quatro primeiros problemas ORM genéricos. Discutimos o modelo de programação para classes persistentes e os NHibernate metadados ORM para granulação fina classes, a identidade do objeto, herança e associações.

Agora você entende que as classes persistentes em um modelo de domínio deve ser livre de preocupações transversais como transações e segurança. Mesmo persistência relacionados com preocupações não devem vazar para o modelo de domínio. Nós já não entreter o uso de modelos de programação restritivas, tais como DataSets para o nosso modelo de domínio. Em vez disso, usamos persistência transparente, juntamente com a programação POCO unrestrictive modelo que é realmente um conjunto de melhores práticas para a criação de devidamente encapsulados. NET.

Você também aprendeu sobre as diferenças importantes entre entidades e valor digitado objetos em NHibernate. Entidades têm sua própria identidade e do ciclo de vida, enquanto valor digitado objetos são dependentes de uma entidade e são persistiu com por valor semântica. NHibernate permite que os modelos de grão fino objeto com menos tabelas do que classes persistentes.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Finalmente, nós introduzimos a três conhecidos herança mapeamento de estratégias em NHibernate. Temos também abrangidos associações e mapeamento de coleções e temos implementado e mapeados nosso primeiro pai / filho associação entre classes persistentes, utilizando banco de dados campos de chave estrangeira ea cascata de operações.

Com esse entendimento, você deve achar que você pode experimentar com NHibernate e tratar mais comuns cenários de mapeamento, talvez alguns dos mais espinhosos demais.

Como você se familiarizar com a criação de modelos de domínio e persistindo-as com NHibernate, você pode enfrentar outros desafios arquitetônicos. A seguir, investigar os aspectos dinâmicos do objeto / relacional incompatibilidade, incluindo um estudo muito mais profundo das operações em cascata e introduzimos o ciclo de vida de objetos persistentes.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Trabalhando com objetos persistentes

O que é persistência?	6
1.1.1 Bancos de dados relacionais	6
1.1.2 Noções básicas sobre SQL	7
1.1.3 Usando SQL em Aplicações .NET	7
1.1.4 Persistência em aplicações orientadas objeto	7
1.1.5 Persistência e da arquitetura em camadas	8
Abordagens para 1.2 Persistence in. NET	10
1.2.1 Escolha da camada de persistência	10
DataSet baseado camada de persistência	10
Mão-codificado camada de persistência	10
Camada de persistência usando NHibernate	11
1.2.2 Implementar as entidades	11
Entidades em um DataSet	11
Mão-codificado entidades	11
Entidades e NHibernate	12
1.2.3 entidades Exibindo na interface do usuário	12
DataSet baseado camada de apresentação	12
Camada de apresentação e entidades	12
1.2.4 Implementar operações CRUD	13
Operações de CRUD com DataSets	13
Mão-codificado operações CRUD	13
Operações de CRUD usando NHibernate	14
1.3 Por que precisamos de NHibernate?	14
1.3.1 O paradigma incompatibilidade	14
O problema de granularidade	14
O problema da herança e polimorfismo	15
O problema da identidade	15
Problemas relacionados com associações	15
1.3.2 Unidade de Trabalho e Conversas	15
O padrão de unidade de trabalho	16
Persistência transparente e lazy loading	16
Cache	17
1.3.3 Consultas complexas e ADO.NET Entity Framework	17
Implementação de um mecanismo de consulta	17
17	17
ADO.NET Entity Framework	18
1.4 Mapeamento Objeto / Relacional	19
O que é o ORM?	19
1.4.1 Por que ORM ?	20
Incompatibilidade de modelagem	20
Produtividade e manutenção	20
Desempenho	20
Independência de banco de dados	21
1.5 Resumo	21
2.1 "Olá Mundo" com NHibernate	22
2.1.1 NHibernate Instalando	22
2.1.2 Criar um projeto novo Visual Studio	23

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.1.3 Criando a classe Employee	23
2.1.4 Configurando o banco de dados	24
2.1.3 Criando um empregado e salvar ao banco de dados	25
1.2.4 Carregando um funcionário do banco de dados	26
1.2.5 Criando um arquivo de mapeamento	26
1.2.6 Configurando Sua Aplicação	27
1.2.7 Atualizando um Employee	28
1.2.8 - A execução do Programa de	29
2.2 Entendendo a Arquitetura	30
2.2.1 As interfaces do núcleo	31
Interface de ISession	31
ISessionFactory interface de	31
Interface de configuração	31
Interface de ITransaction	32
Interfaces de IQuery e ICriteria	32
2.2.2 Callback de interfaces	32
2.2.3 Tipos	33
2.2.4 interfaces de Extensão	33
A configuração básica	34
2.3.1 Criando um SessionFactory	34
2.3.2 Configuração do acesso de banco de dados ADO.NET	36
Configuração do NHibernate usando hibernate.cfg.xml	37
NHibernate partida	38
2.4 definições de configuração avançada	39
2.4.1 Utilizando o aplicativo arquivo de configuração	39
2.4.2 Logging	41
2.5 Síntese	42
3.1 A aplicação CaveatEmptor	43
3.1.1 Analisando o domínio do negócio	44
3.1.2 O modelo de domínio CaveatEmptor	44
3.2 Implementação do modelo de domínio	46
3.2.1 Endereçamento vazamento das preocupações	46
46	
3.2.2 persistência transparente e automatizada	46
3.2.3 POCOs Writing	47
3.2.4 Implementando associações POCO	48
3.2.5 Adicionando lógica para propriedades	51
3.3 Definir os metadados de mapeamento	52
3.3.1 Mapeamento usando XML	53
3.3.2 Atributo programação orientada a	54
3.4 propriedade básica e mapeamentos de classe	56
56	
3.4.1 Resumo mapeamento Propriedade	56
3.4.2 Usando propriedades derivadas	57
3.4.3 estratégias de acesso Propriedade	57
3.4.4 Aproveitando o otimizador de reflexão	59
3.4.5 Controle de inserção e atualizações	60
3.4.6 Usando identificadores entre aspas SQL	60
3.4.7 Convenções de nomenclatura	61
3.4.8 esquemas SQL	62
3.4.9 Os nomes das classes Declarando	62
62	
3.4.10 Manipulação de metadados em tempo de execução	63
3.5 a identidade do objeto Entendendo	64
3.5.1 Identidade versus igualdade	64
3.5.2 identidade banco de dados com NHibernate	65
3.5.3 Escolhendo as chaves primárias	67

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.6 modelos refinada objeto	68
3.6.1 Entidade e tipos de valor	69
3.6.2 A utilização de componentes	69
3.7 herança de classe Mapping	73
3.7.1 Tabela por classe concreta	73
3.7.2 Tabela por hierarquia de classe	74
3.7.3 Tabela por subclasse	77
3.7.4 Escolhendo uma estratégia	79
3.8 associações Introdução	80
3.8.1 associações unidirecionais	80
3.8.2 Multiplicidade	80
3.8.3 A associação mais simples possível	81
3.8.4 Fazendo a associação bidirecional	81
3.8.5 A relação pai / filho	83
3.9 Resumo	84
4.1 O ciclo de vida de persistência	89
4.1.1 objetos Transient	90
4.1.2 Objetos persistentes	90
4.1.3 Objetos destacados	91
4.1.4 O escopo da identidade do objeto	92
4.1.5 Fora do âmbito da identidade	93
4.1.6 Implementação Equals () e GetHashCode ()	94
Usando a igualdade identificador do banco de dados	
..... 94	
Comparando-se por valor	95
Usando a igualdade fundamental de negócio	
..... 96	
4.2 gerente de A persistência	97
4.2.1 Fazendo um objeto persistente	97
4.2.2 Atualizando o estado persistente de uma instância desconectada	
98	
4.2.3 Recuperando um objeto persistente	99
4.2.4 Atualizando um objeto persistente	99
4.2.5 Fazendo um objeto transiente	100
4.3 Utilizar persistência transitiva em NHibernate	100
4.3.1 Persistência por acessibilidade	101
4.3.2 persistência com NHibernate cascata	102
4.3.3 categorias leilão Gerenciando	102
4.3.4 A distinção entre casos transitórios e destacado	105
4.4 Recuperando objetos	106
4.4.1 Recuperando objetos pelo identificador	106
4.4.2 Apresentação do Hibernate Query Language	107
4.4.3 Consulta por critérios	108
4.4.4 Consulta por exemplo	108
4.4.5 Buscando estratégias	109
Busca imediata	110
Busca preguiçosa	0,110
Ansioso (outer join) buscar	110
A busca em lote	110
4.4.6 Seleção de uma estratégia de busca em mapeamentos	
110	
Associações único ponto	111
Coleções	112
Definir a buscar profundidade	113
Inicializando associações preguiçoso	114
4.4.7 Ajuste de recuperação de objetos	114
4.5 Resumo	115
Este capítulo aborda	

O ciclo de vida de objetos em um aplicativo NHibernate

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Usando o gerenciador de persistência de
sessão
Persistência transitiva

Estratégia de busca eficiente

Agora você tem uma compreensão de como NHibernate e ORM resolver os aspectos estáticos do objeto / relacional incompatibilidade. Com o que você sabe até agora, é possível resolver o problema de incompatibilidade estrutural, mas um eficiente solução para o problema requer algo mais. Devemos investigar estratégias para acesso a dados em tempo de execução, uma vez eles são cruciais para o desempenho de nossas aplicações. Você precisa aprender a eficiente armazenar e carregar objetos.

Este capítulo aborda os comportamentais aspecto da incompatibilidade objeto / relacional, listados no capítulo 1 como os quatro Problemas de mapeamento O / R descrito no ponto 1.3.1. Consideramos que esses problemas sejam pelo menos tão importante quanto os problemas estruturais discutidas no capítulo 3. Em nossa experiência, muitos desenvolvedores estão apenas conscientes da desfasamento estrutural e raramente prestam atenção aos aspectos mais dinâmica comportamental.

Neste capítulo, discutimos o ciclo de vida de objetos-como um objeto se torna persistente, e como pára sendo considerado persistente e as chamadas de método e de outras acções que desencadeiam estas transições. O NHibernate persistência gerente, o [ISession](#), É responsável por gerenciar o estado do objeto, de modo que você vai aprender como para usar esta API importante.

Recuperar gráficos de objetos de forma eficiente é outra preocupação central, de modo que introduzir as estratégias básicas neste capítulo. NHibernate oferece várias maneiras para especificar consultas que retornam objetos sem perder muito da poder inerente de SQL. Devido a latência da rede causados por acesso remoto ao banco de dados pode ser um importante fator limitante no desempenho geral de aplicativos. NET, você deve aprender como recuperar um gráfico de objetos com um número mínimo de acertos de banco de dados.

Vamos começar discutindo objetos, seu ciclo de vida, e os eventos que desencadeiam uma mudança de estado persistente.

Estes princípios lhe dará o fundo que você precisa quando se trabalha com o gráfico do objeto, para que você saiba quando e como carregar e salvar seus objetos. O material pode ser formal, mas uma sólida compreensão das persistência do ciclo de vida é essencial.

4.1 O ciclo de vida de persistência

Desde NHibernate é um mecanismo de persistência transparente, as aulas são inconscientes de sua própria persistência capacidade. É possível, portanto, escrever a lógica de aplicação que não tem conhecimento sobre se os objetos que ele opera em representam o estado persistente ou estado temporário que só existe na memória. A aplicação não deve necessariamente necessidade de cuidado para que um objeto é persistente ao chamar seus métodos.

No entanto, em qualquer aplicativo com estado persistente, o aplicativo deve interagir com a camada de persistência sempre que ele precisa transmitir estado guardadas na memória para o banco de dados (ou vice-versa). Para fazer isso, você chama Gerente de persistência NHibernate e interfaces de consulta. Ao interagir com o mecanismo de persistência que forma, é necessário para a aplicação de preocupar-se com o Estado e ciclo de vida de um objeto com relação a persistência. Vamos nos referir a isso como o persistência do ciclo de vida.

Implementações ORM diferentes usam terminologia diferente e definir diferentes estados e transições de estado para o ciclo de vida de persistência. Além disso, os estados de objeto usado internamente podem ser diferentes daquelas expostas para o aplicativo cliente. NHibernate define apenas três estados, escondendo a complexidade de sua interna implementação do código do cliente. Nesta seção, vamos explicar esses três estados: transitória, persistente, e destacados.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Vamos olhar para esses estados e suas transições em um gráfico de estado, mostrado na figura 4.1. Você também pode ver o método chamadas para o gerenciador de persistência que as transições gatilho. Discutimos este gráfico nesta seção, se referem a ele mais tarde sempre que você precisar uma visão geral.

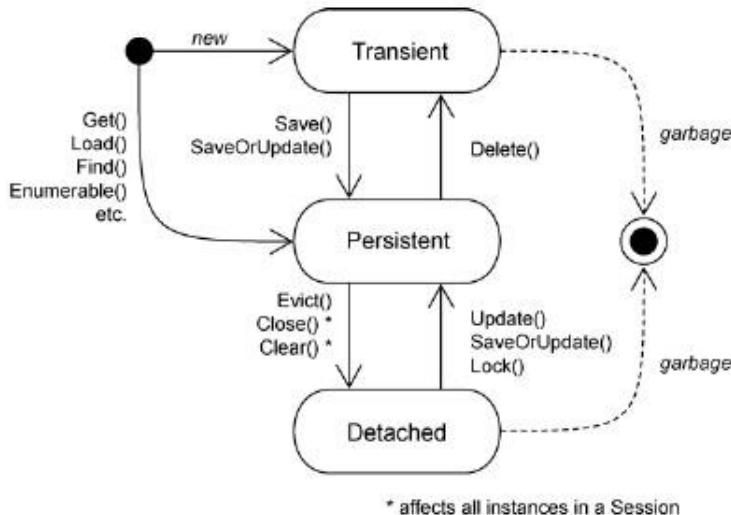


Figura 4.1 Estados de um objeto e as transições em uma aplicação NHibernate

Em seu ciclo de vida, um objeto pode fazer a transição de um objeto transitório em um objeto persistente de um objeto destacado. Vamos dar uma olhada em cada um desses estados.

4.1.1 objetos transitórios

Em NHibernate, objetos instanciados usando o `novo` operador não são imediatamente persistentes. Seu estado é transitório, o que significa que não estão associados com qualquer linha da tabela do banco de dados, e assim eles são como qualquer outro objeto em um aplicativo .NET. Mais especificamente, seu estado é perdido, logo que eles estão dereferenced (não mais referenciada por qualquer outro objeto) pela aplicação. Esses objetos têm uma vida útil que efetivamente termina no que tempo, e eles se tornam inacessíveis e disponíveis para coleta de lixo.

NHibernate considera todas as instâncias transitórias a não-transacional, uma modificação para o estado de um instância transitória não é feita no contexto de qualquer transação. Este NHibernate significa não fornece nenhuma funcionalidade de reversão para objetos transitórios. Na verdade, NHibernate não reverte qualquer alteração do objeto, como você vai ver mais tarde.

Objetos que são referenciados apenas por outras instâncias transitórias são, por padrão, também transitórias. Uma instância pode transição de transitória para o estado persistente de duas maneiras. Uma delas é a `Save()` -lo usando o gerenciador de persistência, outra forma é criar uma referência a ele de uma instância já persistente.

4.1.2 Os objetos persistentes

Uma instância persistente é qualquer instância com uma identidade banco de dados, conforme definido no capítulo 3, seção 3.5, "Identidade de objeto Entendimento." Isto significa que uma instância persistente tem um valor de chave primária definida como sua base de dados identificador.

Instâncias persistentes podem ser objetos instanciados pela aplicação e em seguida, fez persistentes chamando o `Save()` método do gerenciador de persistência (o NHibernate `ISession`, Discutido em mais detalhes posteriormente neste capítulo). Instâncias persistentes são, então, associado com o gerenciador de persistência. Eles podem até ser objetos que tornou-se persistente quando uma referência foi criado a partir de outro objeto persistente já associado a um Por favor, postar comentários ou correções para o forum on-line em Autor <http://www.manning-sandbox.com/forum.jspa?forumID=295>

gerenciador de persistência. Alternativamente, uma instância persistente pode ser uma instância recuperados do banco de dados

execução de uma consulta, por um identificador de pesquisa, ou navegando o objeto gráfico a partir de outro persistente exemplo. Em outras palavras, instâncias persistentes são sempre associados a uma `ISession` e são transacional.

Instâncias persistentes participar de transações, seu estado é sincronizado com o banco de dados no final de a transação. Quando uma transação for confirmada, o estado retido na memória é propagada para o banco de dados pelo execução de SQL `INSERIR`, `ATUALIZAÇÃO` E `APAGAR` declarações. Este procedimento também pode ocorrer em outros momentos.

Por exemplo, NHibernate pode sincronizar com o banco de dados antes da execução de uma consulta. Isso garante que consultas estarão cientes das mudanças feitas anteriormente durante a transação.

Chamamos uma instância persistente novo se tiver sido atribuído um valor de chave primária, mas ainda não foi inserido

no banco de dados. A nova instância persistente continuará a ser "novo" até que a sincronização ocorre.

Claro, NHibernate não tem que atualizar a linha do banco de dados de cada objeto persistente na memória no final da transação. Salvar objetos que não foram alterados seria demorado e desnecessário. ORM software deve ter uma estratégia para a detecção de objetos persistentes, que foram modificados pela aplicação na transação. Chamamos a isto checagem suja automática (Um objeto com modificações que não tenham ainda sido propagadas para o banco de dados é considerada sujo). Novamente, este estado não é visível para a aplicação. Nós

chamar esse recurso transparente no nível da transação write-behind , O que significa que NHibernate propaga as mudanças de estado

ao banco de dados o mais tarde possível, mas esconde esse detalhe da aplicação.

NHibernate pode detectar exatamente quais atributos foram modificados, por isso é possível incluir apenas os colunas que precisam de atualização no SQL `ATUALIZAÇÃO` declaração. Isso pode trazer ganhos de desempenho, especialmente

com certas bases de dados. No entanto, geralmente não é uma diferença significativa, e, em teoria, pode prejudicar desempenho em alguns ambientes. Então, por padrão, NHibernate inclui todas as colunas no SQL `ATUALIZAÇÃO` declaração. Assim, NHibernate pode gerar e armazenar em cache esta SQL básico, uma vez na inicialização, ao invés de on-the-fly

cada vez que um objeto é salvo. Se você só quiser atualizar colunas modificadas, você pode ativar SQL dinâmica geração através da criação `dynamic-update = "true"` em um mapeamento de classe. Note que esta característica é extremamente

difícil e demorado para implementar em uma camada de persistência handcoded. Falamos sobre NHibernate semântica da transação eo processo de sincronização, ou rubor, mais detalhadamente no próximo capítulo.

Finalmente, uma instância persistente pode ser feita através de um transiente `Delete()` chamada para o gerenciador de persistência API, resultando em supressão da linha correspondente da tabela de banco de dados.

4.1.3 Objetos destacados

Quando uma transação é concluída e os dados são gravados no banco de dados, as instâncias persistente associada com o

persistência gerente ainda existem na memória. Se a transação foi bem sucedida, o estado desses casos serão foram sincronizadas com o banco de dados. Em implementações com ORM processo de escopo de identidade (Veja a seguintes seções), as instâncias manter a sua associação ao gerenciador de persistência e ainda são considerados persistente.

No caso do NHibernate, no entanto, estes casos perdem a sua associação com o gerenciador de persistência quando você `Close()` o `ISession`. Porque eles não estão mais associados a nossa gerente de persistência, nos referimos a esses objetos como destacados. Instâncias destacadas pode não ser mais garantido para ser sincronizado com estado do banco de dados, pois eles não estão mais sob a gestão do NHibernate. No entanto, eles ainda contêm persistente.

de dados. É possível, e comum, para a aplicação de manter uma referência e atualizar um objeto destacado fora de uma transação e, portanto, sem acompanhamento NHibernate as alterações. Felizmente, NHibernate permite que você use

Nestes casos, em uma nova transação por Re-associando-os com um novo gerenciador de persistência. Depois de reassociação,

eles são considerados persistentes novamente. Esta característica tem um profundo impacto sobre como os aplicativos de várias camadas podem ser

projeto. A habilidade de retornar objetos de uma transação para a camada de apresentação e depois reutilizá-los em um

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

nova transação é um dos principais pontos de venda NHibernate. Discutimos este uso no próximo capítulo como um técnica de implementação de longa duração operações de aplicação. Nós também mostrar-lhe como evitar o DTO (Anti-) padrão usando objetos destacados no capítulo 10, seção 10.3.1.

NHibernate também fornece uma maneira explícita de detaching instâncias: o `Evict()` método da `ISession`. No entanto, este método é normalmente utilizado apenas para o gerenciamento de cache (uma consideração de desempenho).

É não normal para realizar o desapego explicitamente. Em vez disso, todos os objetos recuperados em uma transação se destaquem quando o `ISession` é fechada ou quando estão em série (se eles são passados de forma remota, por exemplo). Assim, NHibernate não precisa fornecer funcionalidade para o descolamento de controle da subgrafos. Em vez disso, o aplicativo pode controlar a profundidade do subgrafo buscada (as instâncias que estão atualmente carregados na memória)

usando a linguagem de consulta ou navegação gráfico explícito. Então, quando o `ISession` está fechado, todo este subgrafo (todos os objetos associados com um gerenciador de persistência) torna-se destacado.

Vamos olhar para os diferentes estados de novo, mas, desta vez, considerar a âmbito da identidade de objeto.

4.1.4 O escopo da identidade de objeto

Como desenvolvedores de aplicativos, identificamos um objeto usando .NET identidade de objeto (`A, b ==`). Assim, se um objeto muda

Estado, é a sua identidade. .NET garantido para ser o mesmo no novo estado? Em uma aplicação em camadas, que podem não ser o caso.

A fim de explorar este tema, é importante compreender a relação entre os dois. .NET identidade, `Object.ReferenceEquals(a, b)` é identidade de banco de dados, `a.Id == b.Id`. Às vezes, ambos são equivalentes; às vezes eles não são. Referimo-nos às condições sob as quais. .NET identidade é equivalente à identidade do banco de dados como o âmbito da identidade de objeto.

Desse escopo, existem três opções comuns:

A camada de persistência primitivo com qualquer possibilidade de identidade não garante que se uma linha é acessado duas vezes, o mesma instância do objeto. .NET será retornado para o aplicativo. Isso se torna problemático se a aplicação modifica duas instâncias diferentes que ambos representam a mesma linha em uma única transação (como você decide qual Estado deveria ser propagadas para o banco de dados?).

A camada de persistência usando garantias de transação com escopo de identidade que, no contexto de uma única transação, há apenas uma instância do objeto que representa uma linha de banco de dados particular. Isso evita o problema anterior e também permite a alguns cache a ser feito no nível da transação.

Processo de escopo de identidade vai um passo além e garante que há apenas uma instância do objeto que representa a linha em todo o processo (CLR).

Para um típico web ou de aplicações empresariais, de escopo de transação de identidade é o preferido. Processo de escopo de identidade

oferece algumas vantagens potenciais em termos de utilização de cache eo modelo de programação para a reutilização de

casos em várias transações, no entanto, em uma aplicação pervasively vários segmentos, o custo de sempre sincronizar o acesso compartilhado a objetos persistentes no mapa identidade global é um preço muito alto a pagar. É mais simples e mais escalável, para que cada obra discussão com um conjunto distinto de instâncias persistentes em cada

escopo de transação.

Falando vagamente, diríamos que NHibernate implementa transação com escopo de identidade. Na verdade, a NHibernate âmbito identidade é o `ISession` exemplo, objetos de modo idêntico é garantida se o mesmo persistência manager (o `ISession`) É usada para várias operações. Mas uma `ISession` não é o mesmo que um (Database) transação é um elemento muito mais flexível. Vamos explorar as diferenças e as consequências deste conceito no próximo capítulo. Vamos nos concentrar sobre o ciclo de vida de persistência e identidade âmbito novamente.

Se você solicitar dois objetos utilizando o mesmo valor identificador banco de dados na mesma `ISession`, O resultado será

ser duas referências para o objeto na memória mesmo. O exemplo de código a seguir demonstra esse comportamento, com

`ISession session1 = sessionFactory.openSession();`

`Vários Load() operações em dois ISessions:`

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

ITransaction session1.BeginTransaction TX1 = ();
Categoria / Load / identificador com valor "1234"
objeto a = session1.Load (typeof (Categoria), 1234);
objeto b = session1.Load (typeof (Categoria), 1234);
if (Object.ReferenceEquals (a, b)) {
    System.Console.WriteLine ("a e b são idênticos.");
}
tx1.Commit ();
session1.Close ();
ISession session2 sessionFactory.openSession = ();
ITransaction tx2 = session2.BeginTransaction ();
// Vamos usar a versão genérica do Load ()
B2 categoria = session2.Load <category> (1234);
if (! Object.ReferenceEquals (a b2)) {
    System.Console.WriteLine ("A e B2 não são idênticos.");
}
tx2.Commit ();
session2.Close ();

```

Referências a objetos **ume** **b**não somente para ter a identidade mesmo banco de dados, eles também têm a identidade mesmos NET desde que foram carregados na mesma **ISession**. Uma vez fora desse limite, no entanto, não NHibernate garantiu. NET identidade, assim **ume** **b2** não são idênticos e que a mensagem é impressa no console. Naturalmente, uma teste de identidade de banco de dados **a.Id==b2.Id**-Ainda return true.

Para complicar ainda mais a nossa discussão de escopos de identidade, precisamos considerar como a camada de persistência lida com uma referência a um objeto fora de seu escopo de identidade. Por exemplo, para uma camada de persistência com transação com escopo de identidade, como NHibernate, é uma referência a um objeto separado (isto é, uma instância persistiu ou carregadas em uma sessão, anterior concluído) tolerado?

4.1.5 Fora do âmbito da identidade

Se uma referência de objeto sai do escopo de identidade garantida, nós a chamamos de referência a um objeto destacado . Por que

É este conceito é útil?

Em aplicações Windows, você normalmente não mantém uma transação de banco de dados através de uma interação do usuário.

Usuários demorar muito tempo para pensar sobre as modificações, por isso por motivos de escalabilidade, você deve manter banco de dados

operações de curto e recursos de banco de dados de liberação o mais rapidamente possível. Neste ambiente, é útil poder a reutilização de uma referência a uma instância separada. Por exemplo, você pode querer enviar um objeto recuperado em uma unidade

de trabalho para a camada de apresentação e mais tarde reutilizá-la em uma segunda unidade de trabalho algum tempo depois, depois de ter sido

modificada pelo usuário. Para aplicações ASP.NET isso não se aplica, porque você não deve manter os negócios objetos na memória depois que a página tenha sido prestado - em vez disso você recarregá-las a cada solicitação, portanto, não exigindo reatamento.

Para quando você precisa fazer para recolocar os objetos, você geralmente não deseja recolocar o objeto gráfico inteiro no segundo unidade de trabalho. Para o desempenho (e outras) razões, é importante que reassociação de desanexado casos ser seletivo. NHibernate suporta reassociação seletiva de instâncias desconectadas. Isso significa que o aplicação pode recolocar uma forma eficiente subgrafo de um gráfico de objetos soltos com a corrente ("segundo") NHibernate **ISession**. Uma vez que um objeto destacado foi recolocado em um novo gerente de persistência NHibernate, ela pode ser considerada uma instância persistente novamente, e seu estado será sincronizado com o banco de dados no final

da transação. Isto é devido à verificação automática do NHibernate sujo de instâncias persistentes.

Religação pode resultar na criação de novas linhas no banco de dados quando uma referência é criada a partir de um destacado exemplo para uma nova instância transitória. Por exemplo, um novo **Oferta** poderia ter sido adicionado a uma destacada

Ronfou para a instância carregar de apresentação o NHibernate pode detectar que a **Oferta** é novo e deve ser inserido <http://www.manning-sandbox.com/forum.jspa?forumID=295>

banco de dados. Para que isso funcione, NHibernate deve ser capaz de distinguir entre uma instância de "novo" transitórios e

uma instância "velho" destacados. Casos transitórios (como o [Oferta](#)) Talvez precise ser salvo; instâncias destacadas (Como o [Item](#)) Pode precisar ser recolocado (e mais tarde atualizado no banco de dados). Existem várias maneiras de distinguir entre instâncias transitória e individual, mas a abordagem é mais agradável olhar para o valor do de propriedade de identificação. NHibernate pode examinar o identificador de um objeto transiente ou destacado no reatamento e tratar o objeto (e o gráfico associado de objetos) de forma apropriada. Discutimos esta importante questão ainda em seção 5.3.4, "A distinção entre casos transitórios e isenção".

Se você quer aproveitar o suporte do NHibernate para reassociação das instâncias destacadas em suas próprias aplicativos, você precisa estar ciente do escopo do NHibernate identidade ao projetar sua aplicação, isto é, o [ISession](#) escopo que garante instâncias idênticas. Assim que você deixar que o escopo e têm destacado casos, um outro conceito interessante entra em jogo.

Precisamos discutir a relação entre os dois. NET igualdade e identidade de banco de dados. Para uma repescagem na igualdade, veja o capítulo 3, seção 3.5.1 - "Identidade versus igualdade". A igualdade é um conceito de identidade que nós, da classe desenvolvedores, podemos controlar. Às vezes temos que usá-lo para as classes que têm destacado as instâncias. .NET é a igualdade definidos pela implementação do [Equals \(\)](#) e [GetHashCode \(\)](#) métodos nas classes persistentes da o modelo de domínio.

4.1.6 Implementação Equals () e GetHashCode ()

O [Equals \(\)](#) método é chamado pelo código do aplicativo ou, mais importante, pelo coleções. NET. Um [ISET](#) coleta (no [List<Collection>](#) biblioteca), por exemplo, as chamadas [Equals \(\)](#) em cada objeto que você colocar na [ISET](#), para determinar (e evitar) duplicar elementos.

Primeiro, vamos considerar a implementação de padrão de [Equals \(\)](#), Definida por [System.Object](#), Que usa um comparação por. NET identidade. NHibernate garante que não é uma instância única para cada linha da banco de dados dentro de um [ISession](#). Portanto, a identidade do padrão [Equals \(\)](#) é apropriado se você nunca mix casos, isto é, se você nunca colocou instâncias destacadas das sessões diferentes para o mesmo [ISET](#). (Na verdade, a questão que estamos explorando é também visível se casos destacados são da mesma sessão, mas têm sido serializado e desserializado em escopos diferentes.) Assim que você tiver instâncias de várias sessões, No entanto, torna-se possível ter um [ISET](#) contendo dois [Items](#) que cada representar a mesma linha do tabela de banco de dados, mas não tem a identidade. mesmos NET. Isso quase sempre ser semanticamente errado. No entanto, é possível construir uma aplicação complexa com a identidade (padrão) é igual, enquanto você se exercita disciplina ao lidar com objetos destacados por diferentes sessões (e ficar de olho na serialização e desserialização). Uma coisa legal sobre esta abordagem é que você não tem que escrever código extra para implementar sua própria noção de igualdade.

No entanto, se esse conceito de igualdade não é o que você quer, você tem que substituir [Equals \(\)](#) na sua classes persistentes. Tenha em mente que quando você substituir [Equals \(\)](#), Você sempre precisa também substituir [GetHashCode \(\)](#) assim que os dois métodos são consistente (Se dois objetos são iguais, eles devem ter o mesmo hash código). Vejamos algumas das maneiras que você pode substituir [Equals \(\)](#) e [GetHashCode \(\)](#) na persistente classes.

Usando a igualdade identificador do banco de dados

Uma abordagem inteligente é implementar [Equals \(\)](#) comparar apenas a propriedade identificador de banco de dados (geralmente um chave primária substituto) valor:

Usuário public class {

```
    ...
    público substituir bool Equals (outro objeto) {
        se (Object.ReferenceEquals (este, outros)) return true;
        se (Id == null) return false;
        se (! (Que é User)) return false;
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        Usuário que = (User) outras;
        retorno this.Id == that.Id;
    }
    public override int GetHashCode () {
        Id retorno == null?
            base.GetHashCode (this):
            Id.GetHashCode ();
    }
}

```

Observe como esta `Equals ()` método cai de volta para a identidade transitória NET para instâncias (se `id == null`) Que não têm um valor de identificador de banco de dados atribuído ainda. Isso é razoável, já que eles não podem ter o mesmo identidade persistente como outro exemplo.

Infelizmente, esta solução tem um problema enorme: NHibernate não atribuir valores de identificador até que um entidade é salvo. Então, se o objeto é adicionado a um `ISET` antes de ser salvo, seu código de hash mudanças enquanto é contidos pela `ISET`, Ao contrário do contrato definidas por esta coleção. Em particular, este problema torna cascata save (discutido mais tarde neste capítulo) inútil para jogos. Desaconselhamos fortemente esta solução (banco de dados igualdade de identificador).

Comparando-se por valor

A melhor maneira é incluir todas as propriedades persistentes da classe persistente, para além de qualquer identificador do banco de dados propriedade, na `Equals ()` comparação. Isto é como a maioria das pessoas percebem o significado de `Equals ()`; Chamamos ele por valor igualdade.

Quando dizemos "todas as propriedades," não queremos dizer para incluir acervos. Estado coleção é associado a um tabela diferente, por isso parece errado para incluí-lo. Mais importante, você não quer forçar o gráfico inteiro objeto para ser recuperado apenas para realizar `Equals ()`. No caso de `Usuário`, Isso significa que você não deve incluir o itens coleção (os itens vendidos por este utilizador) na comparação. Então, esta é a aplicação que você poderia usar:

```
Usuário public class {
```

```

    ...
    public override bool Equals (outro objeto) {
        if (Object.ReferenceEquals (este, outros)) return true;
        return false se ((outra é do usuário)!);
        Usuário que = (User) outras;
        if (! this.Username that.Username ==)
            return false;
        if (! this.senha == that.Password)
            return false;
        return true;
    }
    public override int GetHashCode () {
        int resultado = 14;
        resultado = 29 * + resultado Username.GetHashCode ();
        resultado = 29 * + resultado Password.GetHashCode ();
        resultado de retorno;
    }
}
```

No entanto, há novamente dois problemas com essa abordagem:

Casos de sessões diferentes não são mais iguais se for modificado (por exemplo, se o usuário mudar sua senha).

Casos com a identidade do banco de dados diferentes (instâncias que representam diferentes linhas da tabela do banco de dados)

poderiam ser considerados iguais, a menos que haja alguma combinação de propriedades que são garantidos para ser únicos (as colunas de banco de dados tem uma restrição de unicidade). No caso de `Usuário`, Há uma única propriedade comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Para chegar à solução, recomendamos, você precisa entender a noção de uma chave de negócio.

Usando a igualdade de negócios

A chave de negócios é uma propriedade, ou alguma combinação de propriedades, que é único para cada instância com o mesmo identidade banco de dados. Essencialmente, é a chave natural que você usaria se você não estivesse usando uma chave substituta. Ao contrário de um chave primária natural, não é um requisito absoluto que nunca a chave de negócios mudança, desde que as mudanças raramente, isso é o suficiente.

Nós argumentamos que cada entidade deve ter uma chave de negócio, mesmo que inclua todas as propriedades da classe (esta seria adequado para algumas classes imutáveis). A chave do negócio é o que o usuário pensa em como exclusivamente identificar um registro específico, enquanto a chave substituta é o que a aplicação e uso de banco de dados.

Igualdade fundamental de negócios significa que o `Equals()` método compara apenas as propriedades que formam o chave de negócio. Esta é uma solução perfeita, que evita todos os problemas descritos anteriormente. A única desvantagem é que requer pensamento extra para identificar a chave de negócios correto em primeiro lugar. Mas este esforço é necessário de qualquer forma, é importante identificar todas as chaves originais, se você quiser que o seu banco de dados para ajudar a garantir a integridade dos dados via verificação de restrição.

Para o `Usuário classe, nome de usuário` é uma chave grande negócio candidato. Nunca é nulo, ele é único, e mudanças raramente (ou nunca):

```
Usuário public class {
    ...
    public override bool Equals (outro objeto) {
        if (Object.ReferenceEquals (este, outros)) return true;
        return false se ((outra é do usuário)!);
        Usuário que = (User) outras;
        retorno this.Username == that.Username);
    }
    public override int GetHashCode () {
        retorno Username.GetHashCode ();
    }
}
```

Para algumas outras classes, a chave de negócios pode ser mais complexo, composto por uma combinação de propriedades.

Por exemplo, o candidato chaves de negócios para o `Oferta classe` são o item de ID juntamente com o valor do lance, ou o item de ID juntamente com a data ea hora da oferta. Uma chave de um bom negócio para a `BillingDetails` abstrato classe é o `número` juntamente com o tipo (subclasse) de detalhes de faturamento. Repare que quase nunca correto substituir `Equals()` em uma subclasse e incluir outra propriedade na comparação. É complicado para satisfazer as requisitos que a igualdade seja simétrica e transitiva, neste caso, e, mais importante, a chave do negócio não corresponderia a qualquer chave candidata bem definidos naturais no banco de dados (propriedades subclasse pode ser mapeada para uma tabela diferente).

Você deve ter notado que o `Equals()` e `GetHashCode()` métodos sempre acessar as propriedades do outro objeto através dos métodos getter. Isto é importante, uma vez que a instância do objeto passado como `outros` pode ser um objeto de proxy não, a ocorrência real que mantém o estado persistente. Este é um ponto onde NHibernate não é completamente transparente, mas é uma boa prática para usar as propriedades em vez de acesso directo a variável de instância de qualquer maneira.

Por fim, tome cuidado ao modificar o valor das propriedades-chave de negócio, não alterar o valor, enquanto o objeto de domínio está em um conjunto.

Até agora falamos sobre como o assunto de forma ampla como o gerenciador de persistência se comporta quando se trabalha com instâncias que são transitórios, persistente ou destacado. Nós também discutimos questões de alcance, e os importâncias da igualdade e identidade. Agora é hora de dar uma olhada no gerenciador de persistência e explorar o NHibernate `ISession` API em maior detalhe. Voltaremos a objetos independentes com mais detalhes no próximo capítulo.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

4.2 O gerenciador de persistência

Qualquer ferramenta de persistência transparente como NHibernate irá incluir alguma forma de persistência gerente API, que

normalmente fornece serviços para
Operações básicas CRUD

Execução da consulta

Controlo das operações

Gerenciamento do cache de nível de transação

O gerenciador de persistência pode ser exposta por várias interfaces diferentes (no caso do NHibernate, **ISession**, **IQuery**, **ICriteria** E **ITransaction**). Nos bastidores, as implementações destes interfaces são acoplados firmemente.

A interface central entre a aplicação eo NHibernate é **Sessão**. É o ponto de partida para todas as apenas as operações listadas. Para a maioria do resto deste livro, vamos nos referir ao persistência gerente eo sessão indistintamente, o que é consistente com o uso da comunidade NHibernate.

Então, como você começar a usar a sessão? No início de uma unidade de trabalho, uma thread obter uma instância de **ISession** a partir da aplicação de **ISessionFactory**. O aplicativo pode ter múltiplas **ISessionFactories** se ele acessa datasources múltiplas. Mas você nunca deve criar um novo **ISessionFactory** apenas para um determinado serviço de solicitação de criação de um **ISessionFactory** é extremamente caro. Por outro lado, **ISession** criação é extremamente baixo custo; o **ISession** nem sequer obter um ADO.NET **IDbConnection** até que uma conexão é necessária.

Depois de abrir uma nova sessão, você usá-lo para carregar e salvar objetos.

4.2.1 Fazendo um objeto persistente

A primeira coisa que você quer fazer com um **ISession** é fazer um novo objeto persistente transitória. Para fazer isso, você usa

O **Save()** método:
usuário Usuário = new ();
usuário.Name.Firstname = "João";
usuário.Name.Lastname = "da Silva";
usando (sessão ISession sessionFactory.openSession = ()
usando (session.BeginTransaction () {
 Session.save (usuário);
 session.Transaction.Commit ();
})

Primeiro, vamos instanciar um novo objeto transiente **usuário** como de costume. Claro, também podemos instanciá-lo após a abertura

um **ISession**, Não são relacionados ainda. Abrimos um novo **ISession** usando o **ISessionFactory** referidas por **sessionFactory**, E então começamos uma nova transação.

Uma chamada para **Save()** faz com que a instância transiente de **Usuário** persistente. Agora é associado com o atual **ISession**. No entanto, não SQL **INSERIR** ainda não foi executada. O NHibernate **ISession** nunca executa qualquer declaração SQL, até absolutamente necessário.

As alterações feitas em objetos persistentes devem ser sincronizados com o banco de dados em algum ponto. Este acontece quando **Commit()** o NHibernate **ITransaction**. Neste caso, NHibernate obtém um ADO.NET conexão (e operação) e emite um único SQL **INSERIR** declaração. Por fim, o **ISession** é fechado e a conexão ADO.NET é liberado.

Note-se que é melhor (mas não obrigatório) para inicializar o pleno **Usuário** exemplo, antes de associá-la com o **ISession**. O SQL **INSERIR** declaração contém os valores que foram mantidos pelo objeto no momento em que

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

`Save()` foi chamado. Você pode, naturalmente, modificar o objeto após a chamada `Save()`, E suas alterações serão propagadas para o banco de dados como um SQL **ATUALIZAÇÃO**.

Tudo entre `session.BeginTransaction()` e `Transaction.Commit()` ocorre em um transação. Não discutimos as operações em detalhe ainda, vamos deixar esse assunto para o próximo capítulo. Mas tenha em mente que todas as operações de banco de dados em um escopo de transação são atômico - Eles completamente bem-sucedidas ou completamente falhar. Se um dos **ATUALIZAÇÃO** ou **INSERIR** declarações feitas em `Transaction.Commit()` falhar, todos os alterações feitas em objetos persistentes nessa transação será revertida ao nível do banco de dados. No entanto, NHibernate faz não roll back in-memory alterações em objetos persistentes, seu estado permanece exatamente como você deixou -lo. Isso é razoável, já que uma falha de uma operação de banco de dados é normalmente não-recuperáveis e você tem que descartar a não `ISession` imediatamente.

4.2.2 Atualizando o estado persistente de uma instância desconectada

Modificar o **usuário** após a sessão é fechada não terá nenhum efeito sobre a sua representação persistente no banco de dados. Quando a sessão é fechada, **usuário** torna-se um destacado exemplo. No entanto, pode ser reassociated com um novo **Sessão** algum tempo depois chamando `Update()` ou `Lock()`.

Vamos primeiro olhar no `Update()` método. Utilização `Update()` vai forçar uma atualização para o estado persistente de o objeto no banco de dados; um SQL **ATUALIZAÇÃO** está programada e será mais tarde cometido. Aqui está um exemplo de manipulação de objetos destacam-se:

```
User.password = "segredo";
using (ISession sessionTwo sessionFactory.openSession = ())
    usando (sessionTwo.BeginTransaction ()) {
        sessionTwo.Update (usuário);
        user.username = "jonny";
        sessionTwo.Transaction.Commit ();
    }
```

Não importa se o objeto é modificado antes ou depois é passado para `Update()`. O importante é que a chamada para `Update()` é usado para reassociate a instância separada para o novo `ISession`, Ea corrente transação. NHibernate irá tratar o objeto como sujo e, portanto, agendar a **SQL UPDATE** regardless de se o objeto foi atualizado ou não. Isso faz com que `Update()` uma maneira "segura" de Re-associando objetos com um **Sessão** porque você sabe que as mudanças serão propagadas ao banco de dados. Na verdade, há uma exceção a esta, que é quando você tiver ativado `select-before-update` na classe persistente. Se você fizer isso, vai NHibernate determinar se o objeto estiver sujo, em vez de assumi-la. Ele faz isso execução de um **SELEÇÃO** declaração e comparando o estado atual do objeto para o estado do banco de dados atual.

Agora vamos olhar um a `Lock()` método. Uma chamada para `Lock()` associa o objeto com o `ISession` sem forçando NHibernate para tratar o objeto como sujo. Considere este exemplo:

```
using (ISession sessionTwo sessionFactory.openSession = ()) {
```

```
    usando (sessionTwo.BeginTransaction ()) {
        sessionTwo.Lock (usuário, LockMode.NONE);
        User.password = "segredo";
        user.LoginName = "jonny";
        sessionTwo.Transaction.Commit ();
    }
```

Ao usar `Lock()`, Ele não importa se as mudanças são feitas antes ou depois do objeto é associado com o sessão. As alterações feitas antes a chamada para `Lock()` não são propagadas para o banco de dados porque não tem NHibernate testemunhou essas mudanças, você só use `Lock()` se você tem certeza que o exemplo destacado não foi modificado antemão.

No código acima, especificamos `LockMode.NONE`, Que diz não NHibernate para executar uma verificação de versão ou obter quaisquer bloqueios de nível de banco quando Re-associando o objeto com o `ISession`. Se especificado `LockMode.READ` ou `LockMode.Upgrade`, NHibernate deveria executar uma **SELEÇÃO** declaração, a fim de

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

executar uma verificação de versão (e para definir um bloqueio de atualização). Tomamos um olhar detalhado em modos de bloqueio na NHibernate próximo capítulo. Tendo discutido como os objetos são tratados quando reassocie-los com um [Sessão](#), Vamos agora olhar para o que acontece quando recuperar objetos.

4.2.3 Recuperando um objeto persistente

O [ISession](#) também é usado para consultar o banco de dados existentes e recuperar objetos persistentes. NHibernate é especialmente poderosas nesta área, como você verá mais adiante neste capítulo e no capítulo 7. No entanto, métodos especiais

são fornecidos no [ISession](#) API para o tipo mais simples de consulta: recuperação por identificador. Um desses métodos é [Get \(\)](#), Demonstrado aqui:

```
int userID = 1234;
usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
    Usuário user = (User) session.get (typeof (User), userID);
    session.Transaction.Commit ();
}
```

O objeto recuperado [usuário](#) podem agora ser passados para a camada de apresentação para uso fora da transação como um
instância desconectada (após a sessão foi fechada). Se nenhuma linha com o valor do identificador dado existe no banco de dados, o [Get \(\)](#) retorna [nulo](#).

Desde NHibernate 1.2, podemos usar NET 2.0 genéricos.:

```
User = usuário session.get <User> (userID);
```

4.2.4 Atualizando um objeto persistente

Qualquer objeto persistente retornado por [Get \(\)](#) ou qualquer outro tipo de consulta já está associado com a corrente [ISession](#) e contexto de transação. Ele pode ser modificado, e seu estado será sincronizado com o banco de dados. Este mecanismo é chamado checagem suja automática, o que significa NHibernate irá monitorar e salvar as alterações você faz a um objeto dentro de uma sessão:

```
int userID = 1234;
usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
    Usuário user = (User) session.get (typeof (User), userID);
    User.password = "segredo";
    session.Transaction.Commit ();
}
```

Primeiro vamos recuperar o objeto do banco de dados com o identificador especificado. Nós modificar o objeto, e estes modificações são propagadas para o banco de dados quando [Transaction.Commit \(\)](#) é chamado. É claro que, tão logo fechamos a [ISession](#), A instância é considerado imparcial. Atualizações em lote também são possíveis desde NHibernate foi ajustada para usar ADO.NET 2.0 batching característica interna. A ativação desse recurso fará NHibernate realizar atualizações em massa; essas atualizações, portanto, tornam-se muito mais rápido. Tudo que você precisa fazer é definir o lote tamanho como uma propriedade NHibernate:

```
<property name="hibernate.adonet.batch_size"> 16 </ property>
```

Por padrão, o tamanho do lote é de 0 o que significa que esse recurso está desativado.

Atualmente esse recurso só funciona em .NET 2.0 quando se utiliza um banco de dados SQL Server. E porque ele usa .NET reflexão, ele pode não funcionar em alguns ambientes restritos.

Finalmente, ao usar esse recurso, ADO.NET 2.0 não retornar o número de linhas afetadas por cada instrução no lote que significa que NHibernate não pode executar a verificação de simultaneidade otimista corretamente. Por exemplo, se uma instrução afeta duas linhas e nenhuma outra linha (em vez de afetar uma em cada), NHibernate só vai saber que duas linhas foram afetadas, e concluir que tudo correu bem.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

4.2.5 Fazendo um objeto transitório

Em muitos casos de uso, que precisam da nossa persistente (ou independente) para se tornar objetos transitórios, novamente, o que significa que vai

já não têm dados correspondentes no database. As discutido no início deste capítulo, persistente objetos são aqueles que estão na sessão e ter dados correspondentes na base de dados. Tornando-os transitórios irá remover seu estado persistente do banco de dados. Isso é facilmente conseguido usando a `Delete()` método:

```
int userID = 1234;
```

```
usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
    User = usuário session.get <User> (userID);
    session.Delete (usuário);
    session.Transaction.Commit ();
})
```

O SQL `APAGAR` será executada somente quando o `ISESsion` é sincronizado com o banco de dados no final do transação.

Após o `ISession` está fechada, o `usuário` objeto é considerado uma instância ordinária transitória. O transitório instância será destruída pelo coletor de lixo, se ele não é mais referenciado por qualquer outro objeto, portanto, tanto a instância na memória e na fila do banco de dados persistente terá sido removido.

Da mesma forma, objetos separados também podem ser feitas transitória (Objetos destacados sendo aqueles que têm estado correspondente no banco de dados, mas que não estão no `ISession`). Você não tem para recolocar uma destacada exemplo, para a sessão com `Update()` ou `Lock()`. Em vez disso você pode diretamente excluir um exemplo destacado como seguinte forma:

```
usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
    session.Delete (usuário);
    session.Transaction.Commit ();
})
```

Neste caso, a chamada para `Delete()` faz duas coisas: Ele associa o objeto com o `ISession` e depois programações objeto de exclusão, executado em `Transaction.Commit()`.

Você sabe agora o ciclo de vida de persistência e as operações básicas do gerenciador de persistência. Juntos com os mapeamentos classe persistente discutimos no capítulo 3, você pode criar seus próprios pequenos NHibernate aplicação. (Se quiser, você pode saltar para o capítulo 10 e ler sobre uma classe helper útil para NHibernate `ISessionFactory` e `ISession` de gestão.) Tenha em mente que nós não mostrar qualquer exceção manipulação de código até agora, mas você deve ser capaz de descobrir o `tentar/pegar` blocos de si mesmo (como no capítulo 2).

Mapear algumas classes de entidade simples e componentes, e depois armazenar e carregar objetos em um console stand-alone aplicação (basta escrever um `Principal` método). No entanto, assim que você tentar armazenar objetos entidade associada, ou seja, quando você lida com um objeto mais complexo gráfico-você ver que a chamada `Save()` ou `Delete()` em cada objeto do gráfico não é uma forma eficiente de escrever aplicações.

Você gostaria de fazer chamadas como poucos para o `ISession` possível. Persistência transitiva fornece uma mais caminho natural para forçar mudanças de estado de objeto e de controlar o ciclo de vida de persistência.

4.3 persistência Usando transitiva em NHibernate

Real, as aplicações não triviais não lidar com objetos simples, mas com gráficos de objetos. Quando o aplicação manipula um gráfico de objetos persistentes, o resultado pode ser um objeto gráfico composto por persistente, individual, e transitórios casos. Persistência transitiva é uma técnica que permite que você para propagar a persistência de subgrafos transitório e destacadas automaticamente.

Por exemplo, se adicionarmos uma nova instanciada `Categoria` a hierarquia já persistente de categorias, deve tornar-se automaticamente persistente sem uma chamada para `Session.save()`. Demos um pouco diferente exemplo no capítulo 4, quando mapeamos uma relação pai / filho entre `Oferta` e `Item`. Nesse caso, não Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

apenas foram propostas feitas automaticamente persistentes quando foram adicionados a um item, mas também foram excluídos automaticamente quando o item possuir foi excluída.

Há mais de um modelo de persistência transitiva. O mais conhecido é persistência por acessibilidade, que vamos discutir em primeiro lugar. Apesar de alguns princípios básicos são os mesmos, NHibernate utiliza a sua própria, mais modelo poderoso, como você verá mais tarde.

4.3.1 Persistência por acessibilidade

Uma camada de persistência do objeto é dito implementar a persistência de acessibilidade se qualquer instância se torna persistente quando o aplicativo cria uma referência de objeto para a instância de outra instância que já está persistente.

Este comportamento é ilustrado pelo diagrama de objetos (note que este não é um diagrama de classes) na figura 5.2.

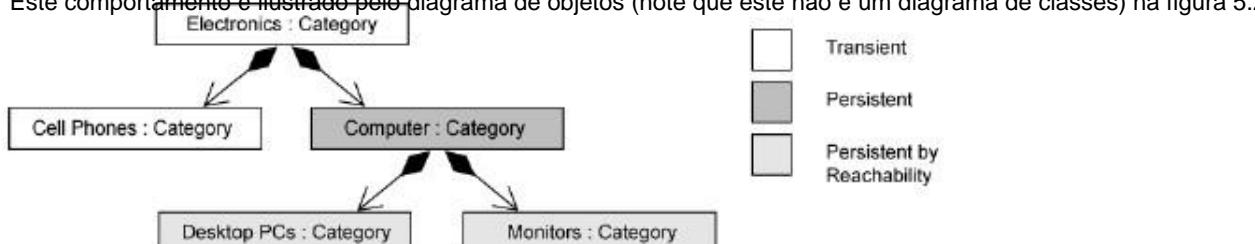


Figura 5.2 Persistência por acessibilidade com um objeto raiz persistente

Neste exemplo, "computador" é um objeto persistente. Os objetos "PCs Desktop" e "monitores" são também persistentes, eles são acessíveis a partir do "Computador" Categoría exemplo. "Electronics" e "Telefones" são transitórios. Note que nós assumimos a navegação só é possível para as categorias infantil, e não para o pai para exemplo, podemos chamar `computer.ChildCategories`. Persistência por acessibilidade é um algoritmo recursivo: Todos os objetos acessíveis a partir de uma instância persistente tornar persistente ou quando a instância original é feita persistentes ou um pouco antes na memória de estado é sincronizado com o armazenamento de dados.

Persistência por acessibilidade garante a integridade referencial, qualquer objeto gráfico pode ser totalmente recriado carregando o objeto raiz persistente. Um aplicativo pode andar o objeto gráfico a partir de associação para associação sem se preocupar com o estado persistente das instâncias. (Bancos de dados SQL tem uma abordagem diferente para integridade referencial, contando com chave estrangeira e outras restrições para detectar um aplicativo mal-comportados.)

Na forma mais pura de persistência por acessibilidade, o banco de dados tem algum nível superior, ou raiz, objeto de que todos os objetos persistentes são acessíveis. Idealmente, deve tornar-se uma instância transitória e ser excluído do banco de dados se não é acessível através de referências do objeto raiz persistente.

Soluções ORM nem NHibernate nem outros implementar esta forma, não há analógico da persistência de raiz objeto em um banco de dados SQL e não coleto de lixo persistente que pode detectar casos sem referência. Objeto armazena dados orientados pode implementar um algoritmo de coleta de lixo semelhante ao implementado no-objetos de memória pelo CLR, mas esta opção não está disponível no mundo ORM; digitalização de todas as tabelas para linhas não referenciados não desempenho aceitável.

Assim, a persistência de acessibilidade é na melhor das hipóteses uma solução de meio campo. Ele ajuda você a fazer objetos transitórios persistente e propagar seu estado para o banco de dados sem muitas chamadas para o gerenciador de persistência. Mas (pelo menos, no contexto de bancos de dados SQL e ORM) não é uma solução completa para o problema de fazer objetos persistentes transitória e remover seu estado a partir do banco de dados. Este acaba por ser um problema muito mais difícil. Você não pode simplesmente remover todas as instâncias acessível quando você remove um objeto; outras instâncias persistentes podem ter referências a eles (lembre-se que as entidades podem ser compartilhadas). Você não pode sequer remover com segurança instâncias que não são referenciada por qualquer objeto persistente na memória; as instâncias na memória são apenas um pequeno subconjunto de todos os objetos representados no banco de dados. Vejamos modelo de persistência NHibernate mais flexível transitiva.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

4.3.2 Cascading persistência com NHibernate

Modelo de persistência NHibernate transitiva usa o mesmo conceito básico como a persistência de acessibilidade, isto é, associações de objetos são examinados para determinar o estado transitivo. No entanto, NHibernate permite que você especifique um estilo cascata para cada mapeamento da associação, que oferece mais flexibilidade e controle refinado para todo o estado transições. NHibernate lê o estilo declarado e operações de cascatas de objetos associados automaticamente.

Por padrão, o NHibernate tem não navegar uma associação na busca de objetos transitórios ou individual, para salvar, excluir ou recolocar um **Categoria** não afetará os objetos categoria infantil. Este é o oposto de o comportamento padrão de persistência de acessibilidade. Se, por uma associação particular, que você deseja habilitar transitiva persistência, você deve substituir esse padrão nos metadados de mapeamento.

Você pode mapear associações entidade em metadados com os seguintes atributos:

cascade = "none", O padrão, diz NHibernate ignorar a associação.

cascade = "save-update" diz NHibernate para navegar a associação quando a transação é comprometido e quando um objeto é passado para **Save ()** ou **Update ()** e salvar recém instanciado transitória instâncias e manter as alterações em instâncias desconectadas.

cascade = "delete" diz NHibernate para navegar pela associação e excluir instâncias persistentes quando um objeto é passado para **Delete ()**.

cascade = "all" significa cascata tanto salvar-atualizar e excluir, bem como chamadas para **Desalojar** e **Fechadura**.

cascade = "all-delete-orphan" significa o mesmo que **cascade = "all"** mas, além disso, NHibernate exclui qualquer instância entidade persistente que tenha sido removido (dereferenced) a partir do associação (por exemplo, de uma coleção).

cascade = "delete-orphan" NHibernate irá excluir qualquer instância da entidade persistente que tem sido removido (dereferenced) da associação (por exemplo, de uma coleção).

Este associação de nível de estilo em cascata modelo é mais rico e menos seguro do que a persistência de acessibilidade.

NHibernate não faz as mesmas garantias fortes de integridade referencial que a persistência de acessibilidade fornece. Em vez disso, NHibernate parcialmente delegados preocupações de integridade referencial para as restrições de chave estrangeira de banco de dados relacional subjacente. Claro, há uma boa razão para esta decisão design: Ela permite que Aplicações para usar NHibernate destacado objetos de forma eficiente, porque você pode controlar o reatamento de um objeto gráfico destacado no nível de associação.

Vamos elaborar sobre o conceito em cascata com alguns mapeamentos associação exemplo. Recomendamos que você ler a próxima seção em um turno, pois cada exemplo baseia-se na anterior. Nossa primeiro exemplo é simples, que permite que você salve categorias recém-adicionado de forma eficiente.

4.3.3 Gerenciando categorias leilão

Administradores de sistema podem criar novas categorias, categorias renomear e subcategorias se movimentar no hierarquia categoria. Essa estrutura pode ser visto na figura 4.3.

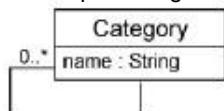


Figura 4.3 Categoria classe com associação com ela mesma

Agora, nós mapeamos essa classe e da associação:

```
<class name="Category" table="CATEGORY">
...
<property name="nome" column="CATEGORY_NAME"/>
<Many-to-one
    name = "ParentCategory"
    class = "Categoria"
    coluna = "PARENT_CATEGORY_ID"
/>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        cascade = "none" />
    <Set
        name = "ChildCategories"
        table = "CATEGORIA"
        cascade = "save-update"
        inverse = "true">
        <key column="PARENT_CATEGORY_ID"/>
        <one-to-many class="Category"/>
    </Set>
...
</Class>
```

Esta é uma recursiva, bidirecional, de associação um-para-muitos, como brevemente discutido no capítulo 3. O one-valorizado

final é mapeado com o `<many-to-one>` elemento e o **Conjunto** propriedade digitado com o `<set>`. Ambos referem-se a a coluna externa mesma chave: `PARENT_CATEGORY_ID`.

Suponha que criar um novo **Categoria** como uma categoria filho de "Computador" (ver figura 4.4).

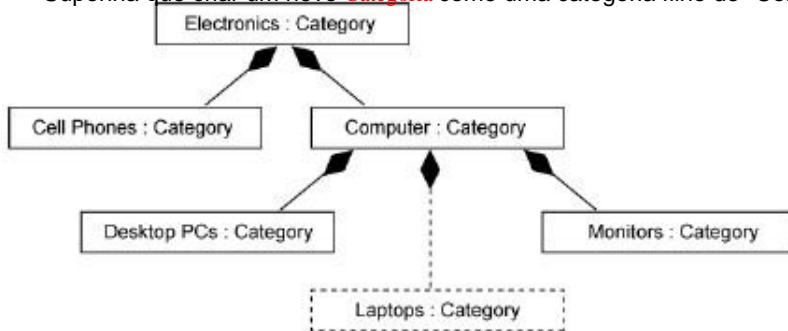


Figura 4.4 Adição de uma nova categoria de objeto gráfico

Nós temos várias maneiras de criar este novo "Laptops" objeto e salve-o no banco de dados. Poderíamos voltar para banco de dados e recuperar o "Computer" categoria para a qual o nosso novo "Laptops" categoria irá pertencer, adicione o

nova categoria, e confirmar a transação:

```

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
    Categoria computador = session.get <category> (computerID);
    Laptops categoria = new Categoria ("laptops");
    computador.ChildCategories.Add (laptops);
    laptops.ParentCategory = computador;
    session.Transaction.Commit ();
})
```

O **computador** instância é persistente (anexado a uma sessão), e os **ChildCategories** associação tem cascata salvar habilitado. Por isso, este código resultados no novo **laptops** categoria tornando-se persistente quando `Transaction.Commit ()` é chamado, porque NHibernate cascatas a operação suja controlo para o filhos de **computador**. NHibernate executa uma **INSERIR** declaração.

Vamos fazer a mesma coisa novamente, mas desta vez criar o link entre "Computer" e "laptops" fora de qualquer transação (em um aplicativo real, é útil para manipular um objeto gráfico em uma camada de apresentação para exemplo, antes de passar o gráfico de volta para a camada de persistência para fazer as alterações persistentes):

```

Computador categoria = ... / Carregado em uma sessão anterior
Laptops categoria = new Categoria ("laptops");
computador.ChildCategories.Add (laptops);
laptops.ParentCategory = computador;
```

O destacado **computador** objeto e quaisquer outros objetos destacado refere-se são associadas agora com o novo transitório **laptops** objeto (e vice-versa). Fazemos esta mudança para o gráfico do objeto persistente, salvando o novo objeto em uma segunda sessão NHibernate:

```

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    // Persistir uma nova categoria eo link para a respectiva categoria mãe
    Session.save (laptops);
    session.Transaction.Commit ();
}

```

NHibernate irá inspeccionar a propriedade de banco de dados identificador da categoria pai de `laptops` e corretamente criar a relação com o "Computer" de categoria no banco de dados. NHibernate insere o valor do identificador de o pai no campo de chave estrangeira da "Laptops" nova linha na **CATEGORIA**.

Desde `cascade = "none"` é definido para o `ParentCategory` associação, NHibernate ignora as alterações a qualquer das outras categorias na hierarquia ("Computer", "Eletrônica"). Não cascata a chamada para `Save ()` para entidades referidas por esta associação. Se tivéssemos habilitado `cascade = "save-update"` no `<many-to-one>` mapeamento de `ParentCategory`, NHibernate teria que navegar no gráfico completo de objetos na memória, sincronizando todas as instâncias com o banco de dados. Este processo seria um mau desempenho, porque um monte de acesso a dados inúteis seria necessário. Neste caso, não precisava nem queria transitiva persistência para o `ParentCategory` associação.

Por que nós temos em cascata de operações? Poderíamos ter salvo o `laptop` objeto, como mostrado nos últimos exemplo, sem qualquer mapeamento cascata sendo usado. Bem, considere o seguinte caso:

```

Computador categoria = ... / Carregadas em uma sessão anterior
Laptops categoria = new Categoria ("laptops");
Categoria laptopAccessories = nova categoria ("Acessórios para Notebook");
Categoria laptopTabletPCs = new Categoria ("Tablet PCs");
laptops.AddChildCategory (laptopAccessories);
laptops.AddChildCategory (laptopTabletPCs);
computer.AddChildCategory (laptops);

```

(Note que usamos o método de conveniência `AddChildCategory ()` para definir as duas extremidades do link de associação em uma chamada, como descrito no capítulo 3.)

Seria indesejável de ter que salvar cada uma das três novas categorias individualmente. Felizmente, porque mapeamos os `ChildCategories` associação com `cascade = "save-update"`, que não precisa. A mesma coisa código que usamos antes de salvar o single "Laptops" categoria irá salvar todas as três novas categorias em uma nova sessão:

```

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction ()) {
    // Continue todas as três instâncias nova categoria
    Session.save (laptops);
    session.Transaction.Commit ();
}

```

Você provavelmente está se perguntando por que o estilo cascata é chamado `cascade = "save-update"` em vez de `cascade = "save"`. Tendo feito apenas três categorias persistente anteriormente, suponha que nós fizemos a seguintes alterações na hierarquia da categoria em uma solicitação subsequente (fora de uma sessão e transação):

```

laptops.Name = "computadores portáteis";
laptopAccessories.Name = "Acessórios e Peças";
laptopTabletPCs.Name = "Computadores Tablet";
Categoria laptopBags = nova categoria ("Laptop Bags");
laptops.AddChildCategory (laptopBags);

```

Nós adicionamos uma nova categoria como um filho do "Laptops" categoria e modificou todas as três categorias existentes.

O código a seguir atualiza três antigos Categoria instâncias e insere o novo:

```

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction ()) {

```

```

    Session.update (laptops);
    session.Transaction.Commit ();
}

```

Especificando `cascade = "save-update"` no `ChildCategories` associação reflete com precisão o fato NHibernate que determina o que é necessário para manter os objetos para o banco de dados. Neste caso, será recolocar / atualizar as três categorias individual (`laptops`, `laptopAccessories` E `laptopTabletPCs`) e salvar a categoria nova criança (`laptopBags`).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Observe que o exemplo de código a última difere da anterior sessão apenas dois exemplos em um único método chamada. O último exemplo usa `Update()` em vez de `Save()` porque `laptops` já era persistente.

Podemos reescrever todos os exemplos para usar o `SaveOrUpdate()` método. Em seguida, os três trechos de código são idêntica:

```
usando (sessão ISession sessionFactory.openSession = ()  
usando (session.BeginTransaction () {  
    // Vamos NHibernate decidir o que é novo eo que é destacada  
    session.SaveOrUpdate (laptops);  
    session.Transaction.Commit ();  
}
```

O `SaveOrUpdate()` método informa NHibernate para propagar o estado de uma instância para o banco de dados criar uma linha nova base de dados se a instância é uma nova instância transitória ou atualizar a linha existente se o instância é uma instância separada. Em outras palavras, ele faz exatamente a mesma coisa com o `laptops` categoria como `cascade = "save-update"` fez com o filho de categorias `laptops`.

Uma pergunta final: Como é que NHibernate saber quais as crianças foram destacados e que eram novos casos transitórios?

4.3.4 A distinção entre casos transitórios e destacada

Desde NHibernate não mantém uma referência a uma instância individual, você tem que deixar NHibernate saber como distinguir entre uma instância separada, como `laptops` (Se ele foi criado em uma sessão anterior) e um novo instância transitória como `laptopBags`.

Uma gama de opções está disponível. NHibernate irá assumir que uma instância é uma instância transitória não salvos se:

A propriedade identificador (se existir) é `nulo`.

A propriedade versão (se existir) é `nulo`.

Você fornece um `unsaved valor` no documento de mapeamento para a classe, eo valor do identificador jogos propriedade.

Você fornece um `unsaved valor` no documento de mapeamento para a `versão` propriedade, eo valor de o `versão` jogos propriedade.

Você fornecer um NHibernate `IInterceptor` e retornar `verdadeiro` a partir de `IInterceptor.IsUnsaved()` depois verificar a instância em seu código.

No nosso modelo de domínio, temos usado o tipo primitivo `longo` como o nosso tipo de propriedade de identificação em todos os lugares. Como

não é anulável, temos que usar o mapeamento seguinte identificador em todas as nossas classes:

```
<class name="Category" table="CATEGORY">  
    <id name="Id" unsaved-value="0">  
        <generator class="native"/>  
    </Id>  
    ...  
</Class>
```

O `unsaved valor` atributo diz NHibernate para tratar casos de `Categoria` com um valor de identificador de `0` como recém instanciado casos transitórios. O valor padrão para o atributo `unsaved valor` é `nulo` se o tipo é anulável; mais, é o valor padrão do tipo (que é `0` para tipo numérico), assim, uma vez que nós escolhemos `longo` como o nosso tipo de propriedade identificador, podemos omitir o `unsaved valor` atributo em nosso aplicativo de leilão classes (que usam o tipo de identificador mesmo em toda parte). Tecnicamente, NHibernate tentar adivinhar a `unsaved valor` instanciando um objeto vazio e recuperar valores de propriedade padrão a partir dele.

Não salvos identificadores atribuídos

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Essa abordagem funciona muito bem para identificadores sintéticos, mas ele quebra, no caso de chaves atribuído pelo a aplicação, incluindo chaves compostas em sistemas legados. Discutimos esta questão no capítulo 9, seção 9.2, "esquemas Legacy e chaves compostas." Evitar a aplicação atribuído (e composto) chaves em novas aplicações, se possível (isto é importante para não-versionados entidades).

Agora você tem o conhecimento necessário para otimizar sua aplicação NHibernate e reduzir o número de chamadas para o gerenciador de persistência, se você deseja salvar e excluir objetos. Verifique o **unsaved valor** atributos de todos os seus classes e experimentar com objetos destacados para obter um sentimento para o modelo de persistência NHibernate transitiva.

Ter focado em como nós persistir objetos com NHibernate, podemos mudar agora e se concentrar em perspectivas como nós vamos sobre como recuperar (ou carregamento) deles.

4.4 Recuperando objetos

Recuperar objetos persistentes do banco de dados é um dos mais interessantes (e complexo) Peças de trabalhar com NHibernate. NHibernate fornece as seguintes maneiras para obter objetos para fora do banco de dados:

Navegando na gráfico do objeto, a partir de um objeto já está carregada, acessando os objetos associados através de métodos de acessador de propriedade, como `aUser.Address.City`. NHibernate automaticamente carga (ou pré-carga) nós do grafo enquanto você navega o gráfico se o `ISession` está aberta.

Recuperando pelo identificador, que é o método mais conveniente e de alto desempenho quando o identificador único

valor de um objeto é conhecido.
Usando o Hibernate Query Language (HQL), que é uma linguagem de consulta completa orientada a objetos.

Usando o NHibernate `ICriteria` API, que fornece uma maneira tipo seguro e orientado a objeto para realizar consultas sem a necessidade de manipulação de string. Esta instalação inclui consultas com base em um exemplo objeto.

Usando consultas SQL nativas, onde NHibernate cuida do mapeamento o resultado ADO.NET conjuntos de gráficos
de objetos persistentes.

Em suas aplicações NHibernate, você vai usar uma combinação dessas técnicas. Cada método de recuperação pode usar uma estratégia de buscar diferentes, isto é, uma estratégia que define que parte do gráfico do objeto persistente deve ser recuperado. O objetivo é encontrar o melhor método de recuperação e estratégia de busca para cada caso de uso em seu aplicação e, ao mesmo tempo minimizando o número de consultas SQL para um melhor desempenho.

Nós não vamos discutir cada método de recuperação em muitos detalhes nesta seção, em vez disso vamos nos concentrar sobre a base estratégias de busca e como ajustar arquivos de mapeamento do NHibernate para o melhor desempenho padrão buscar para todos

métodos. Antes de olharmos para as estratégias de busca, vamos dar uma visão geral dos métodos de recuperação. Note-se que

nós também enção do sistema de cache NHibernate, mas explorá-la totalmente no próximo capítulo.

Vamos começar com o caso mais simples, a recuperação de um objeto, dando o seu valor identificador (navegar o objeto gráfico deve ser auto-explicativo). Você viu uma recuperação simples identificador anteriormente neste capítulo, mas há mais para saber sobre ele.

4.4.1 Recuperando objetos pelo identificador

O seguinte trecho de código recupera uma NHibernate `Usuário` objeto do banco de dados:

```
User = usuário session.get <User> (userID);  
NET 2.0 e sem genéricos.:  
Usuário user = (User) session.get (typeof (User), userID);
```

O `Get ()` método é especial porque o identificador identifica uma única instância de uma classe. Por isso, é comum para aplicações para usar o identificador como uma alça conveniente para um objeto persistente. Recuperação por identificador pode usar o cache quando recuperar um objeto, evitando-se um banco de dados hit se o objeto já está em cache.

NHibernate também fornece uma `Load ()` método:
Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
User = usuário Session.load <User> (userID);
```

A diferença entre estes dois métodos é trivial:

Se `Load()` não pode encontrar o objeto no cache ou banco de dados, uma exceção é lançada. O `Load()` método nunca retorna `nulo`. O `Get()` método retorna `nulo` se o objeto não pode ser encontrado.

O `Load()` método pode retornar um proxy em vez de um exemplo real persistente (durante o carregamento preguiçoso é

) habilitado. Um proxy é um espaço reservado que desencadeia o carregamento do objeto real, quando é acessada pela primeira vez tempo; discutimos proxies posteriormente nesta seção. É importante compreender que `Load()` irá retornar um proxy mesmo que não haja nenhuma linha com o identificador especificado e uma exceção será lançada se (e somente se) NHibernate tentar carregá-lo. Por outro lado, `Get()` nunca retorna um proxy como ela deve retornar null se a entidade não existe.

Escolher entre `Get()` e `Load()` é fácil: Se você estiver certo que o objeto persistente existe, e inexistência seria considerado excepcional, `Load()` é uma boa opção. Se você não está certo há uma instância persistente com o identificador especificado, use `Get()` e testar o valor de retorno para ver se é `nulo`.

E se esse objeto já está no cache da sessão como um proxy não inicializada? Neste caso, `Load()` vontade devolver o proxy como é, mas `Get()` irá inicializar-lo antes de devolvê-lo.

Utilização `Load()` tem uma implicação adicional: A aplicação pode recuperar um válido referência (Um proxy) para um instância persistente sem bater o banco de dados para recuperar seu estado persistente. Assim `Load()` não pode lançar uma exceção quando ele não encontrar o objeto persistente no cache ou banco de dados, a exceção pode ser gerada mais tarde, quando o proxy é acessado.

Há uma interessante aplicação deste comportamento. Vamos dizer que o carregamento lento é habilitado na classe `Categoria` e analisar o seguinte código:

```
usando (sessão ISession sessionFactory.openSession = ()  
usando (session.BeginTransaction ()) {  
    Pai category = Session.load <category> (ANID);  
    Console.WriteLine (parent.Id);  
    Criança categoria = new Categoria ("teste");  
    child.ParentCategory pai =;  
    Session.save (crianças);  
    session.Transaction.Commit ();  
}
```

Nós carregado pela primeira vez uma categoria. NHibernate não atingiu o banco de dados para fazer isso, ele retorna um proxy. Acessando o identificador desta proxy não causar sua inicialização (desde que o identificador é mapeado com o acesso estratégia "Propriedade" ou "Nosetter"). Então, ligamos uma nova categoria para o proxy e nós salvá-lo. Um Instrução INSERT é executado para salvar a linha com o valor de chave estrangeira de identificador do proxy. Não Instrução SELECT é executada em tudo!

Agora, vamos descobrir consultas arbitrárias que são muito mais flexíveis do que recuperar os objetos pelo identificador.

4.4.2 Apresentação do Hibernate Query Language

O Hibernate Query Language é um dialeto orientada a objeto do familiar relacional consulta linguagem SQL. HQL tem semelhanças perto ODMG OQL e EJB-QL (de Java), mas ao contrário OQL, é adaptado para uso com bancos de dados SQL, e é muito mais poderosa e elegante do que EJB-QL. No entanto, JPA QL é na verdade um subconjunto de HQL. HQL é fácil de aprender com conhecimentos básicos de SQL.

HQL não é uma linguagem de manipulação de dados como SQL. É utilizado somente para recuperação de objetos, não para atualização, inserção ou exclusão de dados. Sincronização de estado do objeto é o trabalho do gerenciador de persistência, e não o desenvolvedor.

Na maioria das vezes, você só precisa recuperar objetos de uma classe particular e restringir pelas propriedades de dessa classe. Por exemplo, a consulta a seguir recupera um usuário pelo primeiro nome:
Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
IQuery session.CreateQuery q = ("de onde u usuário u.Firstname =: fname");
q.SetString ("fname", "Max");
IList <User> resultado = q.List <User>();
```

Depois de preparar consulta `q`, Ligamos o valor do identificador a um parâmetro nomeado, `fname`. O resultado é retornado como um genérico `IList` de `Usuário` objetos.

Note-se que, em vez de obter essa lista, podemos oferecer um usando `q.List (myEmptyList)` e NHibernate irá preenchê-lo. Isto é muito útil quando você quiser usar uma coleção com funcionalidades adicionais (Como avançados de ligação de dados).

HQL é poderoso, e mesmo que você não pode usar os recursos avançados o tempo todo, você vai precisar deles para alguns problemas difíceis. Por exemplo, HQL suporta o seguinte:

A capacidade de aplicar restrições às propriedades de objetos conexos relacionados por referência ou realizada em coleções (para navegar o objeto gráfico usando linguagem de consulta).

A capacidade de recuperar apenas as propriedades de uma entidade ou entidades, sem a sobrecarga de carregar a entidade

-se em um escopo transacional. Isso às vezes é chamado de consulta do relatório, é mais corretamente chamada projeção.

A habilidade para ordenar os resultados da consulta.

A capacidade para paginar os resultados.

Agregação com `grupo por`, `ter` Funções e agregado, como `soma, min E max`.

Junções externas ao recuperar vários objetos por linha.

A capacidade de chamar funções definidas pelo usuário

`SQL`

Subconsultas (consultas aninhadas).

Discutimos todas estas características no capítulo 7, juntamente com o mecanismo de consulta SQL nativo opcional. Nós agora olhar para uma outra abordagem para issueing consultas com NHibernate - Consulta por Critérios.

4.4.3 Consulta por critérios

O NHibernate consulta por critérios (QBC) API permite que você construa uma consulta através da manipulação de objetos em critérios tempo de execução. Esta abordagem permite que você especifique as restrições dinamicamente sem manipulações sequência direta, mas não perde muito da flexibilidade ou poder de HQL. Por outro lado, as consultas expressas como critérios são muitas vezes menos legível do que as consultas expressas em HQL.

Recuperando pelo primeiro nome é fácil usar um `Critérios` objeto:

```
ICriteria criteria = session.CreateCriteria (User);
criteria.Add (Expression.Like ("Nome", "Max"));
IList resultado = criteria.List ();
```

Um `ICriteria` é uma árvore de `ICriterion` instâncias. O `Expressão` classe fornece métodos estáticos fábrica que retornam `ICriterion` instâncias. Uma vez que o desejado critérios árvore é construída, é executado no banco de dados.

Muitos desenvolvedores preferem QBC, considerando-a uma abordagem mais orientada a objetos. Eles também gostam do fato de que a sintaxe de consulta pode ser analisado e validado em tempo de compilação, enquanto expressões HQL não são analisados até tempo de execução.

A coisa agradável sobre o NHibernate `ICriteria` API é o `ICriterion` quadro. Este quadro permite a extensão pelo usuário, que é difícil no caso de uma linguagem de consulta como HQL.

4.4.4 Consulta por exemplo

Como parte da instalação QBC, NHibernate suporta consulta por exemplo (QBE). A idéia por trás QBE é que o aplicativo fornece uma instância da classe consultado com certas definir valores de propriedade (para não padrão valores). O

consulta retorna todas as instâncias persistentes com correspondência de valores de propriedade. QBE não é particularmente poderosa

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

abordagem, mas pode ser conveniente para algumas aplicações. O trecho de código a seguir demonstra um NHibernate QBE:

```
Usuário exampleUser = new User();
exampleUser.Firstname = "Max";
ICriteria critérios session.CreateCriteria = (typeof (User));
critérios.add (Example.Create (exampleUser));
IList resultado = critérios.List();
```

Um caso de uso típico para QBE é uma tela de pesquisa que permite aos usuários especificar um intervalo de valores de propriedade para ser acompanhado pelo conjunto de resultados retornado. Este tipo de funcionalidade pode ser difícil de expressar de forma limpa em uma consulta linguagem; manipulações cadeia seria necessária para especificar um conjunto dinâmico de restrições.

Tanto a API QBC eo mecanismo de consulta de exemplo são discutidos em mais detalhes no capítulo 7.

Você já sabe as opções de recuperação básico em NHibernate. Nós nos concentramos sobre as estratégias para a recuperação de objeto gráficos no resto desta seção. A estratégia de busca define que parte do objeto gráfico (ou, o que subgrafo) é recuperado com uma consulta ou operação de carregamento.

4.4.5 Buscando estratégias

Na tradicional de acesso de dados relacional, você obtém todos os dados necessários para um cálculo especial com o único

Consulta SQL, aproveitando interior e exterior une para recuperar entidades relacionadas. Alguns ORM primitiva implementações buscar dados fragmentada, com muitos pedidos de pequenos pedaços de dados em resposta à aplicativo navegar um gráfico de objetos persistentes. Esta abordagem não faz uso eficiente dos capacidades de participar do banco de dados relacional. Na verdade, esses dados escalas estratégia de acesso ao mal por natureza. Um dos mais problemas difíceis na ORM, provavelmente o mais difícil é prever o acesso eficiente aos dados relacionais, dada uma aplicação que prefere tratar os dados como um gráfico de objetos.

Para os tipos de aplicações, muitas vezes nós trabalhamos com (multi-usuário, distribuído, web, e da empresa aplicações), recuperação de objetos utilizando muitas idas e vindas de / para o banco de dados é inaceitável. Por isso defendemos

que as ferramentas devem enfatizar a R ORM para uma extensão muito maior do que tem sido tradicional.

O problema de buscar gráficos de objetos de forma eficiente (com acesso mínimo ao banco de dados) tem sido freqüentemente abordado, proporcionando estratégias de associação em nível de busca especificada nos metadados do mapeamento da associação. O problema com esta abordagem é que cada pedaço de código que utiliza uma entidade requer um diferente conjunto de objetos associados. Mas isso não é suficiente. Argumentamos que o que é necessário é o suporte para granulação fina tempo de execução associação buscar estratégias. NHibernate suporta tanto, ele permite que você especifique uma estratégia de buscar no padrão arquivo de mapeamento e então substituí-lo em tempo de execução no código.

NHibernate permite que você escolha entre quatro estratégias de fetching para qualquer associação, em associação metadados

Buscando o campo de imediata

sequencial (ou

pesquisa de cache).

Coleta de buscar-O objeto associado ou preguiçoso é buscada "preguiçosamente", quando é acessado pela primeira vez. Este

resulta em um novo pedido ao banco de dados (a menos que o objeto associado é armazenado em cache). Coleta de buscar-O objeto associado ou ansiosos é obtido juntamente com o objeto proprietário, usando um SQL junção externa, e nenhum pedido de banco de dados adicional é necessária.

Busca-Este lote abordagem pode ser usada para melhorar o desempenho de preguiçosos buscar por recuperar um lote de objetos ou coleções, quando uma associação preguiçoso é acessado. (A busca em lote também pode ser usado para

melhorar o desempenho da busca imediata.)

Vamos olhar mais de perto em cada estratégia de busca.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Busca imediata

Buscar associação imediata ocorre quando você recuperar uma entidade do banco de dados e logo em seguida recuperar uma outra entidade associada ou entidades em um novo pedido para o banco de dados ou cache. Busca imediata

geralmente não é uma estratégia de busca eficiente, a menos que você espera que as entidades associadas à quase sempre ser armazenados em cache já.

Busca preguiçosa

Quando um cliente solicita uma entidade e suas associadas gráfico de objetos do banco de dados, não é normalmente necessárias para recuperar o gráfico completo de cada objeto (indiretamente) associados. Você não gostaria de carregar o banco de dados inteiro para a memória de uma só vez, por exemplo, o carregamento de um único **Categoria** não deve acionar o carregamento de todos **Items** nessa categoria.

Busca preguiçosa permite que você decida o quanto do objeto gráfico é carregado no sucesso primeiro banco de dados e que associações devem ser carregados somente quando eles estão acessada pela primeira vez. Busca preguiçosa é um conceito fundamental na persistência de objetos eo primeiro passo para alcançar um desempenho aceitável.

Desde NHibernate 1.2, todas as associações estão configurados para a busca preguiçosa por padrão, e você pode facilmente alterar esse comportamento através da criação `default-lazy = "false"` em `<hibernate-mapping>` do seu mapeamento arquivos. Mas recomendamos que você mantenha esta estratégia e substituí-lo em tempo de execução por consultas que a **Ansioso (outer join) fetching**.

Buscar associação preguiçoso pode ajudar a reduzir a carga de banco de dados e muitas vezes é uma estratégia de bom padrão. No entanto, é uma pouco como um cego acho que na medida em que a otimização do desempenho vai.

Procura antecipada permite que você especifique explicitamente que objetos associados devem ser carregados juntamente com o fazendo referência a objeto. NHibernate pode, então, retornar os objetos associados em um pedido único banco de dados, utilizando um **SQL OUTER JOIN**. Otimização de desempenho em NHibernate muitas vezes envolve o uso criterioso de buscar ansiosos para transacções específicas. Assim, mesmo que a busca padrão ansiosos podem ser declaradas no arquivo de mapeamento, **A busca em lote** mais comum para especificar o uso desta estratégia em tempo de execução para uma consulta HQL em particular ou **A busca em lote** não é estritamente uma estratégia de busca de associação, é uma técnica que pode ajudar a melhorar a desempenho de buscar (ou imediata) preguiçoso. Normalmente, quando você carregar um objeto ou uma coleção, o seu **SQL ONDE**

cláusula especifica o identificador do objeto ou objeto que possui a coleção. Se busca em lote é habilitado, NHibernate olha para ver o que outras instâncias proxy ou coleções não inicializadas são referenciados no atual sessão e tenta carregá-los ao mesmo tempo, especificando valores identificador múltiplo na **ONDE** cláusula.

Nós não somos grandes fãs desta abordagem; busca ansiosa é quase sempre mais rápido. A busca em lote é útil para usuários inexperientes que desejam obter um desempenho aceitável em NHibernate sem ter que pensar muito dura sobre o SQL que será executado.

Vamos agora declarar a estratégia de busca de algumas associações em nossa metadados de mapeamento.

4.4.6 Seleção de uma estratégia de busca em mapeamentos

NHibernate permite selecionar estratégias de busca padrão de associação de atributos que especifica no mapeamento metadados. Você pode substituir a estratégia padrão usando os recursos de métodos de consulta do NHibernate, como você verá na

capítulo 7. Uma advertência menor: Você não tem que entender todas as opções apresentadas nesta seção imediatamente;

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

recomendamos que você obter uma visão geral primeiro e use esta seção como uma referência quando você estiver otimizando o padrão buscar estratégias na sua aplicação.

Uma ruga em formato NHibernate de mapeamento significa que os mapeamentos de coleta de função um pouco diferente do que um único ponto de associações, por isso, nós vamos cobrir os dois casos separadamente. Vamos primeiro considerar ambas as extremidades da associação bidirecional entre **Oferta** e **Item**.

Único ponto de associação

Para um **<many-to-one>** ou **<one-to-one>** associação, a busca preguiçosa só é possível se a classe de associado mapeamento permite proxy. Para o **Item** classe, que permitem proxy, especificando **lazy = "true"** (Desde NHibernate 1.2, este é o valor padrão):

```
<class name="Item" lazy="true">
```

Agora, lembre-se a associação de **Oferta** para **Item**:

```
<many-to-one name="item" class="Item">
```

Quando recuperar um **Oferta** do banco de dados, a propriedade associação pode deter uma instância de uma NHibernate subclasse gerada de **Item** que os delegados todas as invocações método para uma instância diferente de **Item** que é trazida de forma tardia do banco de dados (esta é a definição mais elaborada de um proxy NHibernate).

A fim de delegar método (e propriedade) invocações, esses membros precisam ser **virtual**. NHibernate 1,2 usa um validador que verifica se suas entidades proxied ter um construtor padrão que não é privado, que eles não são **selado**, Que todos os métodos públicos e propriedades são virtuais e que não há campo público. Ele é possível desativar isso validador, mas você deve pensar cuidadosamente sobre o porquê de você fazer isso. Aqui é o elemento para adicionar ao seu arquivo de configuração para desligá-lo:

```
<property name="hibernate.use_proxy_validator"> false </property>
```

Ou você pode fazê-lo de forma programática, antes de construir a fábrica de sessão, usando:

```
cfg.Properties [NHibernate.Cfg.Environment.UseProxyValidator] = "false".
```

NHibernate usa duas instâncias diferentes, de modo que até mesmo associações polimórficas podem ser proxy, quando o objeto em proxy é obtido, pode ser uma instância de uma subclasse de mapeado **Item** (Se houver algum subclasses de **Item**, Que é). Podemos até mesmo escolher qualquer interface implementada pelo **Item** classe como o tipo de proxy. Para fazê-lo, declará-la usando o **procuração** atributo, em vez de especificar **preguiçoso = "True"**:

```
<class name="Item" proxy="ItemInterface">
```

Assim que declarar a **procuração** ou **preguiçoso** atributo em **Item**, Qualquer associação de ponto único para **Item** é proxied e trazida de forma tardia, a não ser que a associação substitui a estratégia de busca, declarando a **exterior-Join** atributo.

Existem três valores possíveis para **outer-join**:

outer-join = "auto"-O padrão. Quando o atributo não é especificado; busca o NHibernate objeto associado preguiçosamente se a classe de associado permitiu proxy, ou ansiosamente usando um outer join se

proxy estiver desativado (default).

outer-join = "true"-NHibernate sempre busca a associação inteira usando uma junção externa, mesmo se proxy está habilitado. Isso permite que você escolha diferentes estratégias de fetching para associações diferentes para

mesma classe proxy. É equivalente a **fetch = "join"**.

outer-join = "false"-NHibernate nunca vai buscar a associação com uma associação externa, mesmo se proxy está desativado. Isto é útil se você espera que o objeto associado a existir no cache de segundo nível (Ver capítulo 5). Se ele não estiver disponível no cache de segundo nível, o objeto é obtido imediatamente através de um

extras SQL **SELEÇÃO**. Esta opção é equivalente a **fetch = "select"**.

Assim, se quiséssemos reativar buscar ansioso para a associação, agora que o proxy está habilitado, nós especificar

```
<many-to-one name="item" class="Item" outer-join="true">
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Para um **um-para-um** associação (discutidos em mais detalhes no capítulo 6), a busca preguiçosa é conceitualmente possível

somente quando o objeto associado sempre existe. Nós indicamos isso especificando `constrained = "true"`. Para exemplo, se um item pode ter apenas um lance, o mapeamento para o **Oferta** é

```
<one-to-one name="item" class="Item" constrained="true">
```

O **constrangido** atributo tem uma interpretação um pouco semelhante ao **não-nulo** atributo de um **<Many-to-um>** mapeamento. Ele diz NHibernate que o objeto associado é exigido e, portanto, não pode ser **nulo**.

Para habilitar busca em lote, especificamos o **batch-size** no mapeamento de **Item**:

```
<class name="Item" lazy="true" batch-size="9">
```

O tamanho do lote limita o número de itens que podem ser recuperados em um único lote. Escolher um razoavelmente pequeno número aqui.

Você vai encontrar os mesmos atributos (**outer-join**, **batch-size** E **preguiçoso**) Quando consideramos coleções, mas a interpretação é um pouco diferente.

Coleções

No caso de coleções, as estratégias de busca não se aplica apenas para as associações entidade, mas também para coleções de

valores (por exemplo, uma coleção de seqüências pode ser obtida pela junção externa).

Assim como classes, coleções têm seus próprios proxies, que costumamos chamar wrappers coleção. Ao contrário das classes, o invólucro coleção está sempre lá, mesmo que a busca preguiçosa é desativado (NHibernate necessidades do wrapper para detectar modificações coleção).

Mapeamento de coleção pode declarar uma **preguiçoso** atributo, uma **outer-join** atributo, nenhum dos dois, ou ambos (especificando tanto não é significativo). As opções significativas são as seguintes:

Nem atributo especificado-Esta opção é equivalente a `outer-join = "false" lazy = "false"`.

A coleção é obtido a partir da cache de segundo nível ou por um SQL imediata extras **SELEÇÃO**. Este opção é mais útil quando o cache de segundo nível está habilitado para esta coleção.

outer-join = "true"-NHibernate busca a associação inteira usando uma associação externa. No momento da isto foi escrito, NHibernate é capaz de buscar apenas uma coleção por SQL **SELEÇÃO**, Por isso não é possível declarar várias coleções pertencentes à mesma classe persistente com `outer-join = "true"`.

lazy = "true"-NHibernate obtém a coleção preguiçosamente, quando é acessado pela primeira vez. Desde NHibernate

1.2, esta é a opção padrão e recomendamos que você mantenha esta opção como um padrão para todos os seus coleção de mapeamentos.

Não recomendamos a procura antecipada para as coleções, por isso vamos mapa coleção do item das propostas com **lazy = "true"**. Esta opção é quase sempre utilizado para mapeamento de coleção (embora seja o padrão desde NHibernate 1.2, vamos continuar a escrevê-lo a insistir nele):

```
<set name="Bids" lazy="true">
  <key column="ITEM_ID"/>
  <one-to-many class="Bid"/>
</Set>
```

Podemos até mesmo habilitar busca em lote para a coleção. Neste caso, o tamanho do lote não se refere ao número de lances no lote, que se refere ao número de coleções de lances:

```
<set name="Bids" lazy="true" batch-size="9">
  <key column="ITEM_ID"/>
  <one-to-many class="Bid"/>
</Set>
```

Este mapeamento diz NHibernate para carregar até nove coleções de lances em um lote, dependendo de quantas coleções não inicializadas de licitações estão presentes nos itens associados com a sessão. Em outras palavras, se há cinco **Item** instâncias com estado persistente em um **ISession**, E todos têm um não inicializada **Lances** coleta, NHibernate irá carregar automaticamente todos os cinco coleções em uma única consulta SQL se for acessado. Se existem 11 itens, apenas 9 coleções serão buscados. A busca em lote pode reduzir significativamente o número de

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

consultas necessárias para hierarquias de objetos (por exemplo, ao carregar a árvore de pai e filho **Categoria** objetos).

Vamos falar sobre um caso especial: muitos-para-muitos (discutiremos esse mapeamento com mais detalhes no capítulo 6). Você costuma usar uma tabela link (alguns desenvolvedores também chamá-la de mesa relacionamento ou tabela de associação) que contém apenas os valores-chave das duas tabelas associadas e, portanto, permite uma multiplicidade de muitos para muitos.

Esta tabela adicional deve ser considerado se você decidir usar a busca ansiosa. Olhe para o seguinte exemplo, muitos-para-muitos simples, que mapeia a associação de **Categoria** para **Item**:

```
<set name="Items" outer-join="true" table="CATEGORY_ITEM">
  <key column="CATEGORY_ID"/>
  <many-to-many column="ITEM_ID" class="Item"/>
</Set>
```

Neste caso, a estratégia de busca ansiosa refere-se apenas à tabela de associação **CATEGORY_ITEM**. Se carregar um **Categoria** com esta estratégia de busca, NHibernate irá buscar automaticamente todas as entradas de ligação de **CATEGORY_ITEM** em uma única associação externa consulta SQL, mas não as instâncias item **ITEM**!

As entidades constantes da associação muitos-para-muitos pode também, naturalmente, ser buscado ansiosamente com o

consulta SQL mesmo. O **<many-to-many>** elemento permite que esse comportamento para ser personalizado:

```
<set name="Items" outer-join="true" table="CATEGORY_ITEM">
  <key column="CATEGORY_ID"/>
  <many-to-many column="ITEM_ID" outer-join="true" class="Item"/>
</Set>
```

NHibernate agora vai buscar todas as **Items** em um **Categoria** com uma única consulta de junção externa quando o **Categoria** é carregado. No entanto, tenha em mente que geralmente recomendo carregamento lento como a estratégia de busca padrão e

NHibernate que é limitado a um ansiosamente buscada recolha por classe persistente mapeada.

Definir a profundidade de

buscar

Vamos agora discutir uma definição de estratégia global a busca: a máxima buscar profundidade. Essa configuração controla o número de outer-juntou-se tabelas NHibernate irá usar em uma consulta SQL simples. Considere a associação completa cadeia de **Categoria** para **ItemE**, a partir **Item** para **Oferta**. O primeiro é uma associação muitos-para-muitos e segundo é um um-para-muitos, daí as duas associações são mapeadas com elementos da coleção. Se nós declaramos **outer-join = "true"** para ambas as associações (não esqueça o especial **<many-to-many>** declaração) e carregar um único **Categoria**, Quantas consultas serão NHibernate executar? Só o **Items** ser ansiosamente obtida, ou também todos os **Ofertas** de cada **Item**?

Você provavelmente esperaria uma única consulta, com uma operação de junção externa, incluindo a **CATEGORIA**, **CATEGORY_ITEM**, **ITEM** E **BID** tabelas. No entanto, este não é o caso por padrão.

Comportamento exterior join fetch NHibernate é controlada com a opção de configuração global **hibernate.max_fetch_depth**. Se você ajustar para 1 (Também o padrão), NHibernate irá buscar apenas as **Categoria** e as entradas de link a partir do **CATEGORY_ITEM** tabela de associação. Se você configurá-lo para 2, NHibernate executa uma junção externa que também inclui a **Items** na consulta SQL mesmo. Definir esta opção para 3 não será, como você poderia esperar, também incluem as propostas de cada item na consulta SQL mesmo. A limitação a um coleção juntou exterior se aplica aqui, impedindo lenta produtos cartesianos.

Valores recomendados para a profundidade buscar depender do desempenho e juntar o tamanho do banco de dados mesas; testar seus aplicativos com valores baixos (menos de 4) Primeiro, e diminuir ou aumentar o número enquanto ajuste a sua aplicação. O máximo global profundidade buscar também se aplica a single-ended associação (**<Many-to-one>**, **<one-to-one>**) Mapeado com uma estratégia de busca ansiosa ou usando o padrão auto.

Tenha em mente que as estratégias de busca ansiosa declaradas nos metadados de mapeamento são eficazes apenas se você usar recuperção por identificador, use a API critérios de consulta, ou navegar pelo gráfico do objeto manualmente. Qualquer HQL

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

consulta pode especificar sua própria estratégia de busca em tempo de execução, ignorando assim os padrões de mapeamento. Você também pode substituir os padrões (isto é, não ignorá-los) com consultas critérios. Esta é uma diferença importante, e nós cobri-lo com mais detalhes no capítulo 7, seção 7.3.2, "Buscando associações."

No entanto, você pode às vezes simplesmente como inicializar um proxy ou um invólucro coleção manualmente com um chamada de API simples.

Inicializando associações preguiçoso

Um wrapper proxy ou coleção é automaticamente inicializado quando qualquer um de seus métodos são chamados (exceto o getter do identificador propriedade, que pode retornar o valor do identificador, sem buscar os persistentes subjacentes objeto). No entanto, só é possível inicializar um proxy ou wrapper coleção se é atualmente associado aberto `ISession`. Se você fechar a sessão e tentar acessar um proxy não inicializada ou coleção, NHibernate lança um `LazyInitializationException`.

Devido a esse comportamento, às vezes é útil para inicializar explicitamente um objeto antes de fechar a sessão. Esta abordagem não é tão flexível como recuperar o objeto subgrafo completo exigido com uma consulta HQL, usando arbitrária estratégias de busca em tempo de execução.

Nós usamos o método estático `NHibernateUtil.Initialize()` para inicialização manual:

```
usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction () {
    Categoria cat = session.get <category> (id);
    NHibernateUtil.Initialize (cat.Items);
    session.Transaction.Commit ();
}
foreach (item Item em cat.Items)
...
}
```

`NHibernateUtil.Initialize()` pode ser passado um wrapper coleção, como neste exemplo, ou um proxy. Você pode também, em semelhantes casos raros, verificar o estado atual de uma propriedade chamando `NHibernateUtil.IsInitialized()`. (Note que `Initialize()` não cascata para qualquer associado objetos.)

Outra solução para este problema é manter a sessão aberta até terminar a thread do aplicativo, assim você pode navegar o objeto gráfico sempre que você gosta e tem NHibernate automaticamente inicializar todos os preguiçosos referências. Este é um problema de concepção, aplicação e demarcação da transação, nós discutí-la novamente no capítulo

8 seção 8.1. No entanto, a sua primeira escolha deve ser o de buscar o gráfico completo exigido, em primeiro lugar, usando HQL ou consultas critérios, com uma estratégia sensata padrão e otimizado buscar nos metadados de mapeamento para todos os outros casos. NHibernate permite-lhe olhar para o SQL subjacente que ele envia para o banco de dados, por isso é possível ajustar recuperação de objeto se os problemas de desempenho são observados suspeita. Isto é discutido na próxima seção.

4.4.7 Ajuste de recuperação de objetos

Na maioria dos casos os aplicativos NHibernate terá um bom desempenho quando somes para buscar dados do banco de dados. No entanto, ocasionalmente, você pode observar que algumas áreas de sua application não estão executando bem como deveriam. Pode haver muitas razões para isso, então nós precisamos entender como analisar e ajustar o nosso Aplicações NHibernate para que funcionem de forma eficiente com o banco de dados. Vejamos os passos envolvidos, quando você está ajustando as operações objeto de recuperação na sua aplicação:

Permitir que o NHibernate log SQL, como descrito no capítulo 2, seção 2.4.2. Você também deve estar preparado para ler, compreender e avaliar consultas SQL e suas características de desempenho para o seu relacional específica modelo: Será que uma operação de junção única ser mais rápido do que duas seleciona? São todos os índices usados corretamente, eo que é

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

o cache hit relação dentro do banco de dados? Obtenha seu DBA para ajudá-lo com a avaliação de desempenho; só ele terá o conhecimento para decidir qual SQL plano de execução é o melhor.

Percorrer o caso de uso de aplicativos de caso de uso e observe quantas e declarações que SQL NHibernate executado. Um caso de uso pode ser uma única tela em sua aplicação web ou uma seqüência de diálogos do usuário.

Esta etapa também envolve a coleta os métodos do objeto de recuperação de você usar em cada caso de uso: andar o gráfico, recuperação por identificador, HQL e consultas critérios. Seu objetivo é reduzir o número (e complexidade) de Consultas SQL para cada caso de uso por meio do ajuste do padrão buscar estratégias de metadados.

Você pode encontrar dois problemas comuns:

Se as instruções SQL use as operações de associação que são demasiado complexos e lentos, definido `outer-join` false para

`<many-to-one>` associações (isto é ativado por padrão). Também tentar sintonia com o global `hibernate.max_fetch_depth` opção de configuração, mas tenha em mente que esta é melhor deixar em um valor entre 1 e 4.

Se muitas declarações SQL são executadas, use `lazy = "true"` para todos os mapeamentos de coleção, por padrão, NHibernate irá executar um adicional de imediato para buscar os elementos da coleção (que, se eles estão entidades, pode cascata mais no gráfico). Em casos raros, se tiver certeza, permitir que `outer-join = "true"` e desativar o carregamento lento para coleções particulares. Tenha em mente que apenas uma propriedade de coleção por

classe persistente pode ser buscado ansiosamente. Use a busca lote com valores entre 3 e 15 para continuar coleta de otimizar a busca, se a determinada unidade de trabalho envolve vários "da mesma" coleções ou se você está acessando uma árvore de pai e objetos filho.

Depois de definir uma nova estratégia de busca, execute novamente o caso de uso e verifique o SQL gerado novamente. Observe o SQL declarações, e vá para o caso de uso que vem.

Depois de otimizar todos os casos de uso, verificar cada uma de novo e ver se algum otimizações tinha efeitos colaterais para

os outros. Com alguma experiência, você será capaz de evitar quaisquer efeitos negativos e acertar da primeira vez.

Essa técnica de otimização não só é prático para o padrão buscar estratégias; você também pode usá-lo para sintonia e consultas HQL critérios, que podem ignorar e substituir o padrão fetching para casos de uso específico e unidades de trabalho. Discutimos runtime buscar no capítulo 8.

Nesta seção, nós começamos a pensar sobre problemas de desempenho, especialmente questões relacionadas com a associação

busca. É claro, a maneira mais rápida para buscar um gráfico de objetos é buscá-la a partir do cache na memória, como mostrado no próximo capítulo.

4.5 Resumo

Os aspectos dinâmicos do objeto / relacional incompatibilidade são tão importantes como o mais conhecido e melhor entendida problemas incompatibilidade estrutural. Neste capítulo, foram principalmente preocupado com o ciclo de vida objetos em relação ao mecanismo de persistência. Agora você entende os estados objeto definido por três NHibernate: persistente, individual, e transitórios. Objetos de transição entre estados, quando você invocar métodos de o `ISession` interface ou criar e remover referências a partir de um gráfico de instâncias já persistente. Este comportamento último é regida pelos estilos em cascata configurável, o modelo do NHibernate para persistência transitiva. Este modelo permite que você declare a cascata de operações (tais como a poupança ou supressão) numa base de associação, o que é mais poderoso e flexível do que o tradicional persistência por acessibilidade modelo. Seu objetivo é encontrar o melhor estilo em cascata para cada associação e, portanto, minimizar o número de chamadas gerenciador de persistência você tem que fazer ao armazenar objetos.

Recuperação de objetos do banco de dados é igualmente importante: Você pode andar o gráfico de objetos de domínio como acessar as propriedades e deixar transparente NHibernate buscar objetos. Você também pode carregar objetos pelo identificador,

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

escrever consultas arbitrário no HQL, ou criar uma representação orientada a objeto da sua consulta usando a consulta por

critérios API. Além disso, você pode usar SQL nativas em casos especiais.

A maioria destes métodos do objeto de recuperação de usar as estratégias que busca padrão definido nos metadados de mapeamento

(HQL ignora-los; consultas critérios pode substituí-los). A estratégia de busca correta minimiza o número de Instruções SQL que devem ser executados por preguiçosamente, ansiosamente, ou busca em lote objetos. Você a otimizar seu

NHibernate aplicação, analisando o SQL executadas em cada caso de uso e ajustar o padrão e tempo de execução buscar estratégias.

Em seguida, vamos explorar os temas estreitamente relacionados, do transações e cache.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

5

Transações de concorrência, e caching

Este capítulo aborda

- Operações de banco de dados e bloqueio
- Conversas de longa duração
- O NHibernate caches de primeiro e segundo nível
- O sistema de cache na prática com CaveatEmptor

Agora que você entende o básico de mapeamento objeto / relacional com NHibernate, vamos dar uma olhada uma das questões centrais no desenho das aplicações de banco de dados: gerenciamento de transações. Neste capítulo, examinamos como você usa NHibernate para gerenciar transações, como a concorrência é tratada e como o cache está relacionada com ambos os aspectos. Vamos olhar para a nossa aplicação exemplo.

Algumas funcionalidades do aplicativo requer que várias coisas diferentes ser feito em conjunto. Por exemplo, quando um leilão terminar, a nossa aplicação CaveatEmptor tem de realizar quatro tarefas diferentes:

7. Marca o lance vencedor (maior quantidade).
8. Carregue o vendedor o custo do leilão.
9. Cobrar o licitante vencedor o preço do lance vencedor.
10. Notificar o vendedor eo licitante vencedor.

O que acontece se não podemos cobrar o custo do leilão devido a uma falha no sistema externo de cartão de crédito?

Nossa

requisitos de negócio pode indicar que ou todas as ações listadas devem ter sucesso ou nenhum deve ser bem sucedido.

Se assim for, nós

chamar esses passos coletivamente uma transação ou unidade de trabalho. Se apenas um passo falhar, toda a unidade de trabalho deve

falhar. Dizemos que a transação é atômica: Várias operações são agrupadas como uma única unidade indivisível.

Além disso, as transações permitem que múltiplos usuários trabalhem simultaneamente com os mesmos dados, sem comprometer a integridade e exatidão dos dados, uma transação em particular não deve ser visível e não devem influenciar outras transações executando concorrentemente. Várias estratégias diferentes são utilizados para implementar esse comportamento, que é chamado isolamento. Vamos explorá-los neste capítulo.

As transações são também disse que a exposição consistência e durabilidade. Consistência significa que qualquer transação

trabalha com um conjunto consistente de dados e deixa os dados em um estado consistente quando a transação for concluída.

Garante uma durabilidade que uma vez que uma transação for concluída, todas as alterações feitas durante essa transação se tornam

persistente e não são perdidos mesmo se o sistema falhar posteriormente. Atomicidade, consistência, isolamento e durabilidade

são conhecidas como os critérios ACID.

Começamos este capítulo com uma discussão de nível de sistema operações de banco de dados, onde o banco de dados

para favor, postar comentários ou correções para o fórum online em Autor

para o comportamento ACID. Vamos olhar para o ADO.NET e transações Enterprise Services APIs e ver

<http://www.manning-sandbox.com/forum.jspa?forumID=205>

como NHibernate, trabalhando como um cliente dessas APIs, é usado para controlar as operações de banco de dados.

Em um aplicativo online, transações de banco de dados deve ter expectativa de vida extremamente curto. Uma transação de banco de dados deve abranger um único lote de operações de banco de dados, intercalados com lógica de negócios. Certamente não vão interação com o usuário. Vamos aumentar sua compreensão das operações com a noção de uma longa transação do usuário chamado conversa, onde as operações de banco de dados ocorrem em vários lotes, alternando com o usuário interação. Existem várias maneiras de implementar as conversas em aplicações NHibernate, todos os quais são discutidos neste capítulo.

Finalmente, o tema da cache é muito mais intimamente relacionados com as transações do que pode parecer à primeira vista. Por exemplo, o cache nos permite manter os dados perto de onde ela é necessária, mas correndo o risco de ficar obsoleto ao longo do tempo. Estratégias de caching, portanto, precisa ser equilibrada para também permitir consistente e durável transações. Na segunda metade deste capítulo, armado com uma compreensão das transações, exploramos NHibernate arquitetura de cache sofisticados. Você vai aprender quais os dados são um bom candidato para cache e como lidar com a simultaneidade do cache. Vamos então habilitar o cache na aplicação CaveatEmptor.

Vamos começar com o básico e ver como funcionam as transações no nível mais baixo, o banco de dados.

5.1 banco de dados de transações Compreensão

Bases de dados implementam a noção de uma unidade de trabalho como um transação (Às vezes chamado de sistema transação).

Um banco de dados de grupos de transações de acesso a dados de operações. A transação é garantida ao fim de uma das duas maneiras: ou é cometido ou revertida. Assim, as operações de banco de dados são sempre verdadeiramente atômica. Na figura 5.1, você pode ver isso graficamente.

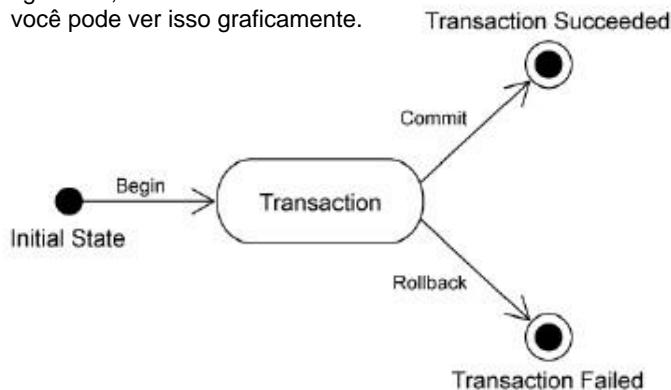


Figura 5.1 Estados de sistema durante uma transação

Se as operações de banco de dados diversos deve ser executado dentro de uma transação, você deve marcar os limites de a unidade de trabalho. Você deve iniciar a transação e, em algum momento, confirmar as alterações. Se ocorrer um erro (Seja durante a execução de operações ou quando cometer as alterações), você tem que reverter a transação para deixar os dados em um estado consistente. Isto é conhecido como a demarcação da transação, e (dependendo da API uso) que envolve, mais ou menos intervenção manual.

Você pode já ter experiência com duas interfaces de operação de manipulação de programação: o ADO.NET API e COM + serviço de processamento automático de transações.

5.1.1 ADO.NET e Enterprise Services / COM + transações

Sem Enterprise Services, o ADO.NET não! Indicador não definido. API é usado para marcar transação limites. Você iniciar uma transação chamando `BeginTransaction()` em um ADO.NET conexão e acabar com ela chamando `Commit()`. Você pode, a qualquer momento, forçar uma reversão imediata chamando `Rollback()`. (Easy, huh?)

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Em um sistema que armazena dados em vários bancos de dados, uma unidade especial de trabalho pode envolver o acesso a mais de um armazenamento de dados. Neste caso, você não pode conseguir atomicidade usando ADO.NET sozinho. Você precisa de uma transação gerente com suporte para transações distribuídas (duas fases). Você se comunica com a transação gerente usando o COM + serviço de processamento automático de transações.

Com o Enterprise Services, o serviço de processamento automático de transações é usado não apenas para sistemas distribuídos transações, mas também para declarativa de processamento de transações características. Declarativa de processamento de transações permite você evitar chamadas transação explícita demarcação em seu código-fonte do aplicativo, mas sim de transação, demarcação é controlado por atributos de transação. O atributo de transação declarativa especifica como um objeto participa de uma transação, e é configurado através de programação.

Não estamos interessados nos detalhes de ADO.NET direta ou demarcação de transação Enterprise Services. Você estará usando essas APIs em sua maioria de forma indireta. Capítulo 10, secção 10.3 explica como fazer e NHibernate Transações Enterprise Services trabalham juntos.

NHibernate se comunica com o banco de dados através de um ADO.NETError! Indicador não definido. IDbConnection e fornece a sua própria camada de abstração, escondendo a API de transações subjacentes. Utilização Enterprise Services não requer qualquer mudança na configuração do NHibernate.

Gerenciamento de transações é exposto ao desenvolvedor da aplicação através do NHibernate ITransaction interface. Você não é obrigado a usar esta API-NHibernate permite controlar ADO.NET operações diretamente. Nós não discutir esta opção como está desanimado, então ao invés, vamos agora concentrar-se na API ITransaction e seu uso.

5.1.2 O NHibernate ITransaction API

O ITransaction interface fornece métodos para declarar os limites de uma transação de banco de dados. Ver listagem 5.1 para um exemplo do uso básico do ITransaction.

Listagem 5.1 usando o NHibernate EuTransação API

```
usando (sessão ISession sessions.OpenSession = ()  
    usando (session.BeginTransaction ()) {  
        ConcludeAuction ();  
        session.Transaction.Commit ();  
    }
```

A chamada para session.BeginTransaction () marca o início de uma transação de banco de dados. Isso inicia um ADO.NET transação na ADO.NET conexão. Com COM +, Você não precisa criar esta transação, o conexão será automaticamente inscrito na transação atual distribuídos ou você terá que fazê-lo manualmente, se você tiver ativado inscrição automática de transação off.

A chamada para Transaction.Commit () sincroniza o ISession estado com o banco de dados. NHibernate em seguida, confirma a transação subjacente se e somente se BeginTransaction () começou uma nova transação (Com + COM, você tem que votar a favor de completar a transação distribuída).

Se ConcludeAuction () uma exceção, o usando () declaração vai alienar a transação (aqui, isso significa fazer um rollback).

Preciso de uma transação para operações somente leitura?

Devido à nova modalidade de conexão lançamento do NHibernate 1.2, uma conexão de banco de dados é aberta e fechadas para cada transação. Enquanto você está executando uma única consulta, você pode deixar NHibernate gerenciar a transação.

É extremamente importante certificar-se que a sessão é fechada no final, a fim de garantir que a ADO.NET conexão é liberado e voltou para o pool de conexão. (Esta etapa é de responsabilidade do aplicativo.)

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Aqui está outra versão mostrando em detalhes que exceções podem ser geradas e como lidar com eles:
(Esta versão é mais complexa do que o apresentado no capítulo 2)

```
Sessão ISession sessions = OpenSession();
ITransaction tx = null;
try {
    tx = session.BeginTransaction();
    ConcludeAuction();

    tx.Commit();
} Catch (Exception e) {
    if (tx != null) {
        try {
            tx.Rollback();
        } Catch (HibernateException ele) {
            // Log ele
        }
    }
    throw;
}
Finally {
    try {
        Session.close();
    } Catch (HibernateException ele) {
        throw;
    }
}
```

Como você pode ver, até mesmo reverter um `ITransaction` e fechando a `ISession` pode lançar uma exceção. Você não quer usar este exemplo como um modelo em seu próprio aplicativo, uma vez que você prefere esconder a tratamento de exceção com o código de infra-estrutura genérica. Você pode, por exemplo, enrole a exceção lançada em seu próprio `InfrastructureException`. Discutimos essa questão de design do aplicativo em mais detalhe no capítulo 8, ponto 8.1, "Inside the camadas de uma aplicação NHibernate".

NOTA

Há um aspecto importante que você deve estar ciente: a `ISession` tem que ser imediatamente fechado e rejeitadas (não reutilizado) quando ocorre uma exceção. NHibernate não pode repetir as operações fracassaram. Este não é um problema na prática, porque as exceções de banco de dados são geralmente fatal (violações de restrição, por exemplo) e não existe um estado bem definido para continuar depois de uma transação que falhou. Uma aplicação em produção não deve lançar todas as exceções de banco de dados também. Você também deve estar ciente de que após conter uma transação, a sessão NHibernate substitui-lo por um novo transação. Isto significa que você deve manter uma referência para a transação que você está cometendo se você acha que você vai precisar dele mais tarde. Isto é necessário se você precisa chamar `transaction.WasCommitted()`. `session.Transaction.WasCommitted` sempre retornará falsa.

Notamos que a chamada para `Commit()` sincroniza o `ISession` estado com o banco de dados. Isso é chamado rubor, um processo que aciona automaticamente quando você usa o NHibernate `ITransaction` API.

5.1.3 Flushing a Sessão

O NHibernate `ISession` implementa transparente escrever para trás. Isto significa que as mudanças para o domínio modelo feito no âmbito de um `ISession` não são imediatamente propagadas para o banco de dados. Instad, NHibernate podem coalescer muitas mudanças em um número mínimo de pedidos de banco de dados, ajudando a minimizar o impacto da latência da rede.

Por exemplo, se uma única propriedade de um objeto é alterado por duas vezes no mesmo `ITransaction`, NHibernate só precisa executar um SQL **ATUALIZAÇÃO**.

Flushes NHibernate ocorrer somente nos seguintes horários:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Quando um `ITransaction` está comprometida

Às vezes, antes de uma consulta é executada

Quando o aplicativo chama `ISession.Flush()` explicitamente

Flushing o `ISession` estado para o banco de dados no final de uma transação de banco de dados é necessária para fazer as mudanças duráveis e é o caso comum. NHibernate não lavar antes de cada consulta. No entanto, se houver mudanças são mantidas na memória que poderiam afetar os resultados da consulta, NHibernate irá, por padrão, sincronizar em primeiro lugar.

Você pode controlar esse comportamento, definindo explicitamente o NHibernate `FlushMode` via à propriedade `session.FlushMode`. Os modos de descarga são as seguintes:

`FlushMode.Auto`-O padrão. Permite que o comportamento descrito.

`FlushMode.Commit`-Especifica que a sessão não será liberado antes da execução da consulta (que será liberadas somente no final da transação). Estar ciente de que esta configuração pode expô-lo a stale dados: as modificações feitas apenas objetos na memória podem entrar em conflito com os resultados da consulta.

`FlushMode.Nempre`-Permite que você especifique que somente chamadas explícitas para `Flush()` resultar em sincronização de

o estado da sessão com o banco de dados.

Nós não recomendamos que você altere essa configuração do padrão. É fornecido para permitir que o desempenho otimização em casos raros. Da mesma forma, a maioria das aplicações raramente precisará chamar `Flush()` explicitamente. Este

funcionalidade é útil quando você está trabalhando com gatilhos, misturando com NHibernate direta ADO.NET. Ou a trabalhar com buggy ADO.NET drivers. Você deve estar ciente da opção, mas não necessariamente olhar para fora para casos de uso.

Nós discutimos como NHibernate lida com ambas as transações e a lavagem de alterações no banco de dados. Outra responsabilidade importante do NHibernate é gerenciar conexões reais para o banco de dados. Discutimos esta próxima.

5.1.4 Entendendo liberação modos de conexão

Como um recurso valioso, a conexão do banco de dados devem ser mantidos abertos para o curto espaço de tempo possível. NHibernate é inteligente o suficiente para abri-lo apenas quando realmente necessário (abrir a sessão não abrir automaticamente a conexão). Desde NHibernate 1.2, também é possível definir quando o banco de dados conexão deve ser fechada.

Neste momento, existem duas opções disponíveis. Eles são definidos pela enumeração `NHibernate.ConnectionReleaseMode`:

`OnClose`-Este era o único modo disponível no NHibernate 1.0. Neste caso, a sessão libera o conexão quando ele é fechado.

`AfterTransaction`-Este é o modo padrão no NHibernate 1.2. A conexão é liberada tão logo como a transação é concluída.

Note que você pode usar o `Desconectar()` método `ISession` interface para forçar a liberação da conexão (sem fechar a sessão) e o `Reconectar()` dizer método para a sessão para obter um novo conexão quando necessário.

Obviamente, estes modos são ativados somente para conexões abertas por NHibernate. Se você abrir um conexão e enviá-lo para NHibernate, você também é responsável de fechar esta conexão.

In order to specify a mode, you must use the configuration parameter `hibernate.connection.release_mode`. Seu valor padrão (e recomendado) é `automático`. Seleciona o melhor modo, que atualmente é `AfterTransaction`. Os dois outros valores são `ON_CLOSE` e `after_transaction`.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Como NHibernate 1.2.0 tem alguns problemas, como lidar com APIs `System.Transactions`, Você tem que usar o `OnClose` modo de se descobrir que a sessão abre múltiplas conexões em uma única transação.

Agora que você compreende o uso básico de transações de banco de dados com o NHibernate `ITransaction` interface, vamos voltar nossa atenção mais de perto ao tema do acesso a dados em simultâneo.

Parece que você não deve se preocupar com transaction-isolation o termo implica que algo é ou não é isolado. Isso é enganoso. Completo isolamento de transações simultâneas é extremamente

caro em termos de escalabilidade da aplicação, de modo bases de dados oferecem vários graus de isolamento. Para a maioria aplicações, de isolamento da transação incompleta é aceitável. É importante entender o grau de isolamento que deve escolher para uma aplicação que utiliza NHibernate NHibernate e como se integra com a transação capacidades do banco de dados.

5.1.5 Entendendo os níveis de isolamento

Bases de dados (e outros sistemas transacionais) tentativa de assegurar isolamento da transação, o que significa que, a partir do ponto de vista de cada operação simultânea, parece que nenhuma outra transação estão em andamento.

Tradicionalmente, esta tem sido implementado usando locking. Uma transação pode colocar um bloqueio em um determinado item de dados, impedindo temporariamente o acesso a esse item por outras transações. Alguns bancos de dados modernos, tais como Oracle e PostgreSQL isolamento da transação implementar usando versão multi-controle de concorrência, o que é geralmente considerado mais escalável. Vamos discutir o isolamento assumindo um modelo de bloqueio (a maioria dos nossos observações também são aplicáveis a versão multi-simultaneidade).

Este debate é sobre as transações de banco de dados eo nível de isolamento fornecido pelo banco de dados. NHibernate não adiciona semântica adicional; ele usa o que estiver disponível com um banco de dados. Se você considerar os muitos anos de experiência que os fornecedores de banco de dados tiveram com a implementação de controle de concorrência, você vai ver claramente

a vantagem dessa abordagem. Sua parte, como um desenvolvedor de aplicativos NHibernate, é entender o capacidades do seu banco de dados e como alterar o comportamento de isolamento de banco de dados, se necessário,

Problemas de

isolamento cenário (e, por suas exigências de integridade de dados).

Primeiro, vamos olhar para vários fenômenos que rompem o isolamento de transações completo. O ANSI SQL norma define as

níveis de isolamento de transação padrão em termos de quais destes fenômenos são permitidas:

Perdeu update-Two simultaneamente operações de atualização de uma linha e, em seguida, anula a transação segundo, fazendo com que ambos os

alterações sejam perdidas. Isso ocorre em sistemas que não implementam qualquer bloqueio. As transações concorrentes

leitura-sua. One transação lê as alterações feitas por outra transação que ainda não tenha sido cometido.

Isto é muito perigoso, porque essas alterações podem ser revertidas mais tarde.

Irrepetível ler-A transação lê uma linha duas vezes e lê estado diferente de cada vez. Por exemplo, outra transação pode ter escrito para a linha, e comprometida, entre as duas leituras.

Segunda atualizações perdidas problema-A caso especial de uma leitura irrepetível. Imagine que dois concorrentes operações de tanto ler uma linha, se escreve a ela e se compromete, em seguida, o segundo escreve a ela e se compromete.

As mudanças feitas pelo primeiro escritor são perdidas. Este problema também é conhecido como Win Escreva Passado. A ler-A transação executa uma consulta duas vezes, eo segundo conjunto de resultados inclui linhas que não eram visíveis no primeiro conjunto de resultados. (Ele não precisa ser necessariamente exatamente a mesma consulta.) Esta situação

é causada por outra transação inserir novas linhas entre a execução das duas consultas.

Agora que você entende todas as coisas ruins que poderiam ocorrer, podemos definir os vários isolamento da transação níveis e ver quais os problemas que impedem.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Níveis de isolamento

Os níveis de isolamento padrão são definidos pelo ANSI SQL padrão e você usar esses níveis para declarar o seu desejado de isolamento da transação mais tarde:

Permite-leitura não confirmada leituras sujas, mas não perdeu atualizações. Uma transação não pode escrever para uma linha se outra transação não confirmada já escreveu para ele. Qualquer transação pode ler qualquer linha, no entanto. Este nível de isolamento pode ser implementado usando bloqueios de gravação exclusivo.

Read committed-Permite irrepetível, mas não lê leituras sujas. Isto pode ser conseguido usando momentânea compartilhada bloqueios de leitura e fechaduras de gravação exclusivo. Operações de leitura não bloqueiam outras transações de acesso a uma linha. No entanto, uma operação de escrita não confirmadas bloqueia todas as outras transações de acessar a linha.

Repetível leitura Permite nem irrepetível lê nem leituras sujas. Leituras fantasmas podem ocorrer. Este pode ser conseguido usando bloqueios de leitura compartilhada e bloqueios de gravação exclusivo. Operações de leitura do bloco de escrita transações (mas não as operações outra leitura) e as transações de escrita bloquear todas as outras transações. Fornecerá serializável o isolamento estrito transação. Ele emula execução serial da transação, como se transações fossem executadas uma após a outra, em série, ao invés de simultaneamente. Seriação pode não ser implementado usando apenas uma linha de nível de fechaduras; deve haver outro mecanismo que impede que um recém-linha inserida de tornar-se visível para uma transação que já foi executada uma consulta que retornaria a linha.

É bom saber como todos esses termos técnicos são definidos, mas como é que ajudar você a escolher um isolamento nível para a sua aplicação?

5.1.6 Escolhendo um nível de isolamento

Desenvolvedores (inclusive nós) são muitas vezes não tem certeza sobre qual o nível de isolamento da transação para usar em uma produção aplicação. Um elevado grau de isolamento irá prejudicar o desempenho de uma aplicação altamente concorrente. Isolamento insuficiente pode causar erros sutis em nossa aplicação que não pode ser reproduzida e de que nós nunca descobrir até que o sistema está trabalhando sob carga pesada no ambiente implantado.

Note-se que nos referimos caching e bloqueio otimista (Usando controle de versão) no seguinte explicação, dois conceitos explicados mais adiante neste capítulo. Você pode querer ignorar esta secção e voltar quando é hora de tomar a decisão para um nível de isolamento em sua aplicação. Escolher o nível de isolamento direito é, afinal, altamente dependente de seu cenário em particular. A discussão a seguir contém recomendações; nada é esculpido em pedra.

NHibernate se esforça para ser o mais transparente possível sobre a semântica transacional do banco de dados. No entanto, cache e bloqueio otimista afetar essas semântica. Então, o que é um banco de dados sensíveis nível de isolamento para escolher em uma aplicação NHibernate?

Primeiro, você elimina a leitura não confirmada nível de isolamento. É extremamente perigoso usar uma transação alterações não confirmadas em uma transação diferente. A reversão ou o fracasso de uma operação afetaria outros transações concorrentes. Reversão da primeira transação poderia trazer outras transações para baixo com ele, ou talvez até mesmo levá-los a deixar o banco de dados em um estado inconsistente. É possível que as alterações feitas por um transação que acaba sendo revertida poderia ser cometido de qualquer maneira, uma vez que poderiam ser lidos e, em seguida, propagadas por outra transação que é sucesso!

Segundo, a maioria das aplicações não precisam serializável isolamento (leituras fantasmas geralmente não são um problema), e este nível de isolamento tende a escalar mal. Poucas aplicações existentes usam o isolamento serializável na produção; em vez disso, eles usam locks pessimistas (ver secção 6.1.8, "Usando o travamento pessimista"), o que efetivamente força uma execução serializada de operações em determinadas situações.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Isto deixa-lhe uma escolha entre read committed e repetível ler. Vamos primeiro considerar leitura repetida. Este nível de isolamento elimina a possibilidade de que uma operação poderia substituir as alterações feitas por outro transação concorrente (o segundo perdeu atualizações problema) se todos os acesso aos dados é realizado em um único atômica

transação. Esta é uma questão importante, mas usando leitura repetida não é a única maneira de resolvê-lo.

Vamos supor que você está usando dados versionados, algo que NHibernate pode fazer por você automaticamente. O combinação de (obrigatório) cache sessão NHibernate primeiro nível e versões já lhe dá a maioria dos as características de isolamento de leitura repetível. Em particular, versionamento impede o problema de atualização segundo perdido, e

o cache de sessão de primeiro nível garante que o estado do instâncias persistentes carregado por uma transação é isolado das mudanças feitas por outras transações. Assim, isolamento de leitura confirmada por todas as transações do banco de dados

seria aceitável se você usa dados versionados.

Repetível ler fornece reproduzibilidade um pouco mais para conjuntos de resultados da consulta (somente para a duração da transação), mas desde leituras fantasmas ainda são possíveis, não há muito valor nisso. (Também não é comum para aplicações web para consultar a mesma tabela duas vezes em uma única transação.)

Você também tem que considerar o (opcional) de cache de segundo nível NHibernate. Ele pode fornecer o mesmo isolamento de transação como a operação de banco de dados subjacente, mas pode enfraquecer ainda mais o isolamento. Se você está

fortemente usando uma estratégia de simultaneidade de cache para o cache de segundo nível que não preserva a leitura repetida

semântica (por exemplo, a leitura e escrita e especialmente o não seja estrita-estratégias de leitura e escrita, tanto discutido mais tarde

neste capítulo), a escolha de um nível de isolamento padrão é fácil: você não pode alcançar repetível ler mesmo, então não há nenhum ponto abrandar o banco de dados. Por outro lado, você pode não estar usando o cache de segundo nível para as classes de crítica, ou você pode estar usando um cache totalmente transacional que fornece isolamento de leitura repetível.

Você deve usar leitura repetida neste caso? Você pode se quiser, mas é provavelmente não vale o desempenho custo.

Definir o nível de isolamento da transação permite que você escolha uma boa estratégia de bloqueio padrão para todos os seus

operações de banco de dados. Como você define o nível de isolamento?

5.1.7 Definir um nível de Isolamento

Cada ADO.NET conexão com um banco de dados utiliza o nível de isolamento padrão do banco de dados, o hábito de ler cometido ou repetível ler. Esse padrão pode ser alterada na configuração de banco de dados. Você também pode definir a transação isolamento para ADO.NET. Indicador não definido. conexões usando uma opção de configuração NHibernate:

<Adicionar

```
key = "hibernate.connection.isolation"  
value = "ReadCommitted"
```

/>

NHibernate irá definir este nível de isolamento em todos os ADO.NET conexão obtido a partir de um pool de conexão antes de iniciar uma transação. Alguns dos valores sensíveis para esta opção são as seguintes (você também pode encontrá-los em `System.Data.IsolationLevel`):

ReadUncommitted-Read isolamento não confirmadas

ReadCommitted-Read isolamento confirmado

RepeatableRead-repetível de isolamento de leitura

Serializable-Serializable isolamento

Note-se que nunca NHibernate mudanças o nível de isolamento de ligações obtidas a partir de uma fonte de dados fornecidos pelo

COM+. Você pode mudar o isolamento padrão usando o `Isolamento` propriedade da `System.EnterpriseServices.TransactionAttribute`.

Até agora, nós introduzimos as questões que cercam o isolamento de transação, os níveis de isolamento disponíveis, e como escolher o correto para sua aplicação. Como você pode ver, definir o nível de isolamento é uma opção global

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

que afeta todas as conexões e transações. De tempos em tempos, é útil para especificar um bloqueio mais restritivo para uma transação particular. NHibernate permite que você especifique explicitamente o uso de uma pessimista bloqueio.

5.1.8 Usando o travamento pessimista

Bloqueio é um mecanismo que impede o acesso simultâneo a um item específico de dados. Quando uma transação mantém um bloqueio sobre um item, nenhuma transação concorrente pode ler e / ou modificar este item. Um bloqueio pode ser apenas um bloqueio momentâneo, realizada enquanto o item está sendo lido, ou pode ser realizada até a conclusão da transação. Abloqueio pessimista é um bloqueio que é adquirida quando um item de dados é lido e que é mantido até que a transação conclusão.

Em modo de leitura confirmada (o nosso nível de isolamento de transação preferido), nunca a base de dados adquire pessimista fechaduras a menos que explicitamente solicitada pelo aplicativo. Normalmente, os bloqueios pessimistas não são os mais escalável abordagem para a concorrência. No entanto, em certas circunstâncias especiais, podem ser usados para prevenir banco de dados impasses nível, o que resulta em fracasso da transação. Alguns bancos de dados (Oracle, MySQL e PostgreSQL, para exemplo, mas não SQL Server) fornecer o SQL **SELECT ... FOR UPDATE** sintaxe para permitir o uso de explícita locks pessimistas. Você pode verificar o NHibernate **Dialeto**s para descobrir se o seu banco de dados suporta esse recurso.

Se o seu banco de dados não é suportado, NHibernate irá sempre executar um normal **SELECIONE** sem a **FOR UPDATE** cláusula.

O NHibernate **LockMode** classe permite que você solicite um bloqueio pessimista sobre um determinado item. Além disso, você pode usar o **LockMode** para forçar NHibernate para contornar a camada de cache ou para executar uma verificação de versão simples.

Você verá o benefício dessas operações quando discutimos versionamento e cache.

Monte uma transação com o seguinte código:
Categoría cat = session.get <category>(catid, LockMode.Upgrade);
cat.Name = "Novo Nome";
tx.Commit();

É possível fazer esta transação use aa bloqueio pessimista da seguinte forma:

```
ITransaction tx = session.BeginTransaction();
Categoria cat = session.get <category>(catid, LockMode.Upgrade);
cat.Name = "Novo Nome";
tx.Commit();
```

Com **LockMode.Upgrade**, NHibernate irá carregar o **Categoria** usando um **SELECT ... FOR UPDATE**, Assim trancando a linhas recuperadas no banco de dados até que eles estão liberados quando a transação termina.

NHibernate define vários modos de bloqueio:

LockMode.NONE-Não vá ao banco de dados, a menos que o objeto não está em um cache.

LockMode.READ-Bypass ambos os níveis de cache, e executar uma verificação de versão para verificar se o objeto na memória é a mesma versão que existe atualmente no banco de dados.

LockMode.Upgrade-Bypass ambos os níveis de cache, fazer uma verificação de versão (se aplicável), e obter um banco de dados de nível de bloqueio de atualização pessimista, se isso é suportado.

LockMode.UpgradeNoWait-O mesmo que **UPGRADE**, Mas usar um **SELECT ... FOR UPDATE NOWAIT**, se isso é suportado. Isso desativa esperando releases bloqueio simultâneo, assim, lançar uma exceção de bloqueio imediatamente se o bloqueio não pode ser obtida.

LockMode.WRITE- O bloqueio é obtido automaticamente quando NHibernate wirtes a uma linha na transação atual (este é um modo interno, você não pode especificá-lo explicitamente).

Por padrão, **Get ()** uso **LockMode.NONE**. **LockMode.READ** é mais útil com **ISession.Lock ()** e um objeto distante. Por exemplo:

```
Item item = ... ;
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
Licitacao Licitacao = new ();
item.AddBid (bid);
...
ITransaction tx = session.BeginTransaction ();
Session.lock (item, LockMode.READ);
tx.Commit ();
```

Este código executa uma verificação de versão no destacada **Item** exemplo, para verificar se a linha do banco de dados não foi atualizada por outra transação, uma vez que foi recuperado. Se ele foi atualizado, um **StaleObjectStateException** é lançada.

Por trás da cena, NHibernate executa um **SELEÇÃO** para se certificar de que há uma fileira de banco de dados com o identificador (e versão, se houver) do objeto destacado. Ele não verifica todas as colunas. Esta não é uma problema quando a versão está presente, porque é sempre atualizado quando persistir o objeto usando NHibernate. Se, por alguma razão, você ignorar NHibernate e usar ADO.NET, não se esqueça de atualizar a versão.

Especificando uma explícita **LockMode** que não **LockMode.NONE**, Você força NHibernate para contornar os níveis de cache e percorrer todo o caminho ao banco de dados. Nós achamos que a maioria dos caching tempo é mais útil de bloqueio pessimista, então não use uma explícita **LockMode** a não ser que realmente precisam. Nossa conselho é que se você tem um DBA profissional em seu projeto e deixar o DBA decidir quais as transações requerem pessimista bloqueio uma vez que o aplicativo está instalado e funcionando. Esta decisão deve depender de detalhes sutis da interações entre transações diferentes e não pode ser adivinhada na frente.

Vamos considerar um outro aspecto do acesso a dados em simultâneo. Nós achamos que a maioria dos desenvolvedores .NET estão familiarizados com a noção de uma transação de banco de dados e é isso que eles costumam dizer por transação. Neste livro, nós considerar que este é um de grão fino transação, mas também consideramos uma noção mais grosseiro. Nossa grossa transações grained corresponderá ao que o usuário da aplicação considera uma única unidade de trabalho. Por que isso deve ser diferente do que a transação de grão fino?

O banco de dados isola os efeitos das transações concorrentes. Ele deve aparecer com a aplicação que cada transação é a operação apenas acessando o banco de dados (mesmo quando não é). O isolamento é caro. O banco de dados deve alocar recursos significativos para cada transação para a duração do transação. Em particular, como já dissemos, muitos bancos de dados bloquear linhas que foram lidos ou atualizados por uma transação, impedindo o acesso por qualquer outra transação, até que a primeira transação concluída. Em alta sistemas concorrentes com centenas ou milhares de atualizações por segundo, esses bloqueios podem impedir que a escalabilidade se eles são mantidos por mais tempo do que o absolutamente necessário. Por esta razão, você não deve segurar a transação (Ou até mesmo a conexão ADO.NET) aberta enquanto aguardando entrada do usuário. Se um usuário leva alguns minutos para entrar dados em um formulário, enquanto o banco de dados é o bloqueio de recursos, em seguida, outras operações podem ser bloqueados para que todo duração! Tudo isso, claro, também se aplica a um NHibernate **ITransaction**. Já que é apenas um adaptador para o subjacente mecanismo de operação de banco de dados.

Se você deseja manipular usuário longo "tempo para pensar" e ainda aproveitando o ACID atributos de transações, transações de banco de dados simples não são suficientes. Você precisa de um novo conceito, de longa duração usuário transações também conhecido como conversa.

5.2 Trabalhar com conversas

Processos de negócios, que possam ser considerados uma única unidade de trabalho do ponto de vista do usuário, necessariamente abrangem várias solicitações de cliente do usuário. Isto é especialmente verdadeiro quando um usuário faz uma decisão de atualização dados sobre a base do estado atual dos dados.

Em um exemplo extremo, suponha que você coletar dados digitados pelo usuário em múltiplas telas, talvez usando assistente de estilo de navegação passo a passo. Você deve ler e escrever itens relacionados de dados em vários pedidos (daí diversas transações) até que o usuário clicar em Concluir na última tela. Ao longo deste processo, os dados

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

devem ser coerentes e o usuário deve ser informado de qualquer alteração aos dados feita por qualquer concorrente transação. Nós chamamos esse conceito de transação granulação grossa um conversa, uma noção mais ampla da unidade de trabalho.

Vamos agora reafirmar esta definição mais precisa. A maioria dos aplicativos .NET incluem vários exemplos de seguinte tipo de funcionalidade:

Os dados são recuperados e exibidos na tela, exigindo que o primeiro banco de dados de transação como os dados são lidos.

11. O usuário tem a oportunidade de ver e modificar os dados em seu próprio tempo. (Claro, não transação precisa aqui).

12. As modificações são feitas persistentes, o que exige uma operação de banco de dados segundo como os dados são gravados.

Em aplicações mais complexas, pode haver várias interações com o usuário antes de um determinado processo de negócio é completa. Isso leva à noção de uma conversa (por vezes chamado de transação longa, transação do usuário, operação de aplicação ou transação comercial). Nós preferimos a conversa termos ou usuário transação, uma vez que estes são menos vagas e enfatizar o aspecto de transações do ponto de vista do usuário.

Durante estas transações longas baseados no usuário, você não pode confiar no banco de dados para reforçar o isolamento ou atomicidade

de conversas simultâneas. Portanto, o isolamento torna-se algo que seu aplicativo precisa lidar com explicitamente - e podem até mesmo exigir pegar entrada do usuário.

Agora, vamos discutir conversa com um exemplo.

Em nossa aplicação CaveatEmptor, tanto o usuário que postou um comentário e qualquer administrador do sistema pode

abrir uma tela Editar Comentário para apagar ou editar o texto de um comentário. Suponha que dois administradores diferentes

abrir a tela de edição para ver o mesmo comentário, ao mesmo tempo. Tanto editar o texto do comentário e apresentar as suas

alterações. Como podemos lidar com isso? Há três estratégias:

Último commit ganha- Ambas as atualizações são salvas no banco de dados, mas o último sobrescreve o primeiro. Nenhum erro

mensagem é mostrada a ninguém, e é a primeira atualização silenciosamente perdido para sempre.

Comprômeter- O primeiro ganha primeira atualização é salvo, mas quando tenta segundo usuário para salvar suas alterações que

receber uma mensagem de erro dizendo "suas atualizações foram perdidas porque alguém atualizou o registro, enquanto

você estava editando-o ". O usuário deve iniciar suas edições de novo e espero que eles tenham mais sorte na próxima vez que

~~Mesmo que a alteração é feita pelo mesmo usuário, a alteração é perdida se o usuário tenta salvar a alteração~~
~~depois de que a alteração é feita por outro usuário. A alteração é perdida se o usuário tenta salvar a alteração~~
~~depois de que a alteração é feita por outro usuário. A alteração é perdida se o usuário tenta salvar a alteração~~

A primeira opção, vence última confirmação, é problemática; o segundo usuário substitui as alterações do primeiro usuário sem ver as alterações feitas pelo primeiro usuário ou até mesmo sabendo que eles existiram. No nosso exemplo, este provavelmente não teria importância, mas seria inaceitável em muitos cenários. A segunda e terceira opções são aceitáveis para a maioria dos cenários. Na prática não há uma solução única e melhor, você deve investigar a sua própria requisitos de negócio e selecione uma destas três opções.

Ao usar NHibernate, acontece a primeira opção por padrão e, portanto, não requer trabalho de sua parte. Você deve avaliar quais partes de sua aplicação, se houver, pode ir longe com este fácil, mas potencialmente abordagem perigosa.

Se você decidir que precisa de outras opções de bloqueio otimista, então você precisará adicionar código apropriado para sua aplicação. NHibernate podem ajudar a implementar isso usando versão gerenciada para bloqueio otimista,

também conhecido como Bloqueio Offline Otimista.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

5.2.1 Usando versões conseguiu

Versionamento conseguiu se baseia em uma versão de número que é incrementado ou um timestamp que é atualizado para o tempo atual, cada vez que um objeto é modificado. Note-se que não tem relação com TIMESTAMP do SQL Server coluna e que apresenta banco de dados orientado a simultaneidade não são suportados.

Para NHibernate versão gerenciados, devemos adicionar uma nova propriedade para o nosso Comentário classe e mapeá-la como um número de versão usando o <versão> tag. Primeiro, vamos olhar para as alterações ao Comentário classe com o atributos de mapeamento:

[Classe (Tabela = "COMENTARIOS")]
Comentário public class {

```
...
    versão int privado;
...
[Version (coluna = "VERSION")]
    Versão public int {
        get { versão return; }
        set { versão = valor; }
    }
}
```

Você também pode usar um âmbito público para o setter e métodos getter. Ao usar XML, o <versão> mapeamento da propriedade deve vir imediatamente após o mapeamento da propriedade de identificação no arquivo de mapeamento para a Comentário classe:

```
<class name="Comment" table="COMMENTS">
    <Id ... >
    </Id>
    <version name="Version" column="VERSION" />
    ...
</Class>
```

O número da versão é apenas um contador de valor que não tem qualquer valor semântico útil. Algumas pessoas preferem

usar um timestamp em vez disso:

[Classe (Tabela = "COMENTARIOS")]

Comentário public class {

```
...
    lastUpdatedDatetime DateTime privado;
...
[Timestamp (coluna = "LAST_UPDATED")]
    pública DateTime LastUpdatedDatetime {
        get { return lastUpdatedDatetime; }
        set { lastUpdatedDatetime = valor; }
    }
}
```

Em teoria, uma hora é um pouco menos seguro, já que duas transações simultâneas pode carregar e atualizar o mesmo item todas no mesmo milésimo de segundo, na prática, é improvável que isso ocorra. No entanto, recomendamos que os novos projetos utilizam uma versão numérica e não um timestamp.

Você não precisa para definir o valor da versão ou a propriedade timestamp si mesmo; NHibernate irá inicializar o valor quando você salvar um ComentárioE incremento ou redefini-la sempre que o objeto é modificado.

FAQ

É a versão atualizada de um dos pais se uma criança é modificado? Por exemplo, se uma oferta única na coleção licitações de um Item é modificado, é o número da versão do Item também aumentou por um ou não? O resposta a perguntas que e semelhante é simples: NHibernate irá incrementar o número da versão sempre que um objeto é sujo. Isto inclui todas as propriedades sujo, sejam eles de valor único ou coleções. Pensar sobre a relação entre Item e Licitação: Se um Oferta é modificada, a versão do relacionadas Item não é incrementado. Se adicionar ou remover um Oferta da coleção de propostas, a versão do Item será atualizado. (É claro que nós faríamos Oferta uma classe imutável, uma vez que não faz sentido para modificar os lances.)

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Sempre que as atualizações NHibernate um comentário, ele usa a coluna de versão no SQL **ONDE**
cláusula
que MÍRIOS atualização do conjunto "texto do comentário New 'COMMENT_TEXT = VERSION = 3
onde COMMENT_ID = 123 e VERSION = 2

Se outra transação tinha atualizado o mesmo item, uma vez que foi lido pela transação atual, o
coluna não deve conter o valor 2, ea linha não seria atualizado. NHibernate iria verificar o número de linhas
devolvido pelo ADO.NET motorista-que neste caso seria o número de linhas atualizadas, zero e lançar um
StaleObjectStateException.

VERSÃO

Usando essa exceção, podemos mostrar ao usuário da transação segundo uma mensagem de erro ("Você foi
trabalhar com dados desatualizados porque outro usuário modificou! ") e deixar o primeiro commit ganhar.
Alternativamente, nós
pode capturar a exceção, fechar a sessão atual, e mostrar ao usuário segundo uma nova tela, permitindo ao usuário
manualmente mesclar as alterações entre as duas versões (usando uma nova sessão).

É possível desabilitar o incremento da versão quando uma propriedade específica está sujo. Para fazer isso, você deve
adicionar `optimistic-lock = "false"` para o mapeamento desta propriedade. Este recurso está disponível para as propriedades,
componentes e coleções (versão da entidade proprietária, não é incrementado).

Também é possível implementar o bloqueio otimista sem uma versão por escrito:

```
<Class ... optimistic-lock = "all">
```

Neste caso, NHibernate irá comparar os estados de todos os campos para descobrir se algum deles como alterado. Este
recurso será
só funciona para objetos persistentes, que não pode trabalhar para objetos destacados porque NHibernate não têm o seu
estado
quando eles foram carregados.

Você também pode escrever `<Class ... optimistic-lock = "sujo">` se você quiser NHibernate para permitir
modificações concorrentes, desde que elas não são feitas sobre as mesmas colunas. Isto irá permitir, por exemplo, um
administrador para alterar o nome de um cliente, enquanto outro muda a sua localização.

É possível evitar a execução de atualizações desnecessárias (que irá desencadear em atualização eventos, mesmo
quando
não foram feitas alterações em instâncias desconectadas), mapeando suas entidades usando `<Class ... select-
before-update = "true">`. NHibernate irá selecionar estas entidades e os comandos de atualização único problema para sujo
instâncias. Entretanto, esteja ciente das implicações deste recurso.

Como você pode ver, NHibernate torna fácil de usar gerido versão para implementar o bloqueio otimista.
Você pode usar o bloqueio otimista e pessimista em conjunto, ou você tem que tomar uma decisão para um?
E por que é chamado otimista?

Uma abordagem otimista sempre pressupõe que vai dar tudo certo e que as modificações de dados conflitantes
são raros. Em vez de ser pessimista e bloqueando o acesso simultâneo de dados imediatamente (e forçando a execução
a ser serializado), controle de simultaneidade otimista só bloco no final de uma unidade de trabalho e gerar um erro.

Ambas as estratégias têm o seu lugar e usa, é claro. Multi-usuário padrão para aplicações geralmente otimista
controle de concorrência e usar locks pessimistas quando apropriado. Note-se que a duração de um bloqueio pessimista
em

NHibernate é uma transação de banco de dados único! Isto significa que você não pode usar um bloqueio exclusivo para
bloquear concorrentes

acesso mais do que uma única transação. Consideramos isso uma coisa boa, porque a única solução
seria um bloqueio extremamente caro retido na memória (ou o chamado tabela de bloqueio no banco de dados) para o
duração de, por exemplo, uma conversa. Isso é quase sempre um gargalo de desempenho, a cada acesso a dados
envolve verificações de bloqueio adicional a um gerenciador de bloqueio sincronizado. Você pode, se for absolutamente
necessário em seu

aplicação em particular, implementar um bloqueio muito simples pessimista mesmo, usando NHibernate para gerenciar o
bloqueio

mesa. Padrões para isto podem ser encontradas no site NHibernate, no entanto, nós definitivamente não recomendo este
abordagem. Você tem que examinar cuidadosamente as implicações de desempenho deste caso excepcional.

Vamos voltar às conversas. Você já sabe o básico de controle de versão e conseguiu o bloqueio otimista.
Nos capítulos anteriores (e, anteriormente neste capítulo), temos falado sobre o NHibernate `ISession` como não sendo
o mesmo que uma transação. Na verdade, uma `ISession` tem um alcance flexível, e você pode usá-lo de diferentes maneiras com

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

banco de dados e conversas. Isto significa que o trabalho que você quer que seja.

granularidade de um **ISession**

é flexível, que pode ser qualquer unidade de

5.2.2 Granularidade de uma Sessão

Para entender como você pode usar o NHibernate **ISession**, Vamos considerar a sua relação com transações. Anteriormente, discutimos dois conceitos relacionados:

O escopo da identidade do objeto (ver secção 4.1.4)

A granularidade de banco de dados e conversas

O NHibernate **ISession** exemplo, define o âmbito da identidade de objeto. O NHibernate **ITransaction** exemplo, corresponde ao escopo de uma transação de banco de dados.

Qual é a relação entre um **ISession** e uma conversa? Vamos começar esta discussão com o máximo uso comum do **ISession**.

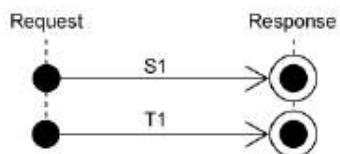


Figura 5.2 Usando 12:59 **ISession** e **ITransaction** por solicitação / resposta ciclo

Normalmente, abrimos um novo **ISession** para cada solicitação do cliente (por exemplo, uma solicitação do navegador web) e iniciar uma nova **ITransaction**. Depois de executar a lógica de negócios, nós nos comprometemos a transação e fechar o **ISession**, Antes de enviar a resposta para o cliente (ver figura 5.2).

A sessão (S1) ea transação (T1), portanto, têm a mesma granularidade. Se você não está trabalhar com o conceito de conversa, esta abordagem simples é tudo que você precisa na sua aplicação. Nós também gosto de chamar esta abordagem session-per-request.

Se você precisa de uma conversação de longa duração, você pode, graças a objetos destacados (e apoio do NHibernate para o bloqueio otimista como discutido na seção anterior), implementá-lo usando a mesma abordagem (ver figura 5.3).

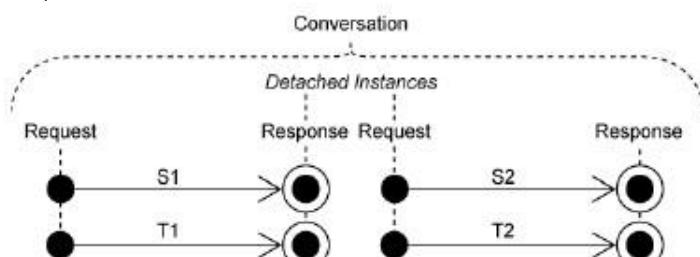


Figura 5.3 Implementação conversas com múltiplas **ISessions**, um para cada ciclo de solicitação / resposta

Suponha que a conversa se estende por dois solicitação do cliente / resposta ciclos, por exemplo, dois pedidos HTTP em uma aplicação web. Você poderia carregar os objetos interessantes em um primeiro **ISession** e depois anexá-los a uma nova **ISession** depois de terem sido modificados pelo usuário. NHibernate irá realizar automaticamente uma verificação de versão.

O tempo entre o (S1, T1) e (S2, T2) pode ser "longo", contanto que seu usuário precisa para fazer suas mudanças. Este abordagem também é conhecida como session-per-request-with-separada-objetos.

Alternativamente, você pode preferir usar um único **ISession** que abrange várias solicitações para implementar a sua conversa. Neste caso, você não precisa se preocupar com reattaching objetos destacados, uma vez que os objetos persistentes no contexto da uma longa **ISession** (Ver figura 5.4). Claro, ainda é NHibernate responsável por realizar o bloqueio otimista.

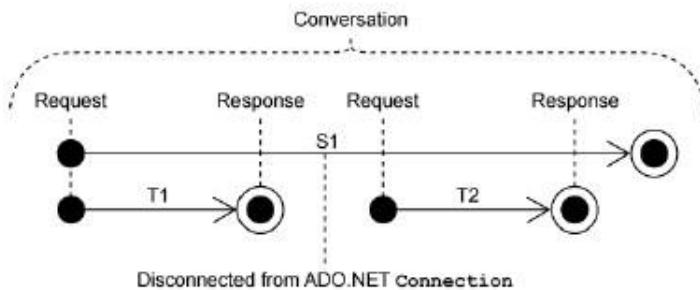


Figura 5.4 Implementação conversas com uma longa `ISession` usando desconexão

Uma conversa mantém uma referência para a sessão, embora a sessão pode ser serializada, se realmente necessário. A conexão ADO.NET subjacente tem de ser fechada, é claro, e uma nova ligação deverá ser obtida em um solicitação subsequente. Esta abordagem é conhecida como session-per-conversation ou longa sessão.

Normalmente, a sua primeira escolha deve ser o de manter o NHibernate `ISession` aberto não mais do que um único transação (sessão-por-requisição). Uma vez que a transação inicial é completa, quanto maior o sessão permanece aberta, maior a chance que detém dados desatualizados em seu cache de objetos persistentes (o sessão é o cache de primeiro nível obrigatório). Certamente, você nunca deve reutilizar uma única sessão por mais tempo do que leva para completar uma única conversa.

A questão de conversas e no âmbito do `ISession` é uma questão de design do aplicativo. Discutimos

estratégias de implementação de exemplos no capítulo 10, seção 10.2, "Implementando conversas."

Finalmente, há uma questão importante que você pode se preocupar. Se você trabalha com um banco de dados legado esquema, você provavelmente não pode adicionar versão ou colunas timestamp para o bloqueio otimista do NHibernate.

5.2.3 Outras formas de implementar o bloqueio otimista

Se você não tiver a versão ou colunas timestamp, NHibernate ainda pode realizar o bloqueio otimista, mas apenas para objetos que são recuperados e modificados no mesmo `ISession`. Se você precisa de bloqueio otimista para a destacadada objetos, você preciso use um número de versão ou timestamp.

Esta implementação alternativa de bloqueio otimista verifica o estado atual banco de dados contra a valores sem modificações das propriedades persistentes no momento que o objeto foi recuperado (ou a última vez que a sessão foi

liberados). Você pode ativar essa funcionalidade, definindo o `optimistic-lock` atributo no mapeamento de classe:

```
<class name="Comment" table="COMMENT" optimistic-lock="all">
    <id ..../>
    ...
</Class>
```

Agora, NHibernate irá incluir todas as propriedades no `ONDE` cláusula:

```
COMENTÁRIOS atualização do conjunto 'Novo texto' COMMENT_TEXT =
onde COMMENT_ID = 123
andCOMMENT_TEXT =
'Texto Antigo' andCOMMENT_TEXT =
andRATING = 5
andITEM_ID = 3
andFROM_USER_ID = 45
```

Alternativamente, NHibernate irá incluir apenas as propriedades modificadas (apenas `COMMENT_TEXT`, Neste exemplo) se você definir `optimistic-lock = "sujo"`. (Note que esta definição também exige que você para definir o mapeamento de classe para `dynamic-update = "true"`.)

Nós não recomendamos essa abordagem, é mais lento, mais complexo e menos confiáveis do que os números de versão e não funciona se a sua conversa se estende por várias sessões (que é o caso, se você estiver usando objetos detached).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note que, ao usar `optimistic-lock = "sujo"`, Duas conversas simultâneas podem atualizar o mesmo linha, desde que a mudança colunas diferentes, eo resultado pode ser desastroso para o usuário final. Então, se você tiver para usar este recurso, faça-o com cautela.

Vamos agora novamente mudar de perspectiva e considerar um novo aspecto do NHibernate. Já mencionamos o estreita relação entre as transações e armazenamento em cache na introdução deste capítulo. Os fundamentos da transações e bloqueio, e também os conceitos de granularidade da sessão, são de importância central quando consideramos cache de dados na camada de aplicativo.

5,3 Caching teoria e prática

A principal justificativa para a nossa afirmação de que os aplicativos usando um objeto / relacional camada de persistência são esperados para outperform aplicações construídas usando direto ADO.NET é o potencial para o cache. Embora nós vamos discutir apaixonadamente que a maioria das aplicações deve ser projetado de modo que é possível obter um desempenho aceitável sem o uso de um cache, não há dúvida de que, para alguns tipos de aplicações, especialmente com mais leitura aplicações ou aplicações que manter metadados significativa no banco de dados em cache pode ter um enorme impacto no desempenho.

Nós começamos nossa exploração de cache com algumas informações de fundo. Isso inclui uma explicação da o armazenamento em cache e escopos diferentes de identidade e do impacto da cache no isolamento da transação. Esta informação e estas regras podem ser aplicadas a cache, em geral, não são válidas apenas para aplicações NHibernate. Este discussão dá-lhe a fundo para entender por que o sistema de cache NHibernate é como é. Vamos então introduzir o sistema de cache NHibernate e mostrar a você como ativar, ajustar e gerenciar a primeira e segunda cache de nível NHibernate. Recomendamos que você estudar cuidadosamente os fundamentos estabelecidos nesta seção antes de começar a usar o cache. Sem o básico, você pode executar rapidamente em hard-to-debug concorrência problemas e risco a integridade de seus dados.

A cache mantém uma representação de cerca banco de dados atual estado da aplicação, seja na memória ou em disco da máquina servidor. Portanto, o cache é essencialmente apenas uma cópia local dos dados. Ela se situa entre sua aplicação e banco de dados. A grande vantagem é que o aplicativo pode poupar tempo por não ir à tempo de banco de dados todos os dados de que necessita. Isto pode ser muito vantajoso quando ele chegou a reduzir a pressão sobre uma movimentada servidor de banco de dados, e garantindo que os dados é servido à aplicação rapidamente. O cache pode ser usado para evitar uma banco de dados hit sempre

O aplicativo executa uma pesquisa por identificador (chave primária)

A camada de persistência resolve uma associação preguiçosamente

Também é possível armazenar em cache os resultados de consultas inteiro, por isso, se a mesma consulta é emitido repetidamente, todo o resultados são imediatamente disponíveis. Como você verá no capítulo 8, este recurso é utilizado com menos freqüência, mas o desempenho ganho de caching resultados da consulta pode ser impressionante em algumas situações.

Antes de olharmos para como cache de NHibernate funciona, vamos percorrer as opções de cache diferentes e ver como eles estão relacionados à identidade e simultaneidade.

5.3.1 estratégias de Caching e escopos

O cache é um conceito tão fundamental em objeto / relacional persistência que você não pode compreender o escalabilidade, desempenho ou transacional semântica de um ORM implementação sem primeiro saber o que caching estratégias que ele usa. Existem três tipos principais de cache:

Transação escopo-Attached para a unidade de trabalho atual, que pode ser uma transação de banco de dados reais ou

uma conversa. É válido e utilizado, desde que a unidade de trabalho é executado. Cada unidade de trabalho tem suas próprias

Por favor postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Processo de escopo Shared entre muitos (possivelmente simultâneas) unidades de trabalho ou para as transações. Isso significa que que os dados no cache do escopo do processo é acessado por transações executando concorrentemente, obviamente com implicações no isolamento da transação. A cache escopo do processo pode armazenar as instâncias persistentes ~~Cluster escopo. O tipo de cache de escopo determina se o cache é usado para persistência entre máquinas ou dentro de uma máquina.~~ cluster. Requer algum tipo de processo de comunicação remota para manter a consistência. Caching informação tem de ser replicada para todos os nós do cluster. Para muitos (não todos) aplicativos, o escopo do cluster cache é de valor duvidoso, já que ler e atualizar o cache pode ser apenas marginalmente mais rápido do que ir direto para o banco de dados. Há muitos parâmetros a ter em conta, para uma série de testes e afinações pode ser obrigado a decidir.

Camadas de persistência pode fornecer vários níveis de cache. Por exemplo, um cache miss (Um cache de pesquisa para um item que não está contido no cache) no escopo da transação pode ser seguido por uma busca no processo de escopo. Se isso falhar, indo para o banco de dados para os dados pode ser o último recurso. O tipo de cache usado por uma camada de persistência afeta o âmbito da identidade do objeto (a relação entre .NET identidade de objeto e identidade banco de dados).

Cache e identidade de objeto

Considere um cache de escopo de transação. Faz sentido se esse cache é usado também como o âmbito da identidade persistente objetos. Assim, se durante uma transação o aplicativo tenta recuperar o mesmo objeto duas vezes, a transação cache de escopo garante a pesquisas retornar a instância. mesmos NET. A cache escopo da transação é um bom ajuste para mecanismos de persistência que fornecem transação com escopo de identidade de objeto.

No caso do cache de escopo do processo, os objetos recuperados pode ser devolvido por valor. Em vez de armazenar e retornando casos, a cache contém tuplas de dados. Cada unidade de trabalho, em primeiro lugar recuperar uma cópia do estado a partir do cache (uma tupla) e depois usar isso para construir sua própria instância persistente na memória. Ao contrário da cache de escopo de transação discutido anteriormente, o escopo do cache e do âmbito da identidade do objeto não são mais o mesmo.

A cache âmbito do cluster sempre requer comunicação remota, uma vez que é provável que operam em vários máquinas. No caso de soluções POCO orientada persistência como NHibernate, os objetos são sempre passados remotamente por valor. Portanto, o cache de escopo de cluster trata de identidade, da mesma forma como o escopo do processo cache; cada um armazena cópias de dados, e passar esses dados para a aplicação para que eles possam criar as suas próprias casas a partir dele. Em termos NHibernate, ambos são caches de segundo nível, a principal diferença é que um cache de escopo de cluster pode ser distribuído entre vários computadores, se necessário.

Vamos discutir quais os cenários beneficiar de cache de segundo nível, quando ligar o processo (ou cluster) cache de escopo de segundo nível. Note que o primeiro nível de cache escopo da transação é sempre, e é obrigatória. Assim, as decisões a serem feitas são a possibilidade de usar o cache de segundo nível ou não, e para selecionar o tipo de usar,

Cache e de isolamento da transação

quais os dados que devem ser usados.

Um processo ou cache âmbito do cluster torna os dados recuperados do banco de dados em uma unidade de trabalho visível para outro unidade de trabalho. Essencialmente, o cache está permitindo que os dados em cache para ser compartilhado entre diferentes unidades de trabalho, múltiplas threads ou mesmo computadores. Isto pode ter alguns muito desagradável efeitos colaterais sobre isolamento da transação. Nós agora discutir algumas considerações críticas quando choosing usar um processo ou conjunto de cache de escopo.

Em primeiro lugar, se mais de um aplicativo está atualizando o banco de dados, então o cache de escopo do processo não deve ser usado, ou usado apenas para dados que raramente muda e pode ser seguramente refrescado por uma validade cache. Este tipo de dados ocorre freqüentemente em gerenciamento de conteúdo tipo de aplicações, mas raramente em aplicações financeiras.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Se você está projetando sua aplicação em escala ao longo de várias máquinas, você vai querer construir o seu aplicação para suportar o funcionamento em cluster. A cache escopo do processo não mantém a coerência entre os caches diferentes em diferentes máquinas do cluster. Para conseguir isso, você deve usar um escopo de cluster (Distribuído) cache em vez do cache escopo do processo.

Muitos. NET compartilhar o acesso a suas bases de dados com outras aplicações legadas. Neste caso, você não deve usar qualquer tipo de cache para além do cache obrigatória transação escopo. Não há nenhuma maneira de um cache sistema para saber quando a aplicação legada atualizados os dados compartilhados. Na verdade, é possível para implementar funcionalidade em nível de aplicativo para acionar uma invalidação do processo (ou cluster) de cache escopo quando alterações são feitas ao banco de dados, mas não sabemos de qualquer maneira padrão ou melhor para o conseguir. Certamente, ele irá nunca ser um recurso interno do NHibernate. Se você implementar essa solução, você provavelmente vai ser no seu próprio país, porque é extremamente específico para o ambiente e os produtos utilizados.

Depois de considerar não-exclusiva de acesso de dados, você deve estabelecer qual o nível de isolamento é necessário para o dados da aplicação. Nem todos os aspectos implementação de cache todos os níveis de isolamento da transação, e é fundamental para classes para armazenamento em cache são classes que descrevem o que é necessário. Vamos olhar os dados que a maioria dos benefícios de cache de um processo (ou cluster) de escopo Dados que mudam raramente

A ORM completa solução permitirá que você configure o cache de segundo nível separadamente para cada classe. Bom candidato Dados não críticos (por exemplo, de gestão de conteúdo de dados)

Dados que é local para a aplicação e não compartilhados

Candidatos ruim para o cache de segundo nível são

Dados que é atualizado com

frequência

Dados financeiros

Dados que são compartilhados com um aplicativo de legado

No entanto, estas não são as únicas regras que se aplicam normalmente. Muitos aplicativos têm um número de classes com o seguintes propriedades:

Um pequeno número de casos

Cada instância referenciado por várias instâncias de outra classe ou classes

Casos raramente (ou nunca) atualizado

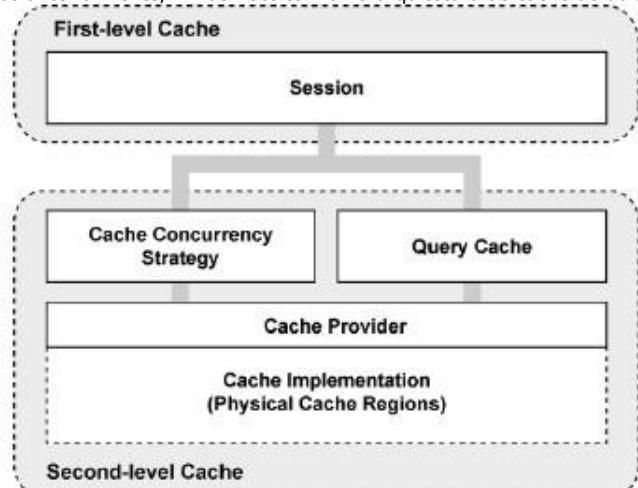
Este tipo de dados é muitas vezes chamado dados de referência. Dados de referência é um excelente candidato para o cache com um processo ou extensão cluster, e qualquer aplicação que utiliza dados de referência muito vai beneficiar muito se esses dados é armazenada em cache. Você permite que os dados a serem atualizados quando o tempo limite de cache expira.

Nós temos um quadro em forma de um sistema de dupla camada de cache nas seções anteriores, com um escopo de transação de primeiro nível e um processo de segundo nível ou cache opcional âmbito cluster. Isso é perto da cache NHibernate do sistema.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

5.3.2 A arquitetura de cache NHibernate

Como dissemos anteriormente, NHibernate tem uma arquitetura de cache de dois níveis. Os vários elementos deste sistema pode ser visto



na figura 5.5.

Figura 5.5 NHibernate arquitetura de cache de dois níveis

O cache de primeiro nível é o `ISession` em si. Uma sessão de vida corresponde a uma transação de banco de dados ou uma conversa (como explicado anteriormente neste capítulo). Consideramos o cache associado com o `ISession` para ser um cache de escopo de transação. O cache de primeiro nível é obrigatório e não pode ser desligado, mas também garante objeto identidade dentro de uma transação.

O cache de segundo nível em NHibernate é pluggable e pode ser delimitado para o processo ou cluster. Este é um cache do Estado (retornado por valor), não de instâncias persistentes. A estratégia de simultaneidade de cache define o detalhes da transação de isolamento para um determinado item de dados, enquanto que o provedor de cache representa o físico, implementação de cache real. Uso do cache de segundo nível é opcional e pode ser configurado em um per-classe e por associação de base.

NHibernate também implementa um cache para conjuntos de resultados de consulta que integra estreitamente com o segundo nível cache. Este é um recurso opcional. Discutimos a cache de consultas no capítulo 8, uma vez que seu uso está intimamente ligada à consulta real que está sendo executado.

Vamos começar com o uso do cache de primeiro nível, também chamado de cache de sessão.

Usando o cache de primeiro nível

O cache de sessão garante que quando o aplicativo solicita o mesmo objeto persistente duas vezes em um determinado sessão, ele recebe de volta o mesmo (idêntico). NET exemplo. Isso às vezes ajuda a evitar a banco de dados desnecessários tráfego. Mais importante, ele garante o seguinte:

A camada de persistência não é vulnerável a pilha estoura no caso de referências circulares em um gráfico de objetos.

Nunca pode haver representações conflitantes da linha mesmo banco de dados no final de um banco de dados transação. Não há, no máximo, um único objeto que representa todas as linhas do banco de dados. Todas as alterações feitas para que objeto pode ser escrito de forma segura para o banco de dados (liberados). As alterações feitas em uma determinada unidade de trabalho sempre são imediatamente visíveis para todos os outros códigos executados dentro daquela unidade de trabalho.

Você não tem que fazer nada especial para permitir que o cache de sessão. É sempre ligado e, pelos motivos apresentados, não pode ser desligado.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Sempre que você passar um objeto para `Save()`, `Update()` ou `SaveOrUpdate()`, o objeto é adicionado à sessão cache. Quando `Flush()` é chamado, o estado desse objeto será sincronizado com o banco de dados.

Se você não quer que esta sincronização ocorra, ou se você está processando um grande número de objetos e necessidade para gerenciar a memória de forma eficiente, você pode usar o `Evict()` método da `ISession` para remover o objeto e suas coleções a partir do cache de primeiro nível. Há vários cenários em que isso pode ser útil.

Gerenciamento do cache de primeiro nível

Considere esta pergunta freqüente: "Eu recebo um `OutOfMemoryException` quando eu tento carregar 100.000 objetos e manipular todos eles. Como posso fazer atualizações em massa com NHibernate?"

É nossa opinião que ORM não é adequado para atualização em massa (ou massa delete) operações. Se você tiver um caso de uso assim, uma estratégia diferente é quase sempre melhor: chamar um procedimento armazenado no banco de dados SQL ou uso direto

ATUALIZAÇÃO e **APAGAR** declarações para esse caso de uso particular. Não transferir todos os dados para a memória principal para um operação simples se ele pode ser realizado de forma mais eficiente pelo banco de dados. Se sua aplicação é principalmente massa

casos de uso, operação ORM não é a ferramenta certa para o trabalho!

Se você insistir em usar NHibernate mesmo para operações em massa, você pode imediatamente `Evict()` cada objeto depois de ter sido processada (durante a iteração através de um resultado de consulta), e, assim, evitar a exaustão de memória.

Completamente expulsar todos os objetos do cache de sessão, chamada `Session.Clear()`. Nós não estamos tentando convencê-lo que expulsar objetos do cache de primeiro nível é uma coisa ruim em geral, mas que os casos de bom uso são raros. Às vezes, usando projeção e uma consulta do relatório, como discutido no capítulo 8, seção 8.4.5, "Melhoria desempenho com consultas relatório, "pode ser uma solução melhor.

Note-se que o despejo, como salvar ou excluir as operações, pode ser automaticamente aplicada a objetos associados.

NHibernate irá despejar instâncias associadas da `ISession` se o atributo de mapeamento `cascata` está definido para todos ou `all-delete-orphan` para uma associação particular.

Quando a miss cache de primeiro nível ocorre, NHibernate tenta novamente com o cache de segundo nível se ele é habilitado para uma determinada classe ou associação.

O cache de segundo nível NHibernate

O cache de segundo nível NHibernate tem processo ou extensão cluster; todas as sessões compartilham o mesmo de segundo nível cache. O cache de segundo nível na verdade tem o escopo de uma `ISessionFactory`.

Instâncias persistentes são armazenados em um cache de segundo nível em um desmontado formulário. Pense em como a desmontagem

processo um pouco como a serialização (o algoritmo é muito, muito mais rápido do que a serialização .NET, no entanto).

A implementação interna deste cache de escopo do processo / cluster não é de muito interesse, mais importante é o uso correto do cache de políticas, que é, o cache de estratégias e provedores de cache física.

Diferentes tipos de dados requerem políticas de cache diferentes: a relação de leituras para escrever varia, o tamanho das tabelas do banco de dados varia, e algumas mesas são compartilhadas com outras aplicações externas. Assim, o cache de segundo nível é configurável na granularidade de uma classe individual ou função de coleta. Isto lhe permite, por exemplo, permitir que o cache de segundo nível para as classes de dados de referência e desativá-lo para as classes que representam os registros financeiros. O cache de segundo nível é ativado:

A estratégia de simultaneidade NHibernate

O cache de políticas de expiração (como vencimento ou prioridade)

Não beneficiar todas as classes de cache, por isso é extremamente importante ser capaz de desativar o cache de segundo nível.

Para repetir, o cache é geralmente útil somente para leitura em sua maioria classes. Se você tiver dados que são

atualizados com mais freqüência

do que é ler, não permitir que o cache de segundo nível, mesmo se todas as outras condições para o cache são verdadeiras!

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Além disso, o cache de segundo nível pode ser perigoso em sistemas que compartilham o banco de dados com a outra escrita

aplicações. Como explicamos nas seções anteriores, você deve exercer um juízo cuidado aqui.

O cache de segundo nível NHibernate é configurado em duas etapas. Primeiro, você tem que decidir qual simultaneidade

estratégia de usar. Depois disso, você configurar a expiração de cache e atributos de cache usando o provedor de cache. Built-in estratégias de concorrência

A estratégia de simultaneidade é um mediador, é responsável por armazenar itens de dados no cache e recuperá-los a partir do cache. Este é um papel importante, pois ele também define a semântica de isolamento de transação para que determinado item. Você vai ter que decidir, para cada classe persistente, que estratégia de simultaneidade de cache para usar, se você deseja ativar o cache de segundo nível.

Há três estratégias built-in de concorrência, o que representa diminuição dos níveis de rigor em termos de isolamento da transação:

read-write-Mantém read committed isolamento, usando um mecanismo timestamping. Ele está disponível apenas em

não agrupado ambientes. Use esta estratégia para leitura de dados, onde a maioria é crítico para evitar obsoleto dados em transações simultâneas, no caso raro de uma atualização.

não seja estrita-read-write-Faz nenhuma garantia de coerência entre o cache e banco de dados. Se houver uma possibilidade de acesso simultâneo à mesma entidade, você deve configurar uma expiração suficientemente curto

timeout. Caso contrário, você pode ler dados desatualizados no cache. Use esta estratégia se os dados raramente muda (muitos

horas, dias ou mesmo uma semana) e uma probabilidade pequena de dados obsoletos não é uma preocupação crítica. NHibernate

invalida o elemento em cache se a versão dele foi modificada para levá-lo para esta é uma operação assíncrona, apenas

sem bloqueio cache ou garantia de que os dados recuperados é a versão mais recente.

Note-se que com a diminuição da rigidez vem aumentando o desempenho. Você tem que avaliar cuidadosamente a desempenho de uma cache cluster com isolamento de transações completo antes de usá-lo em produção. Em muitos casos,

você pode ser melhor desativar o cache de segundo nível para uma classe particular se os dados obsoletos não é uma opção. Primeiro

candidatura do seu benchmark com o cache de segundo nível com deficiência. Então habilite-lo para as classes bom candidato,

um de cada vez, enquanto continuamente testar o desempenho do seu sistema e avaliar a simultaneidade estratégias.

It's possible to define your own concurrency strategy by implementing
[NHibernate.Cache.ICacheConcurrencyStrategy](#). Mas esta é uma tarefa relativamente difícil e só
apropriada para casos extremamente raros de otimização.

Seu próximo passo depois de considerar as estratégias de concorrência que você usará para suas classes candidato cache é

escolher um provedor de cache. O provedor é um plug-in, a implementação física de um sistema de cache.

Escolha de um provedor de cache

Por enquanto, NHibernate força você a escolher um provedor de cache único para todo o aplicativo. O seguinte fornecedores são liberados com o NHibernate:

Hashtable não se destina para uso em produção. Ele só caches na memória e podem ser definidos usando seu provedor:

[NHibernate.Cache.HashtableCacheProvider](#) disponíveis no interior [NHibernate.dll](#))

SysCache depende [System.Web.Caching.Cache](#) para a implementação subjacente para que você possa consultar a documentação do ASP.NET cache recurso para compreender como ele funciona. Sua NHibernate provedor é a classe [NHibernate.Caches.SysCache.SysCacheProvider](#) na biblioteca

[NHibernate.Caches.SysCache.dll](#). Este provedor só deve ser usado com ASP.NET Teia Aplicações.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Predomínio torna possível a utilização da base Bamboo.Prevalence implementação como cache provider. ItsNHibernateprovideris NHibernate.Caches.Prevalence.PrevalenceCacheProviderinthe library NHibernate.Caches.Prevalence.dll. Você também pode visitar Bamboo.Prevalence Website's: <http://bbooprevalence.sourceforge.net/>

Existem também alguns fornecedores de cache distribuído que você vai aprender sobre na próxima seção. E é fácil escrever um adaptador para outros produtos, implementando NHibernate.Cache.ICacheProvider.

Cada provedor de cache suporta Cache de Consultas do NHibernate e é compatível com todos os simultaneidade estratégia (somente leitura, não seja estrita-leitura e escrita, e read-write).

Criação de cache, portanto, envolve duas etapas:

Olhe para os arquivos de mapeamento de suas classes persistentes e decidir que estratégia de simultaneidade de cache você

gosto de usar para cada categoria e cada associação.

13. Permitir que o seu provedor de cache preferida na configuração do NHibernate e personalizar o provedor configurações específicas.

Vamos adicionar ao nosso cache CaveatEmptor Categoria e Item classes.

5.3.3 Caching na prática

Lembre-se que você não tem explicitamente habilitar o cache de primeiro nível. Então, vamos declarar as políticas de caching e configurar provedores de cache para o cache de segundo nível em nossa aplicação CaveatEmptor.

O Categoria tem um pequeno número de casos e raramente é atualizado, e as instâncias são compartilhados entre muitos usuários, por isso é um grande candidato para o uso do cache de segundo nível. Começamos por adicionar o elemento de mapeamento

obrigado a dizer NHibernate para armazenar em cache Categoria instâncias:

```
Class <table>CATEGORIA>
  public class Categoria {
    [Cache (-1, Uso = CacheUsage.ReadWrite)]
    [Id ....]
    Id longo público { ... }
  }
```

Note que, como o atributo [Discriminador], Você pode colocar [Cache] em qualquer propriedade de campo /, basta ter cuidado ao misturá-lo com outros atributos (aqui, usamos a posição -1, uma vez que deve vir antes).

Aqui está o mapeamento XML correspondente:

```
Class <
  name = "Categoria"
  table = "CATEGORIA">
  <cache usage="read-write"/>
  <Id .... >
</ Class>
```

O uso = "read-write" atributo diz NHibernate para usar uma estratégia de simultaneidade de leitura e escrita para o Categoria cache. NHibernate irá agora tentar o cache de segundo nível, sempre que navegar até uma Categoria ou quando carregar um Categoria pelo identificador.

Nós escolhemos leitura e escrita em vez de não seja estrita-leitura e escrita, Uma vez Categoria é altamente classe concorrente, compartilhada entre muitas transações simultâneas, e é claro que um isolamento de leitura confirmada nível é bom o suficiente. No entanto, não seja estrita-leitura e escrita provavelmente seria uma alternativa aceitável, desde uma pequena probabilidade de inconsistência entre o cache e banco de dados é aceitável (a hierarquia da categoria tem importância financeira pouco).

Este mapeamento foi o suficiente para dizer NHibernate para armazenar em cache todos os simples Categoria valores de propriedade, mas não o estado de entidades associadas ou coleções. Coleções de exigir os seus próprios <cache> elemento. Para o Itens coleta, usaremos um leitura e escrita estratégia de simultaneidade:

```
Class <
  name = "Categoria"
  table = "CATEGORIA">
```

Por favor, postar comentários ou correções para o fórum on-line em Autor <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```





```

Esse cache será usada ao enumerar a coleção **category.Items**, Por exemplo. Observe que a exclusão um item que existe em uma coleção no cache de segundo nível causará uma exceção, por isso, certifique-se de remover o item da coleção no cache antes de excluí-lo.

Agora, um cache coleção contém apenas os identificadores das instâncias item associado. Então, se nós exigimos o casos-se a ser armazenada em cache, devemos habilitar o cache do **Item** classe. Uma estratégia de leitura e escrita é especialmente apropriado aqui. Nossos usuários não querem tomar decisões (colocação de um **Oferta**), Com base possivelmente obsoletas de dados. Vamos dar um passo adiante e considerar o conjunto de **Ofertas**. A especial **Oferta** no **Lances** coleção é imutável, mas nós temos que mapear a coleção usando **leitura e escrita**, Uma vez que novas propostas podem ser feitas a qualquer momento (E é fundamental que sejamos imediatamente ciente de novas propostas):

```

Class <
    name = "Item"
    table = "ITEM">
    <cache usage="read-write"/>
    <Id ....
    <set name="Bids" lazy="true">
        <cache usage="read-write"/>
        <Chave ....
    </ Set>
</ Class>
```

Para o imutável **Oferta** classe, nós aplicamos uma estratégia de somente leitura:

```

Class <
    name = "Oferta"
    table = "BID">
    <cache usage="read-only"/>
    <Id ....
</ Class>
```

Em cache **Biddados** é válida por tempo indeterminado, pois os lances não são atualizados. Invalideção de cache não é necessário.

(As instâncias podem ser despejados pelo cache do provedor por exemplo, se o número máximo de objetos no cache é atingida.)

Usuário é um exemplo de uma classe que pode ser armazenado em cache com o **não seja estrita-leitura e escrita** estratégia, mas nós

não estão certos de que faz sentido para os usuários cache.

Vamos definir o fornecedor de cache, políticas de expiração, e propriedades físicas do nosso cache. Usamos esconderijo

regiões configurar a classe e cache coleção individualmente.

Compreendendo as regiões de cache

NHibernate mantém diferentes classes / coleções em cache diferentes regiões. A região é um cache chamado: uma alça por que você pode referenciar classes e coleções na configuração do provedor de cache e definir a expiração políticas aplicáveis a essa região.

O nome da região é o nome da classe, no caso de um cache de classe, ou o nome da classe junto com o nome da propriedade, no caso de um cache de coleção. **Categoria** instâncias são armazenadas em cache em uma região chamada

NHibernate.Auction.Category, E os **itens** coleção é armazenada em cache em uma região chamada **NHibernate.Auction.Category.Items**.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Você pode usar a propriedade de configuração NHibernate `hibernate.cache.region_prefix` para especificar um nome da raiz para uma determinada região `ISessionFactory`. Por exemplo, se o prefixo foi definido para `Node1`, `Categoria` seria armazenado em cache em uma região chamada `Node1.NHibernate.Auction.Category`. Esta configuração é útil se seu aplicativo inclui múltiplas `ISessionFactory` instâncias.

Agora que você sabe sobre regiões cache, vamos configurar as políticas de expiração para a `Categoria` cache. Primeiro vamos escolher um provedor de cache.

Criação de um provedor de cache local

Precisamos definir a propriedade que seleciona um provedor de cache:

```
key = "hibernate.cache.provider_class"
value = "NHibernate.Caches.SysCache.SysCacheProvider, NHibernate.Caches.SysCache"
/>>
```

Aqui, nós escolhemos `SysCache` como a nossa cache de segundo nível.

Agora, precisamos especificar as políticas de validade para as regiões cache. `SysCache` fornece dois parâmetros: um expiração valor que é o número de segundos de espera antes de expirar cada item (o valor padrão é de 300 segundos) e uma prioridade valor que é um custo numérico de expirar cada item, onde 1 é um custo baixo, 5 é o mais alta, e 3 é normal. Note que somente valores de 1 a 5 são válidos e se referem ao

`System.Web.Caching.CacheItemPriority` enumeração.

`SysCache` tem um manipulador da seção de arquivo de configuração para permitir a configuração de vencimentos diferentes e as prioridades para diferentes regiões. Veja como podemos configurar o `Categoria` classe:

```
<? Xml version = "1.0"?>
<configuration>
  <configSections>
    <Section>
      name = "syscache"
      type = "NHibernate.Caches.SysCache.SysCacheSectionHandler, NHibernate.Caches.SysCache"
    />
  </ConfigSections>
  <syscache>
    <cache region="Category" expiration="36000" priority="5" />
  </Syscache>
</ Configuration>
```

Há um pequeno número de categorias, e todos eles são compartilhados entre muitas transações concorrentes. Nós portanto, definir um valor de expiração alta (10 horas) e dar-lhe uma alta prioridade para que eles fiquem no cache como tempo possível.

`Ofertas`, por outro lado, são pequenos e imutáveis, mas existem muitos deles, por isso devemos configurar `SysCache` para gerenciar cuidadosamente o consumo de memória cache. Usamos tanto um valor de expiração baixo e um baixo prioridade:

```
<cache region="Bid" priority="1" expiration="300" />
```

O resultado é que os lances em cache são removidos do cache depois de 5 minutos ou se o cache estiver cheio (como eles têm a prioridade mais baixa).

Políticas de remoção de cache são ótimas, como você pode ver, específicas para os dados particulares e particulares aplicação. Você deve considerar muitos fatores externos, incluindo a memória disponível no servidor de aplicativos carga da máquina, prevista para a máquina do banco de dados, a latência da rede, a existência de aplicações legadas, e assim por diante.

Alguns desses fatores não pode ser conhecido em tempo de desenvolvimento, por isso muitas vezes você vai precisar testar a iterativamente

impacto no desempenho de diferentes configurações no ambiente de produção ou uma simulação dela.

Isto é especialmente verdadeiro em um cenário mais complexo, com um cache replicado implantado em um cluster de servidor máquinas.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Usando um cache distribuído

SysCache e Prevalência são provedores de cache excelente se seu aplicativo é implantado em uma única máquina. No entanto, aplicativos corporativos dando suporte a milhares de usuários simultâneos pode exigir mais de computação poder, ea escalabilidade do seu aplicativo pode ser crítico para o sucesso de seu projeto. Aplicações NHibernate são naturalmente escalável, isto é, NHibernate se comporta da mesma seja implantada em uma única máquina ou muito muitas máquinas. A única característica do NHibernate que devem ser configurados especificamente para operação de cluster é o cache de segundo nível. Com algumas mudanças para a nossa configuração de cache, somos capazes de usar um cache cluster do sistema.

Não é necessariamente errado usar um provedor de cache puramente local (non-cluster-aware) em um cluster. Alguns dados, especialmente dados imutáveis, ou dados que podem ser atualizados pelo cache do tempo limite doesn't exigem-cluster invalidação e pode ser seguramente armazenados em cache localmente, mesmo em um ambiente em cluster. Podemos ser capazes de ter cada nó do cluster usar uma instância local do SysCache, e escolher cuidadosamente suficientemente curto expiração valores.

No entanto, se você precisar de consistência do cache estrita em um ambiente em cluster, você deve usar um mais provedor de cache sofisticados. Alguns provedores de cache distribuído estão disponíveis para NHibernate. Você pode considerar os seguintes:

MemCache é liberado com NHibernate. Ele usa memcached, Um sistema de cache distribuído disponível sob Linux, assim você pode usá-lo com Mono (ou use VMWare Memcached aparelho no Windows). Para mais detalhes, visite: <http://www.danga.com/memcached/>. Seu provedor NHibernate é a classe `NHibernate.Caches.MemCache.MemCacheProvider` in the library `NHibernate.Caches.MemCdr.Dll`.

NCache é um provedor de cache comerciais distribuídos. Seu site é <http://www.alachisoft.com/ncache/>.

Nós não vamos cavar os detalhes destes prestadores de cache distribuído. Cache distribuído é um tema complexo, nós recomendamos que você leia alguns artigos sobre este tema e para testar estes fornecedores.

Note que alguns provedores de cache distribuído trabalhar apenas com algumas estratégias de concorrência cache. Um truque legal pode nos ajudar a evitar a verificar nossos arquivos de mapeamento um por um: em vez de colocar um `[Cache]` atributo na nossa entidades ou colocação de `<cache>` elementos em nossos arquivos de mapeamento, podemos centralizar a configuração de cache `<hibernate-configuration>` `<session-factory>`

```
<Class-cache>
    class = "NHibernate.Auction.Model.Bid, NHibernate.Auction"
    uso = "read-only" />
<Coleção cache>
    collection = "NHibernate.Auction.Model.Item.Bids"
    uso = "read-write" />
</Session-factory>
</Hibernate-configuration>
```

Nós habilitado somente leitura para o cache `Oferta` e de leitura e escrita para o cache `Lances` coleção neste exemplo. No entanto, há uma ressalva importante: no momento da redação deste texto, NHibernate vai correr em um conflito se também têm `<cache>` elementos no arquivo de mapeamento para `Item`. Nós, portanto, não pode usar a configuração global para substituir as configurações de arquivo de mapeamento. Recomendamos que você use a configuração de cache centralizado a partir do começar, especialmente se você não tem certeza de como seu aplicativo pode ser implantado. É também mais fácil para sintonizar de cache configurações com uma configuração centralizada.

Existe uma configuração opcional a considerar. Para os provedores de cache cluster, talvez seja melhor para definir o NHibernate opção de configuração `hibernate.cache.use_minimal_puts` para `verdadeiro`. Quando essa configuração é habilitado, NHibernate só adicionar um item ao cache após a verificação para garantir que o item não é já Por favor, postar comentários ou correções para o fórum on-line em Autor <http://www.manning-sandbox.com/forum.jspa?forumID=295>

em cache. Essa estratégia funciona melhor se o cache escreve (puts) são muito mais caros que o cache lê (fica). Este é o caso de um cache replicado em um cluster, mas não para um cache local (o padrão é **falso**, Otimizado para um cache local). Se você estiver usando um cluster ou um cache local, às vezes você precisa controlá-lo programática para fins de teste ou tuning.

Controlar o cache de segundo nível

NHibernate tem alguns métodos úteis que irão ajudá-lo a testar e ajustar o seu cache. Você pode se perguntar como desabilitar o cache de segundo nível completamente. NHibernate só vai carregar o provedor de cache e começar a usar o cache de segundo nível se você tiver quaisquer declarações de cache em seus arquivos de mapeamento ou XML arquivo de configuração. Se você comentá-las, o cache está desativado. Esta é uma outra boa razão para preferir configuração de cache centralizado em `hibernate.cfg.xml`.

Assim como o `ISession` fornece métodos para controlar o cache de primeiro nível de programação, assim que faz o `ISessionFactory` para o cache de segundo nível.

Você pode chamar `Evict()` para remover um elemento da cache, especificando a classe eo identificador objeto

valor:
`SessionFactory.Evict (typeof (Categoria), 123);`

Você também pode expulsar todos os elementos de uma determinada classe ou apenas despejar ~~um~~ ~~papel~~ ~~coleção~~ ~~particular~~:

Você raramente vai precisar destes mecanismos de controle e você deve usá-los com cuidado, pois eles não respeitam qualquer operação semântica de isolamento da estratégia de uso.

5.4 Resumo

Este capítulo foi dedicado a transações (de grão fino e coars-grained), simultaneidade e cache de dados.

Você aprendeu que para uma única unidade de trabalho, quer todas as operações devem ser completamente bem sucedida ou o unidade inteira de trabalho deve falhar (e as alterações feitas para o estado persistente deve ser revertida). Isto levou-nos à noção de uma transação e os atributos ACID. Uma transação é atômica, deixa de dados em um estado consistente, e é isolado de transações executando concorrentemente, e você tem a garantia de que os dados alterados por uma transação é durável.

Você usa dois conceitos de transação em aplicações NHibernate: transações de curto e longo execução conversas. Normalmente, você usa isolamento de leitura confirmada para transações de banco de dados, juntamente com

controle de concorrência otimista (versão e verificar timestamp) para longas conversas. NHibernate muito simplifica a implementação de conversas porque ele gerencia números de versão e timestamps para você.

Finalmente, discutimos os fundamentos da cache, e você aprendeu a usar o cache de forma eficaz em Aplicações NHibernate.

NHibernate fornece um sistema dual-layer de cache com um cache de objetos de primeiro nível (o `ISession`) E um pluggable segundo nível de cache de dados. O cache de primeiro nível é sempre ativo que é usado para resolver circular referências em seu gráfico do objeto e para otimizar o desempenho em uma única unidade de trabalho. O cache de segundo nível

por outro lado, é opcional e funciona melhor para as aulas de leitura em sua maioria candidato. Você pode configurar um não-

cache de segundo nível volátil para referência (somente leitura) de dados ou até mesmo um cache de segundo nível com a operação total

isolamento para dados críticos. No entanto, você tem que examinar cuidadosamente se o ganho de performance vale a pena o

esforço. O cache de segundo nível pode ser personalizado de grão fino, para cada classe persistente e até mesmo para cada

coleta e classe de associação. Usado corretamente e completamente testado, caching em NHibernate dá-lhe um nível de desempenho que é quase inatingível em uma camada de acesso mão-codificado dados.

Agora que nós cobrimos a maior parte dos aspectos fundamentais chave para aplicações NHibernate, podemos ir para mergulhar em algumas das capacidades mais avançadas de NHibernate. O próximo capítulo vai começar por discutir Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

alguns dos conceitos avançados de mapeamento NHibernate que irá permitir-lhe lidar com as mais exigentes requisitos de persistência.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

6

Avançados conceitos de mapeamento

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

No capítulo 3, introduzimos os recursos ORM mais importantes fornecidos pelo NHibernate incluindo classe básica e mapeamentos de propriedade, mapeamentos de herança, mapeamentos componente, e um-para-muitos mapeamentos associação.

Nós agora estender sobre esses temas, girando para a coleção mais exóticos e mapeamentos de associação que permitem

você lidar com os casos mais complicados uso. É interessante notar que esses mapeamentos mais exóticos só deve ser utilizado com consideração muito cuidadosa, na verdade, é possível implementar qualquer modelo de domínio usando simples componentes (um-para-muitos e um-para-um). Ao longo deste capítulo iremos aconselhá-lo sobre quando você pode ou não quer usar os recursos avançados como eles são discutidos.

Alguns desses recursos será necessário que você tenha uma compreensão mais aprofundada do NHibernate tipo de sistema, particularmente da distinção entre entidade e tipos de valor. Então, é aí que nós vamos começar

6.1 Compreender o sistema de tipo NHibernate

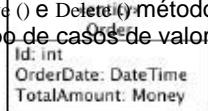
Primeiro, a distinção entre entidade e tipos de valor de volta no capítulo 3, seção 3.6.1 - "Entidade e tipos de valor ". , A fim de lhe dar uma melhor compreensão do sistema tipo NHibernate, vamos elaborar sobre isso um pouco mais.

Entidades são as classes de textura granulada em um sistema. Você geralmente definem as características de um sistema em termos das entidades envolvidas: "o usuário coloca um lance para um item" é uma definição característica típica que menciona três entidades - lance do usuário, e item. Em contraste, tipos de valor são os grãos mais finos classes em um sistema, tais como strings, números, datas e quantias monetárias. Essas classes de granulação fina pode ser usado em muitos lugares e servir a muitos propósitos, o tipo string valor pode armazenar o endereço de e-mail, nomes de usuários e muitas outras coisas. Strings são tipos de valores simples, mas é possível (mas menos comum) para criar tipos de valor que são mais complexas. Por exemplo, um tipo de valor pode conter vários campos, como um endereço.

Então, como vamos diferenciar entre tipos de valor e entidades? Do ponto de vista mais formal, nós Pode-se dizer uma entidade é qualquer classe cujas instâncias têm sua própria identidade persistente, e um tipo de valor é uma classe que está casos não. As instâncias entidade pode, portanto, ser em qualquer um dos três persistente ciclo de vida afirma: transitória, individual, ou persistente. No entanto, não consideramos esses estados ciclo de vida para

se aplicam aos casos mais simples tipo de valor. Além disso, porque as entidades têm o seu próprio ciclo de vida, o

Save () e Delete () métodos do NHibernate ISession interface irá aplicar-lhes, mas nunca para Tipo de casos de valor. Para ilustrar, vamos considerar a Figura 6.1.1.



QuantiaTotal é uma instância de tipo de valor Dinheiro. Porque os tipos de valor são completamente ligado a esse suas entidades possuir, QuantiaTotal só é salvo quando o Ordem é salvo.

Associações e tipos de valor

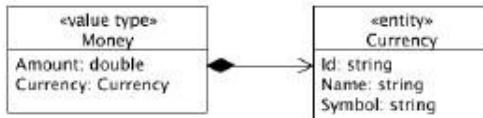
Como dissemos, nem todos os tipos de valor são simples. É possível para tipos de valor também para definir as associações.

Por exemplo, nosso Dinheiro tipo de valor pode ter uma propriedade chamada Moeda que é uma associação com uma

Moeda entidade como mostrado na figura 6.1.2

Figura 6.1.2 - O tipo de valor do dinheiro com a associação a uma entidade de moeda.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



Se os seus tipos de valor têm associações, devem sempre ponto para entidades. A razão é que, se essas associações poderiam apontar a partir de entidades para tipos de valor, um tipo de valor poderia pertencer a vários entidades, que não é desejável. Esta é uma das grandes coisas sobre tipos de valor, se você atualizar um valor instância do tipo, você sabe que ela afeta apenas a entidade que possui -lo. Por exemplo, alterar o QuantiaTotal de um Ordem simplesmente não pode acidentalmente afetar os outros. Até agora falamos sobre o valor tipos e entidades a partir de uma perspectiva orientada a objetos. Para construir uma imagem mais completa, vamos agora dar uma olhada em como o modelo relacional vê os tipos de valores e entidades, e como o NHibernate ponte lacuna.

Ponte de objetos para banco de dados

Você pode estar ciente de que um arquiteto de banco de dados seria ver o mundo de tipos de valor e entidades ligeiramente diferente com esta visão orientada a objetos das coisas. No banco de dados, tabelas representam as entidades, e colunas representam os valores. Mesmo se juntar tabelas e tabelas de pesquisa são entidades. Assim, se todas as tabelas representam entidades no banco de dados, isso significa que temos que mapear todas as tabelas para entidades em nosso domínio. NET modelo? Que sobre aqueles tipos de valor que queríamos em nosso modelo? NHibernate fornece construções para lidar com isso. Por exemplo, um mapeamento da associação many-to-many esconde o intermediário tabela de associação da aplicação, de modo que não acabem com uma entidade indesejadas em nosso domínio modelo. Da mesma forma, uma coleção de sequências valor digitado se comporta como um tipo de valor a partir do ponto de vista do modelo de domínio. NET apesar de ser mapeada para sua própria tabela no banco de dados.

Estas características têm seus usos e muitas vezes pode simplificar o seu código C#. No entanto, ao longo do tempo temos ficar desconfiado deles, estes "escondido" entidades muitas vezes acabam necessitando de exposição em nossos aplicativos como requisitos de negócio evoluir. A tabela de associação many-to-many, por exemplo, tem muitas vezes colunas adicionais adicionadas como a aplicação amadurece, para que o relacionamento se torna uma entidade. Você pode não ir muito errado se você fizer todas as entidades do banco de dados de nível de ser exposto para a aplicação externa. Classes de entidade de banco de dados rapidamente se tornam tipos de elementos de entidades, muitas vezes carregando objeto-mapeamento de perspectiva de banco de dados. Sabemos que as entidades de mapeamento é voltado simplesmente para mapeamento de elementos.

- Classes de entidade são simplesmente sempre mapeado para tabelas de banco de dados usando <class>, <subclass> E <joined-subclass> mapeamento de elementos.

Tipos de valor precisa de algo mais, que é onde tipos de mapeamento entram em cena. Considerar este mapeamento das CaveatEmptor Usuário e endereço de e-mail:

```

<class name = "Email">
    <column name = "EMAIL" type = "String" />

```

Em ORM, você tem que se preocupar tanto. NET e Tipos de dados SQL. No exemplo acima, imagine que o E-mail campo é um NET. string, e EMAIL coluna é um SQL varchar. Queremos dizer NHibernate saber como realizar esta conversão, que é onde os tipos de mapeamento NHibernate virá. Neste caso, nós especificado o tipo de mapeamento "String", que sabemos que é apropriado para esse conversão particular.

O Corda tipo de mapeamento não é o único construído em NHibernate; NHibernate vem com vários tipos de mapeamento que define estratégias de persistência padrão para primitivos, NET e certas classes, tais como como DateTime.

6.1.1 Built-in tipos de mapeamento

Built-in do NHibernate tipos de mapeamento geralmente refletem o nome do tipo. NET eles mapeiam. Às vezes você terá uma escolha de tipos de mapeamento disponíveis para mapear um tipo. NET especial à base de dados. No entanto, o built-in tipos de mapeamento não são projetados para realizar conversões arbitrárias, tais como mapeamento de uma VARCHAR valor do campo para um NET. Int32 valor da propriedade. Se você quer este tipo de funcionalidade, você terá que definir os seus próprios personalizado tipos de valor. Vamos chegar a esse tópico um pouco mais adiante neste capítulo.

Vamos agora discutir os tipos básicos; data e hora, objetos, objetos grandes, e outros vários built-in mapeamento de tipos e mostrar o que. NET e System.Data.DbType tipos de dados que manipulam. DbTypes são usados para inferir os tipos de dados do provedor (daí tipos de dados SQL).

. NET tipos primitivos de mapeamento

Os tipos básicos de mapeamento na tabela mapa 7.1. NET tipos primitivos para DbTypes apropriado.

Tabela 7.1 Tipos primitivos

Tipo de mapeamento	. NET tipo	System.Data.DbType
Int16	System.Int16	DbType.Int16
Int32	System.Int32	DbType.Int32
Int64	System.Int64	DbType.Int64
Único	System.Single	DbType.Single
Duplo	System.Double	DbType.Double
Decimal	System.Decimal	DbType.Decimal
Byte	System.Byte	DbType.Byte
Char	System.Char	DbType.StringFixedLength - 1 caractere
AnsiChar	System.Char	DbType.AnsiStringFixedLength - 1 caractere
Boolean	System.Boolean	DbType.Boolean
Guid	System.Guid	DbType.Guid
PersistentEnum	System.Enum (uma enumeração)	O DbType para o valor subjacente
TrueFalse	System.Boolean	DbType.AnsiStringFixedLength - ou 'T' ou 'F'
YesNo	System.Boolean	DbType.AnsiStringFixedLength - ou 'Y' ou 'N'

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Você provavelmente já percebeu que seu banco de dados não suporta alguns dos DbType's listados na tabela 6.2.

No entanto, ADO.NET fornece uma abstração parcial do fornecedor específico tipos de dados SQL, permitindo NHibernate para trabalhar com ANSI-padrão tipos durante a execução de linguagem de manipulação de dados (DML). Para

banco de dados específico geração de DDL, traduz NHibernate do tipo ANSI-padrão para um tipo de fornecedor específico apropriado, usando o built-in suporte para dialetos SQL específico. (Você normalmente

não precisa se preocupar com tipos de dados SQL se você estiver usando NHibernate para acesso a dados e esquema de dados definição.)

NHibernate suporta um número de tipos de mapeamento do Hibernate para vindo de compatibilidade (útil para quem vem ao longo de hibernação ou utilizando ferramentas Hibernate para gerar arquivos hbm.xml).

Tabela 6.2

lista os nomes de tipos adicionais de mapeamento NHibernate.

Tipo de mapeamento	Nome adicional
Binário	binário
Boolean	boolean
Byte	byte
Caráter	caráter
CultureInfo	localidade
DateTime	datetime
Decimal	big_decimal
Duplo	duplo
Guid	guid
Int16	curto
Int32	int
Int32	número inteiro
Int64	longo
Único	flutuar
Corda	corda
TrueFalse	true_false
Tipo	classe
YesNo	yes_no

A partir desta tabela, você pode ver que a escrita type = "inteiro" ou type = "int" é idêntica à type = "Int32". Note que esta tabela contém tipos de mapeamento de muitos que serão discutidas nas seções seguintes.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Data / hora tipos de mapeamento

Tabela 6.3 lista os tipos NHibernate associados com datas, horas e carimbos do tempo. Em seu modelo de domínio, você pode escolher para representar data e hora de dados usando o System.DateTime OU System.TimeSpan. Como eles têm finalidades diferentes, a escolha deve ser fácil.

TypesNHIAMEAP07.doc Tabela 6.3 Data e hora

Tipo de mapeamento	.NET tipo	System.Data.DbType
DateTime	System.DateTime	DbType.DateTime - ignora o milissegundos
Carrapatos	System.DateTime	DbType.Int64
TimeSpan	System.TimeSpan	DbType.Int64
Timestamp	System.DateTime	DbType.DateTime - tão específico quanto banco de dados suporta

Mapeamento de tipos de objetos

Todos os tipos .NET nas tabelas 6.1 e 6.3 são tipos de valor (ou seja derivado do System.ValueType). Isso significa que que não pode ser nulo, a menos que você usar o NET 2.0. <T> Anulável estrutura ou o Nullables add-in, como discutido na próxima seção. Tabela 6.4 lista os tipos NHibernate para manipulação. NET derivado System.Object (que pode armazenar valores nulos).

TypesNHIAMEAP07.doc objeto Nullable

Tipo de mapeamento	.NET tipo	System.Data.DbType
Corda	System.String	DbType.String
AnsiString	System.String	DbType.AnsiString

Esta tabela é completada por tabelas 6.5 e 6.6 que também contêm tipos de mapeamento anulável.

Tipos de mapeamento objeto grande

Tabela 6.5 lista os tipos NHibernate para manipulação de dados binários e objetos grandes. Note-se que nenhuma dessas tipos podem ser utilizados como o tipo de uma propriedade de identificador.

Binary tabela 6.5 e typesNHIAMEAP07.doc objeto grande

Tipo de mapeamento	.NET tipo	System.Data.DbType
Binário	System.Byte []	DbType.Binary
BinaryBlob	System.Byte []	DbType.Binary
StringClob	System.String	DbType.String

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Serializable	Qualquer System.Object marcados com SerializableAttribute	DbType.Binary
--------------	--	---------------

BinaryBlob e StringClob são principalmente suportados pelo SQL Server. Eles podem ter um tamanho muito grande e são totalmente carregado na memória. Este pode ser um assassino de desempenho se usado para armazenar objetos muito grandes. Portanto, use esse recurso com cuidado. Note que você deve definir a propriedade NHibernate "Prepare_sql" para "Verdadeiro" para ativar esse recurso.

Você pode encontrar os padrões de up-to-date design e dicas para o uso do objeto grande no site do NHibernate.

Vários tipos de mapeamento CLR

Tabela 6.6 lista os tipos NHibernate para vários outros tipos do CLR que pode ser representado como DbType.StringS no banco de dados.

Tabela 6.6 Outros CLR relacionados typesNHIAMEAP07.doc

Tipo de mapeamento	.NET tipo	System.Data.DbType
CultureInfo	System.Globalization.CultureInfo	DbType.String - 5 chars para a cultura
Tipo	System.Type	DbType.String segurando Assembléia Nome qualificado

Certamente, <property> não é apenas o elemento de mapeamento NHibernate que tem um tipo atributo.

6.1.2 Usando tipos de mapeamento

Todos os tipos de mapeamento básico podem aparecer quase em qualquer lugar no documento de mapeamento NHibernate, em propriedade normal, de propriedade de identificação, mapeamento e elementos de outro.

O <id>, <property>, <versão>, <discriminador>, <index> E <element> todos os elementos definir um atributo chamado tipo. (Existem certas limitações sobre quais tipos de mapeamentos básicos podem funcionar como um tipo de identificador ou discriminador, no entanto.)

Você pode ver o quanto útil os tipos built-in de mapeamento estão neste mapeamento para o BillingDetails classe:

```

<Nome da classe = "BillingDetails"
  table = "BILLING_DETAILS"
  lazy = "false"
  discriminator-value = "0">

  <id name="Id" type="Int32" column="BILLING_DETAILS_ID">
    <generator class="native"/>
  </ Id>

  <discriminator type="Char" column="TYPE"/>

  <property name="Number" type="String"/>
  ...
</ Class>

```

O BillingDetails classe é mapeada como uma entidade. Sua discriminador,idE Número propriedades são valor digitado, e usar o built-in tipos de mapeamento NHibernate para especificar a estratégia de conversão.

Muitas vezes não é necessário especificar explicitamente um built-in tipo de mapeamento no mapeamento XML documento. Por exemplo, se você tem uma propriedade de. NET System.String, NHibernate irá descobrir esta reflexão usando e selecione Corda por padrão. Podemos facilmente simplificar o anterior mapeamento de exemplo:

```

<Nome da classe = "BillingDetails"
  table = "BILLING_DETAILS"
  lazy = "false"
  discriminator-value = "0">

  <id name="Id" column="BILLING DETAILS_ID">
    <generator class="native"/>
  </ Id>

  <discriminator type="Char" column="TYPE"/>

  <property name="Number"/>
  ...
</ Class>

```

Para cada um dos tipos built-in de mapeamento, uma constante é definida pela classe NHibernate. NHibernateUtil. Por exemplo, NHibernate.String representa a Corda mapeamento de tipo. Essas constantes são úteis para vinculativo consulta parâmetro, como discutido em maiores detalhes no capítulo 8:

```

session.CreateQuery ("do item i, onde i.Description como: desc")
  . SetParameter ("desc", desc, NHibernate.String)
  . List ();

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Essas constantes são também úteis para manipulação programática do mapeamento NHibernate meta-modelo, como discutido no capítulo 3.

Claro, NHibernate não se limita ao built-in tipos de mapeamento, você pode criar seu próprio personalizado mapeamento de tipos para manipulação de determinados cenários. Nós vamos dar uma olhada neste próximo, e explica como o mapeamento sistema de tipos é uma central para a flexibilidade NHibernates.

Criar tipos de mapeamento customizados

Linguagens orientadas a objeto como C # torná-lo fácil de definir novos tipos escrevendo novas classes. Na verdade,

esta é uma parte fundamental da definição de orientação a objetos. Se você fosse limitada à pré-definidos built-in tipos de mapeamento NHibernate ao declarar propriedades de classes persistentes, você perderia muito do

Expressividade C # 's. Além disso, a implementação do modelo de domínio seria intimamente ligado ao o modelo de dados físicos, já que as conversões de tipo novo seria impossível.

A fim de evitar que, NHibernate fornece uma característica muito poderosa chamada tipos de mapeamento customizados.

NHibernate fornece duas interfaces amigáveis que as aplicações podem utilizar na definição de novos tipos de mapeamento, sendo a primeira NHibernate.UserTypes.IUserType é adequado para a maioria casos simples e até mesmo para alguns problemas mais complexos. Vamos usá-lo em um cenário simples.

Nosso Oferta classe define um Quantidade propriedade e os nossos Item classe define um InitialPrice propriedade, ambos os valores monetários. Até agora, só usei uma simples System.Double para representar o valor, mapeados com Duplo a um único DbType.Double coluna.

Suponha que nós queríamos para apoiar várias moedas em nosso aplicativo de leilão e que tínhamos que refatorar o modelo de domínio já existente para esta mudança. Uma forma de implementar essa mudança seria adicionar novas propriedades para Oferta e Item: AmountCurrency e InitialPriceCurrency. Nós, então, mapear essas novas propriedades adicionais VARCHAR colunas com o built-in Corda mapeamento de tipo. Imagine se tivéssemos moeda armazenados em 100 lugares, este seria muitas mudanças. Nós esperamos que você nunca usar essa abordagem!

Criação de uma implementação de IUserType

Em vez disso, devemos criar um MonetaryAmount classe que incorpora tanto a moeda e montante. Isto é uma classe do modelo de domínio e não tem qualquer dependência em interfaces NHibernate:

```
[Serializable]
MonetaryAmount public class
{

    valor readonly private double;
    moeda private string readonly;

    (duplo valor, moeda string) MonetaryAmount pública
    {
        this.value = valor;
        this.currency = moeda;
    }

    Valor public double {get {valor de retorno;}}
    Moeda cadeia pública {get {moeda return;}}

    public override bool Equals (object obj) {...}
    public override int GetHashCode () {...}
}
```

Nós também fizemos a vida mais simples, fazendo MonetaryAmount uma classe imutável, o que significa que não pode ser

mudou depois é instanciado. Teríamos que implementar Equals () e GetHashCode () para completa a classe -, mas não há nada especial a considerar aqui de lado que eles devem ser consistentes, e GetHashCode () deve retornar em sua maioria números únicos.

Vamos usar esse novo MonetaryAmount para substituir o Duplo, conforme definido na InitialPrice propriedade para Item. Gostaríamos de beneficiar usando esta nova classe em outros lugares, como o Bid.Amount.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O próximo desafio é no mapeamento nosso novo MonetaryAmount propriedades para o banco de dados. Supor estamos trabalhando com um banco de dados legado que contém todas as quantias monetárias em dólares. Nossa nova classe significa que o nosso aplicação código não está mais restrito a uma única moeda, mas isso vai levar tempo para obter o mudanças feitas pela equipe do banco de dados. Até que isso aconteça, gostaríamos de armazenar apenas a Quantidade propriedade da MonetaryAmount ao banco de dados. Porque não podemos guardar a moeda, no entanto, vamos converter todos os Quantidades para USD antes de salvá-los, e de USD quando nós carregá-los.

O primeiro passo para lidar com isso é para dizer NHibernate como lidar com os nossos Monetaryamount tipo. Para fazer isso discutimos um tipo de mapeamento personalizado que implementa a interface NHibernate IUserType. Nossa tipo de mapeamento personalizado é mostrado na listagem 6.1.

```

using System;
using System.Data;
usando NHibernate.UserTypes;

public class MonetaryAmountUserType: IUserType
{
    private static readonly NHibernate.SqlTypes.SqlType [] = SQL_TYPES
        {NHibernateUtil.Double.SqlType};

    pública NHibernate.SqlTypes.SqlType [] {SqlTypes
        get {SQL_TYPES return;}}
    }
    ReturnedType Tipo pública {get {return typeof (MonetaryAmount);}}
    }

    novo público bool Equals (objeto x, y objeto) {
        if (Object.ReferenceEquals (x, y)) return true;
        if (x == null || y == null) return false;
        x.Equals retorno (y);
    }
    pública objeto deepcopy (valor do objeto) {valor de retorno;}
    }

    public bool IsMutable {get {return false;}}
    }

    pública objeto NullSafeGet (IDataReader dr, string [] nomes, proprietário do objeto) {|
        obj = objeto NHibernateUtil.Double.NullSafeGet (dr, nomes [0]);
        if (obj == null) return null;
        dupla valueInUSD obj = (double);
        retorno MonetaryAmount novo (valueInUSD, "USD");
    }

    NullSafeSet public void ( IDbCommand cmd, objeto obj, int index) {|
        if (obj == null) {
            ((IDataParameter) cmd.Parameters [indice]) = Valor DBNull.Value.;
        Else {}
            AnyCurrency MonetaryAmount = (MonetaryAmount) obj;
            MonetaryAmount amountInUSD =
                MonetaryAmount.Convert (anyCurrency, "USD");

            ((IDataParameter) cmd.Parameters [indice]) = Valor amountInUSD.Value.;
        }
    }

    Converter public static MonetaryAmount (m MonetaryAmount,
        targetCurrency string)
    {
        ...
    }
}

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O `SqlTypes` propriedade diz NHibernate que tipos de coluna SQL a ser usado para geração de esquema DDL, como

visto em # 1. Os tipos são subclasses de `NHibernate.SqlTypes.SqlType`. Note-se que essa propriedade retorna uma variedade de tipos. Uma implementação de `IUserType` pode mapear uma propriedade única múltiplo colunas, mas o nosso modelo de dados único legado tem um único Duplo.

Em # 2, podemos ver que `ReturnedType` diz o NHibernate. NET é mapeado por esta `IUserType`.

O `IUserType` é responsável por valores suja controlo propriedade (# 3). O `Equals()` método compara o valor da propriedade atual para um instantâneo anterior e determina se a propriedade é sujo e deve, por salvas no banco de dados.

O `IUserType` também é parcialmente responsável por criar o instantâneo, em primeiro lugar, como mostrado na # 4. Desde `MonetaryAmount` é uma classe imutável, o `Deepcopy()` método retorna seu argumento. No caso de um tipo mutável, seria necessário retornar uma cópia do argumento a ser utilizado como o instantâneo valor. Este método também é chamado quando uma instância do tipo é gravado ou lido a partir da segunda cache de nível.

NHibernate pode fazer algumas otimizações de desempenho menor para tipos imutáveis. O `IsMutable` (# 5) propriedade NHibernate diz que este tipo é imutável.

O `NullSafeGet()` método mostrado perto # 6 recupera o valor da propriedade do ADO.NET `IDataReader`. Você também pode acessar o proprietário do componente, se você precisar dele para a conversão. Todos os valores banco de dados são em dólares, então você tem que converter o `MonetaryAmount` retornado por esse método antes de mostrá-lo para o usuário. Em # 7, o `NullSafeSet()` método grava o valor da propriedade para o ADO.NET `IDbCommand`. Este método leva o que quer que moeda é definida e converte-lo a um simples Duplo USD valor antes de salvar.

Note-se que, por brevidade, não têm proporcionado uma `Converter` funcionar como mostrado no # 8. Se fôssemos implementá-lo, teria um código que faz a conversão entre diversas moedas.

Mapeamento da `InitialPrice` propriedade da `Item` pode ser feito da seguinte forma:

```
<Nome da propriedade = "InitialPrice"
    coluna = "INITIAL_PRICE"
    type = "NHibernate.Auction.CustomTypes.MonetaryAmountUserType,
    NHibernate.Auction "/>
```

Este é o tipo mais simples de transformação que a implementação de um `IUserType` poderia realizar. Ele tem uma classe Tipo de valor e mapas para uma coluna de banco de dados único. Muito coisas mais sofisticadas são possível, um tipo de mapeamento personalizado pode executar a validação, pode ler e escrever dados de e para um Active Directory, ou pode até mesmo recuperar objetos persistentes de uma NHibernate diferentes `ISession` para um banco de dados diferente. Você está limitado principalmente pela sua imaginação e as preocupações performance!

Em um mundo perfeito, nós preferimos para representar a quantidade e moeda dos nossos valores monetários no banco de dados, de modo que não está limitado a armazenar apenas USD. Podemos ainda utilizar um `IUserType` para isso, mas

ele é limitado; Se um `IUserType` é mapeado com mais de uma propriedade, não podemos usá-los ou os nossos HQL

Consultas critérios. O mecanismo de consulta NHibernate não sabe nada sobre o indivíduo propriedades de `MonetaryAmount`. Você ainda acessar as propriedades em seu código C # (`MonetaryAmount` é apenas uma classe regular do modelo de domínio, afinal), mas não em consultas NHibernate.

Para permitir um tipo de valor personalizado com várias propriedades que podem ser acessados em consultas, que deve usar o `ICompositeUserType` interface. Esta interface expõe as propriedades do nosso `MonetaryAmount` para NHibernate.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Criação de uma implementação de ICompositeUserType

Para demonstrar a flexibilidade de tipos de mapeamento de costume, não teremos de mudar o nosso MonetaryAmount classe de domínio modelo em todos os nós mudamos apenas o tipo de mapeamento personalizado, como mostrado na listagem 6.2.

Listagem 6.2 tipo de mapeamento personalizado para quantias monetárias em esquemas novo banco de dados

```
using System;
using System.Data;
using NHibernate.UserTypes;

public class MonetaryAmountCompositeUserType: ICompositeUserType {
    ReturnedClass Tipo pública {get {return typeof (MonetaryAmount);}}
    novo público bool Equals (objeto x, y objeto) {
        if (Object.ReferenceEquals (x, y)) return true;
        if (x == null || y == null) return false;
        x.Equals retorno (y);
    }
    pública objeto deepcopy (valor do objeto) {valor de retorno;}
    public bool IsMutable {get {return false;}}
}

pública objeto NullSafeGet (IDataReader dr, string [] nomes,
    Sessão NHibernate.Engine.ISessionImplementor, proprietário do objeto) {
    objeto obj0 = NHibernateUtil.Double.NullSafeGet (dr, nomes [0]);
    obj1 = objeto NHibernateUtil.String.NullSafeGet (dr, nomes [1]);
    if (obj0 == null || obj1 == null) return null;
    duplo valor = (double) obj0;
    string de moeda = obj1 (string);
    retorno MonetaryAmount novo (moeda, valor);
}

NullSafeSet public void (IDbCommand cmd, objeto obj, int index,
    Sessão NHibernate.Engine.ISessionImplementor) {
    if (obj == null) {
        ((IDataParameter) cmd.Parameters [indice]) = Valor DBNull.Value.;
        ((IDataParameter) cmd.Parameters [index + 1]) = Valor DBNull.Value.;
    } Else {
        Quantidade MonetaryAmount = (MonetaryAmount) obj;
        ((IDataParameter) cmd.Parameters [indice]) = Valor amount.Value.;
        . ((IDataParameter) cmd.Parameters [index + 1]) = Valor amount.Currency;
    }
}

public string [] {propertyNames
    get {return new string [] {"Valor", "Moeda";}}
}
pública NHibernate.Type.IType [] {PropertyTypes
    get {return new NHibernate.Type.IType [] {
        NHibernateUtil.Double, NHibernateUtil.String;}}
}
pública objeto GetPropertyValue (componente objeto, propriedade int) {
    Quantidade MonetaryAmount = (MonetaryAmount) componente;
    if (propriedade == 0)
        retorno amount.Value;
    outro
        retorno amount.Currency;
}
public void SetPropertyValues (objeto comp, propriedade int, valor do objeto) {
    throw new Exception ("Imutável!");
}

| 1
| 2
| 3
| 4
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        objeto público Montar (objeto em cache,
            Sessão NHibernate.Engine.ISessionImplementor, proprietário do objeto) {
            retorno em cache;
        }
        objeto público Desmonte (valor do objeto,
            Sessão NHibernate.Engine.ISessionImplementor) {
            valor de retorno;
        }
    }

    | 5
    | 6

```

1 mostra como uma implementação de ICompositeUserType tem suas próprias propriedades, definida por PropertyNames.

Da mesma forma, as propriedades de cada um tem seu próprio tipo, conforme definido pela PropertyTypes (# 2). O GetPropertyValue () método, mostrado na # 3, retorna o valor de uma propriedade individual do MonetaryAmount.

Desde MonetaryAmount é imutável, não pode definir valores de propriedade individual (ver item 4) Esta não é uma problema porque este método é opcional de qualquer maneira.

Em # 5, o Montar () método é chamado quando uma instância do tipo é lido a partir do segundo nível cache.

O Desmonte () método é chamado quando uma instância do tipo é escrito para o segundo nível cache, como mostrado na # 6.

A ordem das propriedades deve ser o mesmo na PropertyNames,PropertyTypesE GetPropertyValues () métodos. O InitialPrice propriedade agora mapas de duas colunas, de modo que declarar tanto no arquivo de mapeamento. A primeira coluna armazena o valor, o segundo armazena a moeda de o MonetaryAmount. Note que a ordem de colunas deve corresponder a ordem das propriedades em seu tipo implementação:

```

<Nome da propriedade = "InitialPrice"
    type = "NHibernate.Auction.CustomTypes.MonetaryAmountCompositeUserType,
    NHibernate.Auction ">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CURRENCY"/>
</ Property>

```

Em uma consulta, nós podemos agora referem-se ao Quantidade e Moeda propriedades do tipo custom, apesar de eles não aparecem em qualquer lugar do documento de mapeamento de propriedades individuais:

```

do item i
onde i.InitialPrice.Value > 100,0
    e i.InitialPrice.Currency = 'XAF'

```

Neste exemplo, nós temos expandido o buffer entre o modelo de objeto. NET e banco de dados SQL esquema com o nosso tipo de composto personalizado. Ambas as representações podem agora lidar com as mudanças de forma mais enérgica.

Se implementar tipos personalizados parece complexo, relaxar, você raramente precisará usar um mapeamento personalizado tipo. Uma forma alternativa para representar o MonetaryAmount classe é a utilização de um mapeamento de componentes, como em Secção 4.4.2, "Usando componentes." A decisão de usar um tipo de mapeamento personalizado é muitas vezes uma questão de gosto.

Há poucos mais interfaces que podem ser usados para implementar tipos personalizados, que são introduzidas Outras interfaces para criar tipos de mapeamento customizados No Você pode achar que as interfaces IUserType e ICompositeUserType não permitem que você facilmente adicionar proxima seção mais recursos para seus tipos personalizados. Neste caso, você precisará usar algumas das outras interfaces que estão no NHibernate.UserTypes namespace:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O IParameterizedType interface permite que você fornecer parâmetros para o seu tipo personalizado no mapeamento de arquivo. Esta interface tem um método único: SetParameterValues (parâmetros IDictionary) que será chamado na inicialização do seu tipo. Aqui está um exemplo de mapeamento de fornecer um parâmetro:

```
<property name="Price">
    <type name="NHibernate.Auction.CustomTypes.MonetaryAmountUserType">
        <param Euro name="DefaultCurrency"> </ param>
    </ Type>
</ Property>
```

Esse mapeamento indica o tipo personalizado para usar Euro como moeda, se não for especificado.

O IEhancedUserType interface torna possível implementar um tipo personalizado que pode ser marshalled de e para sua representação string. Esta funcionalidade permite que este tipo para ser utilizado como identificador ou tipo discriminador. Para criar um tipo que pode ser usado como versão, você deve implementar a IUserVersionType interface.

O INullableUserType interface permite que você para interpretar valores não nulos em uma propriedade como nulo em banco de dados. Ao usar dynamic-insert ou dynamic-update, Os campos identificados como nula não será inseridos ou atualizados. Esta informação pode também ser usado na geração do cláusula where do SQL comando quando o bloqueio otimista é habilitado.

A interface último é diferente dos anteriores porque se destina a implementar definido pelo usuário tipos de coleção: IUserCollectionType. Para mais detalhes, dê uma olhada na implementação NHibernate.Test.UserCollection.MyListType no código fonte do NHibernate.

Agora, vamos olhar para uma aplicação extremamente importante de tipos de mapeamento customizados. Tipos anulável são encontrada em quase todas as aplicações empresariais.

Usando tipos Nullable

. Com Framework 1.1, tipos primitivos não pode ser nulo, mas isso não é mais o caso na NET 2.0.. Vamos dizer que queremos adicionar um DismissDate para a classe Usuário. Enquanto um usuário está ativo, seu DismissDate deve ser nulo. Mas o System.DateTime struct não pode ser nulo. E nós não queremos usar algum Valor "mágico" para representar o estado nulo.. Com NET 2.0 (e 3.5, é claro), você pode simplesmente escrever:

```
Usuário public class
{
    ...
    DateTime privada? dismissDate;
    DateTime público? DismissDate
    {
        get {return dismissDate;}
        set {dismissDate = valor;}
    }
    ...
}
```

Omitimos outras propriedades e métodos, porque nos concentrarmos na propriedade anulável. E nenhuma mudança é requerido no mapeamento.

Se você trabalha com. NET 1.1, o Nullables add-in (no pacote NHibernateContrib para versões antes NHibernate 1.2.0) contém um número de tipos de mapeamento customizados que permitem tipos primitivos de ser nulo. Para o nosso caso anterior, podemos usar o Nullables.NullableDateTime classe:

```
usando Nullables;
[Classe]
Usuário public class {
    ...
    dismissDate NullableDateTime privado;
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    [Propriedade]
    pública NullableDateTime DismissDate
    {
        get {return dismissDate;}
        set {dismissDate = valor;}
    }
    ...
}

}

```

O mapeamento é bastante simples:

```

<class name="Example.Person, Example">
    ...
    <Nome da propriedade = "DateOfBirth"
        type = "Nullables.NHibernate.NullableDateTimeType,
        Nullables.NHibernate "/>
</ Class>

```

É importante notar que, no mapeamento, o tipo de DismissDate deve ser Nullables.NHibernate.NullableDateTimeType (A partir do Nullables.NHibernate.dll arquivo). Este tipo é um invólucro usado para traduzir os tipos Nullables de / para tipos de banco de dados. Mas, se ao usar o NHibernate.Mapping.Attributes biblioteca, esta operação é automática, é por isso que só tinha de colocar o atributo [Propriedade].

O NullableDateTime tipo se comporta exatamente como System.DateTime; Há mesmo implícita operadores facilmente interagir com ele. A biblioteca contém Nullables tipos anuláveis para a maioria. NET tipos primitivos suportados pelo NHibernate. Você pode encontrar mais detalhes na documentação do NHibernate.

Usando tipos enumerados

Uma enumeração (enum) É uma forma especial de tipo de valor, que herda de System.Enum e fontes de nomes alternativos para os valores de um tipo primitivo subjacente.

Por exemplo, o Comentário classe define um Classificação. Se você lembrar, em nossa aplicação CaveatEmptor, os usuários são capazes de dar outras comentários sobre outros usuários. Em vez de usar um simples int propriedade para o classificação, criamos uma enumeração:

```

public enum Rating {
    Excelente,
    Ok,
    Baixo
}

```

Em seguida, usamos este tipo para o Classificação propriedade do nosso Comentário classe. No banco de dados, avaliações seriam

representado como o tipo de valor subjacente. Neste caso (e por padrão), é Int32. E isso é tudo temos que fazer. Podemos especificar type = "Rating" em nosso mapeamento, mas é opcional; NHibernate pode usar reflexão para encontrar isso.

Um problema que você pode correr em enumerações está usando em consultas NHibernate. Considere o seguinte consulta em HQL que recupera todos os comentários classificado como "Low":

```

IQuery q =
    session.CreateQuery ("de onde c Comentário c.Rating Rating.Low =");

```

Esta consulta não funciona, porque NHibernate não sabe o que fazer com Rating.Low e vai tentar usá-lo como um literal. Temos que usar um vincular parâmetro e definir o valor de classificação para a comparação dinamicamente (que é o que precisamos por outros motivos na maioria das vezes):

```

IQuery q =
    session.CreateQuery ("de onde c Comentário c.Rating =: rating");
q.setParameter ("rating",
    Rating.Low,
    NHibernateUtil.Enum (typeof (Rating));

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

A última linha do exemplo usa o método auxiliar estática NHibernateUtil.Enum () para definir o NHibernate Tipo, Uma forma simples de saber sobre o nosso mapeamento NHibernate enumeração e como lidar com o Rating.Low valor.

Nós agora discutidos todos os tipos de tipos de mapeamento NHibernate: built-in tipos de mapeamento, definido pelo usuário tipos personalizados, e até mesmo componentes (capítulo 4). Todos eles são considerados tipos de valor, porque mapa objetos do tipo valor (não entidades) ao banco de dados. Com uma boa compreensão do que são tipos de valor, e como eles são mapeados, agora podemos passar para a questão mais complexa de coleções de valor casos digitado.

6.2 coleções mapeamento de tipos de valor

No capítulo 4, introduzimos coleções usando para representar as relações entidade. Explicitamos como, por exemplo, um Item poderia ter uma coleção de Lances em nossa aplicação CaveatEmptor. Coleções são não apenas limitado a tipos de entidades, e portanto esta seção se concentrará em como criar os mapeamentos, onde coleções armazenar instâncias de um tipo de valor. Começamos esta seção, mostrando-lhe como usar básica coleções para conter tipos de valores simples, como uma lista de corda ou DateTimes. Em seguida, passar para como você trabalha com coleções ordenadas e classificadas. Finalmente, discutiremos como você pode mapear coleções de componentes, juntamente com as armadilhas possíveis e como elas podem ser tratadas. Você verá muitos hands-on amostras de código ao longo do caminho (note que todos os mapeamentos nesta seção funcionam com ambos .NET 1.1 e coleções. NET 2.0 genéricos).

6.2.1 Armazenamento de tipos de valor em Conjuntos, sacolas, listas e

mapas Suponha que os nossos vendedores é possível anexar imagens para Items. Uma imagem só é acessível através do contendo item; ele não precisa de apoio às associações a qualquer outra entidade em nosso sistema. Neste caso, é razoável para modelar a imagem como um tipo de valor. Item teria uma coleção de imagens que NHibernate consideraria a ser parte do Item, e, portanto, sem a sua própria persistência ciclo de vida. Neste cenário de exemplo particular, vamos supor que as imagens são armazenadas como arquivos no arquivo sistema em vez de BLOBs no banco de dados, e nós vamos simplesmente armazenar arquivos no banco de dados para registro o que as imagens de cada Item tem. Vamos agora caminhar através de várias maneiras isso pode ser implementado usando NHibernate, começando com a implementação mais simples - o conjunto. A implementação mais simples é um ISET de corda nomes de arquivos. Como um lembrete, ISET é um recipiente que apenas não permite duplicar objetos, e está disponível na biblioteca Iesi.Collections. Para armazenar as imagens contra o Item usando um ISET, Nós adicionamos uma propriedade de coleção para o Item classe como segue:

```
usando Iesi.Collections.Generic;

imagens <string> privada ISET = <string> HashSet new ();

[Set (Lazy = true, Tabela = "ITEM_IMAGE")]
[Key (1 Coluna, = "item_id")]
[Element (2, TypeType = typeof (string), Coluna = "FILENAME", NotNull = true)]

Imagens <string> pública ISET {
    get {this.images return;}
    set {this.images = valor;}
}
```

Aqui está o mapeamento XML correspondente:

```
<set name="Images" lazy="true" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <element type="String" column="FILENAME" not-null="true"/>
</ Set>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

No banco de dados, nomes de arquivo de imagem estão armazenados em uma tabela chamada ITEM_IMAGE e ligado a seu dono através ITEM_ID. A partir da perspectiva do banco de dados, temos duas entidades. No entanto, NHibernate é usado para esconder esse fato para que possamos apresentar Imagens meramente como um parte de Item. O <key> declara o elemento chave estrangeira, Item_id da entidade-mãe. O <element> tag declara essa coleção como uma coleção de casos tipo de valor: neste caso, de cordas.

Como você pode recordar, um conjunto não pode conter elementos duplicados, então a chave primária da ITEM_IMAGE

tabela consiste em duas colunas na <set> declaração: Item_id e FILENAME. Veja a figura 6.1 para um exemplo de esquema da tabela:

ITEM		ITEM_IMAGE	
ITEM_ID	NAME	ITEM_ID	FILENAME
1	Foo	1	fooimage1.jpg
2	Bar	1	fooimage2.jpg
3	Baz	2	barimage1.jpg

Figura 6.1 Tabela de estrutura de dados e exemplo para uma coleção de seqüências

Não parece provável que permitiria ao usuário para anexar a mesma imagem mais de uma vez, mas suponha que nós fizemos. Que tipo de mapeamento seria apropriado, então?

Usando um saco

Uma coleção não-ordenada que permite elementos duplicados é chamado de saco. Curiosamente, a .NET quadro não define uma IBAG interface. NHibernate permite que você use um IList in .NET para simular comportamento saco; isso é consistente com o uso comum da comunidade .NET.. Para usar um saco, mudar o tipo de Imagens em Item a partir de ISET para IList, Provavelmente usando ArrayList como uma implementação. Mudando a definição da tabela da seção anterior para permitir duplicar FILENAMES requer um chave primária diferente. Usamos um <idbag> mapeamento para anexar uma coluna chave substituta para a coleção

mesa, bem como os identificadores sintéticos que usamos para as classes de entidade:

```
[Idbag (Lazy = true, Tabela = "ITEM_IMAGE")]
[CollectionId (1, TypeType = typeof (int), Coluna = "ITEM_IMAGE_ID")]
[Generator (2, Class = "sequence")]

[Key (3, coluna = "item_id")]
[Element (4, TypeType = typeof (string), Coluna = "FILENAME", NotNull = true)]

público Imagens ISET {...}
```

O mapeamento XML parecido com este:

```
<idbag name="Images" lazy="true" table="ITEM_IMAGE">
  <collection-id type="Int32" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </ Recolha-id >

  <key column="ITEM_ID"/>
  <element type="String" column="FILENAME" not-null="true"/>
<Idbag />
```

Neste caso, a chave primária é gerada ITEM_IMAGE_ID. Você pode ver uma visualização gráfica dos tabelas do banco de dados na figura 6.2.

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	Foo	1	1	fooimage1.jpg
2	Bar	2	1	fooimage1.jpg
3	Baz	3	2	barimage1.jpg

Figura estrutura Tabela 6.2 usando um saco com uma chave primária substituta

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Se você está se perguntando por que usamos `<idbag>` em vez de `<bag>`, Então tenha em mente que estaremos discutindo

sacos muito em breve. Antes disso, vamos discutir uma outra abordagem comum para armazenar nossas imagens - em um

lista ordenada.

Usando uma lista

A `<list>` mapeamento requer a adição de um coluna do índice com a tabela do banco de dados. A coluna do índice define a posição do elemento na coleção. Assim, NHibernate pode preservar a ordenação dos elementos da coleção ao recuperar a coleção do banco de dados se mapear a coleção como um `<list>`:

```
<enumerar name="Images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="POSITION"/>
  <element type="String" column="FILENAME" not-null="true"/>
</ List>
```

A chave primária consiste na `Item_id` e `POSIÇÃO` colunas. Note-se que elementos duplicados (`FILENAME`) São permitidos, o que é consistente com a semântica de uma lista. (Nós não temos de mudar o `Item classe`; os tipos que usamos anteriormente para o saco são os mesmos).

Note-se que, embora o `IList` contrato não especifica que uma lista é uma coleção ordenada; Implementação do NHibernate preserva a ordem, quando persiste a coleção.

Se a coleção é `[Fooimage1.jpg, fooimage1.jpg, fooimage2.jpg]`, O `POSIÇÃO` coluna contém os valores `0,1E 2`, Como mostrado na figura 6.3.

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	POSITION	FILENAME
1	Foo	1	0	fooimage1.jpg
2	Bar	1	1	fooimage1.jpg
3	Baz	1	2	fooimage2.jpg

Figura 6.3 Tabelas para uma lista com elementos de posicionamento

Alternativamente, poderíamos usar um array. NET, em vez de uma lista. NHibernate suporta este uso, na verdade, os detalhes de um mapeamento da matriz são virtualmente idênticos aos de uma lista. No entanto, temos muito fortemente

recomendam contra o uso de matrizes, já que matrizes não podem ser inicializadas preguiçosamente (não há maneira de procurar).

um array no nível CLR? Aqui está o mapeamento:

```
<key column="ITEM_ID"/>
<index column="POSITION"/>
<element type="String" column="FILENAME" not-null="true"/>
</ Array primitivo>
```

Agora, suponha que nossas imagens têm nomes inseridos pelo usuário, além de os nomes de arquivos. Uma forma de

este modelo in. NET seria a utilização de um Mapa, Com nomes como chaves e nomes de arquivos como valores.

Usando um mapa

Mapeamento de uma `<map>` (Desculpem-nos) é semelhante ao mapeamento de uma lista:

```
<MAP name="Images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="String" column="FILENAME" not-null="true"/>
</ Mapa>
```

A chave primária consiste na `Item_id` e `Nome_da_imagem` colunas. O `Nome_da_imagem` lojas de coluna as chaves do mapa. Mais uma vez, elementos duplicados são permitidos; ver figura 6.4 para uma visualização gráfica dos tabelas.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	IMAGE_NAME	FILENAME
1	Foo	1	Foo Image 1	fooimage1.jpg
2	Bar	1	Foo Image One	fooimage1.jpg
3	Baz	1	Foo Image 2	fooimage2.jpg

Figura 6.4 Tabelas de um mapa, usando strings como índices e elementos

Este Mapa é desordenada. E se nós queremos sempre tipo o nosso mapa com o nome da imagem?

Coleções classificadas e ordenadas

Em um abuso surpreendente do idioma Inglês, as palavras classificadas e pedido significar coisas diferentes quando

se trata de NHibernate coleções persistentes. A coleção ordenada é classificada em memória usando um.NET. IComparer. Um coleção ordenada está ordenada ao nível do banco de dados usando uma consulta SQL com uma ordem por cláusula.

Vamos fazer nosso mapa de imagens de um mapa ordenado. Esta é uma mudança simples de documento de mapeamento:

```

<Nome do mapa = "Imagens"
  lazy = "true"
  table = "ITEM_IMAGE"
  sort = "natural">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="String" column="FILENAME" not-null="true"/>
</ Mapa>
```

Especificando sort = "natural", Dizemos NHibernate para usar uma SortedMap, A triagem dos nomes de imagem de acordo com a CompareTo () método de System.String. Se você quiser algum outro fim-classificadas para exemplo, inverter a ordem alfabética, você pode especificar o nome de uma classe que implementa System.Collections.IComparer no espécie atributo. Por exemplo:

```

<Nome do mapa = "Imagens"
  lazy = "true"
  table = "ITEM_IMAGE"
  sort = "NHibernate.Auction.ReverseStringComparer, NHibernate.Auction">

  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="String" column="FILENAME" not-null="true"/>
</ Mapa>
```

NHibernate mapa classificadas usa System.Collections.SortedList na sua implementação. Um conjunto classificado, que se comporta como Iesi.Collections.SortedSet, É mapeado em uma maneira similar:

```

<Nome do conjunto = "Imagens"
  lazy = "true"
  table = "ITEM_IMAGE"
  sort = "natural">
  <key column="ITEM_ID"/>
  <element type="String" column="FILENAME" not-null="true"/>
</ Set>
```

Sacos não podem ser classificados, e não há SortedBag, Infelizmente. Nem podem ser classificados listas, porque a ordem dos elementos da lista é definido pelo índice da lista.

Alternativamente, você pode optar por usar um mapa ordenado, usando os recursos de classificação do banco de dados, em vez de na memória classificação:

```

<Nome do mapa = "Imagens"
  lazy = "true"
  table = "ITEM_IMAGE"
  fim-by = "nome da imagem asc">
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<key column="ITEM_ID"/>
<index column="IMAGE_NAME" type="String"/>
<element type="String" column="FILENAME" not-null="true"/>
</ Mapa>

```

A expressão no fim-by atributo é um fragmento de um SQL por fim cláusula. Neste caso, nós fim pelo Nome_da_imagem coluna, em ordem crescente. Você pode até escrever chamadas de função no SQL fim-by atributo:

```

<Nome do mapa = "Imagens"
  lazy = "true"
  table = "ITEM_IMAGE"
  fim-by = "inferior (FILENAME) asc">
<key column="ITEM_ID"/>
<index column="IMAGE_NAME" type="String"/>
<element type="String" column="FILENAME" not-null="true"/>
</ Mapa>

```

Observe que você pode ordenar por qualquer coluna da tabela de coleção. Ambos os conjuntos e sacos de aceitar a ordem

por atributo, mas novamente, listas não. Este exemplo usa um saco:

```

<Nome idbag = "Imagens"
  lazy = "true"
  table = "ITEM_IMAGE"
  fim-by = "ITEM_IMAGE_ID desc">
<collection-id type="Int32" column="ITEM_IMAGE_ID">
  <generator class="sequence"/>
</ Recolha-id>
<key column="ITEM_ID"/>
<element type="String" column="FILENAME" not-null="true"/>
<Idbag />

```

Em o cobre, NHibernate.Collections.ListSetanda

System.Collections.Specialized.ListDictionary para implementar conjuntos ordenados e mapas, de modo que este funcionalidade deve ser usada com cautela, uma vez que não funciona bem com um grande número de elementos.

Em um sistema real, é provável que precisaríamos para manter mais do que apenas a imagem e nome de arquivo;

nós provavelmente necessidade de criar uma Imagem classe para armazenar esta informação extra. É claro, poderíamos mapa

Imagen como uma classe de entidade, mas uma vez que já concluiu que este não é absolutamente necessário, vamos

ver o quanto ainda podemos obter sem um Imagen entidade, o que exigiria uma associação mapeamento e manuseio do ciclo de vida mais complexas.

No capítulo 3, você viu que NHibernate permite mapear classes definidas pelo usuário como componentes, que são

7.2.2 considerados tipos de valor. Isto ainda é verdade mesmo quando instâncias de componentes são elementos da coleções de componentes.

Mas agora classe define suas próprias regras de negócios de componentes, ex: SizeY. Tem uma única associação, com seu pai Item classe, conforme mostrado na figura 6.5.

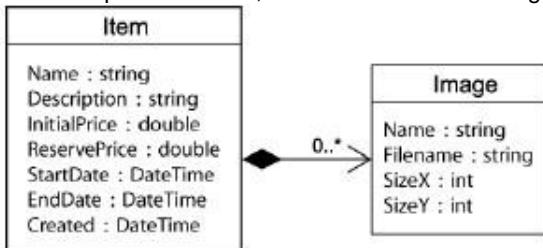


Figura 6.5 Recolha de Imagem componentes em Item

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Como você pode ver a partir do estilo de associação de agregação representado por um diamante negro, Imagem é uma componente do ItemE Item é a entidade que é responsável pelo ciclo de vida de Imagem. Referências a imagens não são compartilhadas, assim a nossa primeira escolha é um mapeamento componente NHibernate. A multiplicidade do associação declara ainda esta associação como muitos de valor, isto é, zero ou mais Imagens para o mesmo Item.

Escrever a classe de componente

Primeiro, vamos implementar a Imagem classe. Este é apenas um POCO, com nada de especial a considerar. Como você sabe do capítulo 4, classes de componentes não têm um identificador propriedade. No entanto, devemos executar Equals () e GetHashCode () para comparar a Nome, Nome do arquivo, SizeX E SizeY propriedades. Isto permite verificar suja NHibernate para funcionar corretamente. Estritamente falando, implementação Equals () e GetHashCode () não é necessária para todas as classes de componentes. No entanto, nós recomendá-lo para qualquer classe de componente porque a implementação é bastante fácil e "melhor prevenir do que sorry "é um bom lema.

O Item classe não mudou realmente, mas os objetos na coleção são agora Imagens em vez de Cordas. Vamos mapear isso para o banco de dados.

Mapeamento da coleção

Coleções de componentes são mapeados de forma semelhante ao de outras coleções de instâncias tipo de valor. A única diferença é o uso de <composite-element> no lugar do familiar <element> tag. Um conjunto ordenado de imagens pode ser mapeada como este:

```
<Nome do conjunto = "Imagens"
    lazy = "true"
    table = "ITEM_IMAGE"
    fim-by = "nome_da_imagem asc">
<key column="ITEM_ID"/>
<composite-element class="Namespaces.Image, Assembly">
    <property name="nome" column="IMAGE_NAME" not-null="true"/>
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX" not-null="true"/>
    <property name="SizeY" column="SIZEY" not-null="true"/>
</ Elemento composto->
</ Set>
```

Este é um conjunto de instâncias tipo de valor, de modo NHibernate deve ser capaz de dizer instâncias além, apesar da fato de que eles não têm coluna de chave primária em separado. Para fazer isso, todas as colunas do compósito são usados em conjunto para determinar se um item é único: Item_id, Nome_da_imagem, FILENAME, SizeX E SizeY. Desde estas colunas irão aparecer todos em uma chave primária composta, não pode ser nulo. Este é claramente um desvantagem deste mapeamento particular. Elementos compostos em uma conjunto às vezes são úteis, mas usando **NAME** não é obrigatório que lhe permitirá contornar a restrição de não nulo.

Até agora, a associação de Item para Imagem é unidirecional. Se decidimos torná-lo bidirecional, que daríamos as nossas Imagem propriedade uma classe chamada Item que é uma referência de volta para o dono Item. No mapeamento de arquivo, nós precisamos adicionar uma <parent> tag para o mapeamento:

```
<Nome do conjunto = "Imagens"
    lazy = "true"
    table = "ITEM_IMAGE"
    fim-by = "nome_da_imagem asc">
<key column="ITEM_ID"/>
<composite-element
    class="Image"> <parent name="Item"/>
    <property name="nome" column="IMAGE_NAME" not-null="true"/>
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX" not-null="true"/>
```

```

<property name="SizeY" column="SIZEY" not-null="true"/>
</ Elemento composto>
</ Set>

```

Isto leva a um problema em potencial, você vai ser capaz de carregar Imagem casos por meio de consulta para eles, mas o referência a sua propriedade será pai nulo. A melhor coisa a fazer é sempre carregar o pai, a fim para acessar suas partes componentes, ou usar uma associação entidade completa pai / filho, como descrito no capítulo 4.

Ainda temos o problema de ter que declarar todas as propriedades como não-nulo, e seria bom se nós poderia evitar isso. Vamos agora ver como podemos usar a melhor chave primária para a IMAGEM mesa. Evitar não-nulo colunas

Se um conjunto de Imagens não é a única solução, outros estilos coleção mais flexíveis são possíveis. Para exemplo, um <idbag> oferece uma chave de coleção substituto:

```

<Nome idbag = "Imagens"
      lazy = "true"
      table = "ITEM_IMAGE"
      fim-by = "nome_da_imagem asc">
  <collection-id type="Int32" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </ Recolha-id>
  <key column="ITEM_ID"/>
  <composite-element class="Namespaces.Image, Assembly">
    <property name="nome" column="IMAGE_NAME"/>
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX"/>
    <property name="SizeY" column="SIZEY"/>
  </ Elemento composto>
</idbag />

```

Desta vez, a chave primária é o ITEM_IMAGE_ID coluna. NHibernate agora não requer que nós deve implementar Equals () e GetHashCode (), Nem precisamos declarar as propriedades com não-null = "true". Eles podem ser anulável no caso de uma idbag, como mostrado na figura 6.6.

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGE_NAME	FILENAME
1	1	Foo Image 1	fooimage1.jpg
2	1	Foo Image 1	fooimage1.jpg
3	2	Bar Image 1	barimage1.jpg

Figura 6.6 Recolha de Imagem componentes usando um saco com uma chave substituta

Devemos salientar que não há uma grande diferença entre este e um saco de mapeamento entidade relacionamento padrão pai / filho. As tabelas são idênticas, e até mesmo o código C # é extremamente semelhantes, a escolha é principalmente uma questão de gosto. Naturalmente, uma relação pai / filho suporta compartilhada referências para a entidade criança e navegação bidirecional verdade.

Poderíamos até mesmo eliminar os Nome propriedade do Imagem classe e usar novamente o nome da imagem como a

chave de um mapa:

```

<Nome do mapa = "Imagens"
      lazy = "true"
      table = "ITEM_IMAGE"
      fim-by = "nome_da_imagem asc">
<key column="ITEM_ID"/>
<index type="String" column="IMAGE_NAME"/>
<class="Image"> <composite-element
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX"/>

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<property name="SizeY" column="SIZEY"/>
</ Elemento composto>
</ Mapa>

```

Como antes, a chave primária é composta de Item_id e Nome_da_imagem.

Uma classe de compostos como elemento Imagem não se limita a propriedades simples do tipo básicas, como nome do arquivo.

Pode conter componentes, usando a <nested-composite-element> declaração, e até mesmo <Many-to-one> associações de entidades. Pode não possuir coleções, no entanto. Um elemento composto com uma many-to-one associação é útil, e nós vamos voltar a este tipo de mapeamento mais adiante neste capítulo.

Estamos finalmente terminou com tipos de valor, e espero que você tenha uma visão aprofundada do que é possível e onde você pode ser capaz de fazer uso deles. A próxima coisa que veremos é avançado entidade técnicas de mapeamento da associação. A associação de pais / criança simples mapeamos no capítulo 3 é

apenas um dos muitos estilos de mapeamento possível associação. Como os mapeamentos anteriores discutimos, a maioria

destes mapeamentos são considerados "exóticos" e só precisam ser usados em casos especiais. No entanto, ter consciência das técnicas disponíveis, certamente, ajudar a resolver o mapeamento mais espinhosos desafios que você encontra na natureza.

6.3 Mapeamento de associações entidade

Quando usamos a palavra associações, estamos sempre referindo-se a relações entre as entidades. Em capítulo 4, demonstramos uma unidirecional muitos-para-uma associação, fez bidirecional, e, finalmente, transformou-o em uma relação pai / filho (um-para-muitos e muitos-para-um).

Associações um-para-muitos são o tipo mais importante e popular você vai encontrar. Na verdade, nós ir tão longe

a ponto de desestimular o uso de mais estilos associação exótica quando um simples bidirecional many-to-uma / um-para-muitos vão fazer o trabalho. Em particular, uma associação muitos-para-muitos pode ser sempre representado

como dois many-to-one associações a uma classe de intervir. Este modelo é geralmente muito mais extensível, e que raramente vai usar um mapeamento de muitos-para-muitos em nossas aplicações.

Armando com este aviso, vamos investigar associação rica do NHibernate mapeamentos começando com um-para-um.

6.3.1 One-to-one associações

Argumentamos no capítulo 4, que as relações entre Usuário e Endereço foram as melhores representadas usando <component> mapeamentos. Se bem se lembram, o usuário tem tanto uma BillingAddress e um HomeAddress em nosso modelo de amostra. Mapeamentos componente são geralmente a maneira mais simples para representar um-para-um relacionamentos, uma vez que o ciclo de vida de uma classe é quase sempre dependente do ciclo de vida dos outros classe, ea associação é uma composição.

Mas e se nós queremos uma mesa dedicada para Endereço, e para mapear tanto Usuário e Endereço como entidades?

Neste caso, as aulas têm uma associação um-para-um verdadeiro. Porque um Endereço é uma entidade, começávamos a

```

<component name="Address" table="ADDRESS" lazy="false">
    <id name="Address_ID" type="int" generator="native"/>
    <property name="Street"/>
    <property name="City"/>
    <property name="Zipcode"/>
</ Class>

```

Note-se que Endereço agora exige uma propriedade identificador; ele não é mais uma classe de componente. Há dois maneiras diferentes para representar uma associação um-para-um para este Endereço em NHibernate. A primeira abordagem adiciona uma coluna de chave estrangeira para a USUÁRIO mesa.

Usando uma associação de chave estrangeira

A maneira mais fácil para representar a associação de Usuário à sua BillingAddress é usar um <Many-to-one> mapeamento com uma restrição de unicidade na chave estrangeira. Isto pode surpreendê-lo, uma vez muitos não

parecem ser uma boa descrição de uma ou outra extremidade de uma associação um-para-um! No entanto, a partir do NHibernate

ponto de vista, não há muita diferença entre os dois tipos de associações estrangeiras chave. Então, nós adicionamos uma coluna de chave estrangeira chamada BILLING_ADDRESS_ID ao USUÁRIO mesa e mapeá-lo da seguinte forma:

```
<Many-to-one name = "BillingAddress"
  class = "Endereço"
  coluna = "BILLING_ADDRESS_ID"
  cascade = "save-update" />
```

Note-se que nós escolhemos save-update como o estilo em cascata. Isso significa que o Endereço se tornará persistentes quando criamos uma associação de um persistente Usuário. O cascade = "all" cascata seria também faz sentido para esta associação, uma vez que a exclusão do Usuário deve resultar em supressão do Endereço.

Nosso esquema de banco de dados ainda permite valores duplicados na BILLING_ADDRESS_ID coluna do USUÁRIO

mesa, de forma que dois usuários poderiam ter uma referência para o mesmo endereço. Para fazer esta associação verdadeiramente one-to-

um, nós adicionamos unique = "true" ao <many-to-one> elemento, restringindo o modelo relacional, para que pode haver apenas um endereço por usuário:

```
<Many-to-one name = "BillingAddress"
  class = "Endereço"
  coluna = "BILLING_ADDRESS_ID"
  cascade = "all"
  unique = "true" />
```

Essa alteração adiciona uma restrição de unicidade para o BILLING_ADDRESS_ID coluna no DDL gerado pelo NHibernate, resultando na estrutura da tabela ilustrada pela figura 6.7

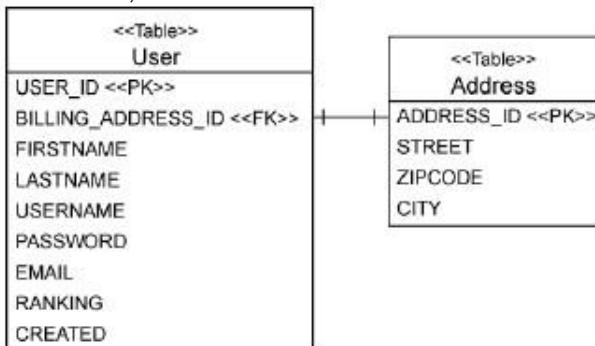


Figura 6.7 Uma associação um-para-um com uma coluna de chave extra estrangeiros

Mas e se nós queremos esta associação a ser navegável desde Endereço para Usuário in. NET? Para alcançar isso, vamos adicionar uma propriedade chamada Usuário que aponta para o Endereço classe, e mapeá-la como tal em nosso Endereço mapeamento:

```
<One-to-one name = "User"
  class = "User"
  property-ref = "BillingAddress" />
```

Este diz que o NHibernate Usuário associação em Endereço é o sentido inverso da BillingAddress associação em Usuário.

No código, criamos a associação entre os dois objetos da seguinte forma:

```
Endereço Endereço = new ();
address.Street = "Nowhere 73 Street";
address.City = "Pretória";
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

address.Zipcode = "1923";
usando (session.BeginTransaction ()) {
    Usuário user = (User) session.get (typeof (User), userId);
    address.User user =;
    user.BillingAddress address =;
    session.Transaction.Commit ();
}

```

Para terminar o mapeamento, nós também temos que mapear a `HomeAddress` propriedade da `Usuário`. Este é bastante fácil:
nós adicionamos uma outra `<many-to-one>` elemento para o `Usuário` metadados, o mapeamento de uma coluna nova chave estrangeira,

```

<many-to-one name = "HomeAddress"
    class = "Endereço"
    coluna = "HOME_ADDRESS_ID"
    cascade = "save-update"
    unique = "true" />

```

O `USUÁRIO` tabela agora define duas chaves estrangeiras referenciando a chave primária da `ENDEREÇO` tabela: `HOME_ADDRESS_ID` e `BILLING_ADDRESS_ID`.

Infelizmente, não podemos fazer ambas as `BillingAddress` e `HomeAddress` associações bidirecional, já que não sei se um determinado endereço é um endereço de cobrança ou de um endereço residencial. Mais

especificamente, nós teríamos que de alguma forma dinamicamente decidir qual propriedade de nome-`BillingAddress` ou

`HomeAddress`-A utilizar para a `property-ref` atributo no mapeamento das `usuário` propriedade. Nós poderia tentar fazer `Endereço` uma classe abstrata com subclasses `HomeAddress` e `BillingAddress` e mapeamento as associações para as subclasses. Esta abordagem iria funcionar, mas é complexo e provavelmente não sensatas neste caso.

Nosso conselho é evitar definir mais de um um-para-uma associação entre duas classes. Se você deve, deixar a associações unidirecionais. Se você não tem mais de um se realmente existe exatamente uma instância de `Endereço` por `Usuário`-Há uma abordagem alternativa à que acabamos de mostrado. Em vez de definir uma coluna de chave estrangeira na `USUÁRIO` tabela, você pode usar um chave primária associação.

Usando uma associação de chave primária

Duas tabelas relacionadas por uma associação de chave primária partes os mesmos valores de chave primária. A chave primária

de uma tabela também é uma chave estrangeira da outra. A principal dificuldade com esta abordagem é garantir que

instâncias associadas são atribuídos o valor primário mesma tecla quando os objetos são salvos. Antes de tentar resolver este problema, vamos ver como poderíamos mapear a associação de chave primária.

Para uma associação de chave primária, ambos fins da associação são mapeados usando o `<one-to-one>` declaração. Isto também significa que não podemos mais mapear tanto de faturamento e endereço residencial, apenas um propriedade. Cada linha da `USUÁRIO` tabela tem uma linha correspondente na `ENDEREÇO` mesa. Dois endereços exigiria uma tabela adicional, e este estilo de mapeamento, portanto, não seria adequada. Vamos chamar esta propriedade único endereço Endereço e mapeá-la com o `Usuário`:

```

<one-to-one name = "User"
    class = "User"
    constrained = "true" />

```

Em seguida, está aqui a `Usuário` de `Endereço`:

```

<one-to-one name = "User"
    class = "User"
    constrained = "true" />

```

A coisa mais interessante aqui é o uso de `constrained = "true"`. Ele diz NHibernate que há uma restrição de chave estrangeira na chave primária de `ENDEREÇO` que se refere a chave primária da `USUÁRIO`.

Agora devemos assegurar que os casos de recém-salvo `Endereço` são atribuídos o mesmo valor identificador como sua `Usuário`. Nós usamos uma estratégia especial de geração de identificador de chamada NHibernate estrangeiro:

```

<class name="Address" table="ADDRESS" lazy="false">
    <id name="Id" column="ADDRESS_ID">

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<generator class="foreign">
    <param name="property"> usuário </ param>
    <Gerador />
</ Id>
...
<One-to-one name = "User"
            class = "User"
            constrained = "true" />
</ Class>

```

O <param> nomeado propriedade do estrangeiro gerador nos permite o nome de uma associação de um-para-um do Endereço classe, neste caso, o usuário associação. O estrangeiro inspeciona o gerador objeto associado (o Usuário) E usa o seu identificador como o identificador do novo Endereço. Olhar para o estrutura da tabela na figura 6.8.

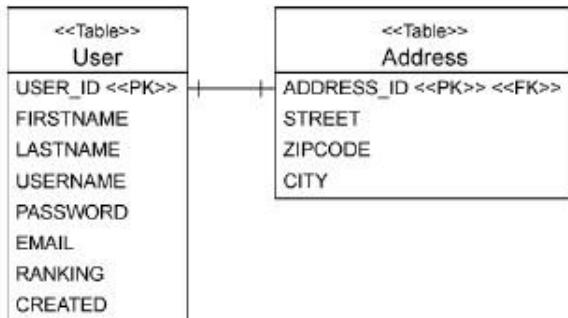


Figura 6.8 As tabelas de uma associação um-para-um com valores de chave primária compartilhada

O código para criar a associação objeto não é alterado para uma associação de chave primária, é o mesmo código que usamos anteriormente para o many-to-one estilo de mapeamento.

Agora há apenas uma multiplicidade associação restantes entidade não discutimos: many-to-muitos.

6.3.2 Muitos-para-muitos

A associação entre Categoria e Item é uma associação muitos-para-muitos, como você pode ver na figura 6.9.

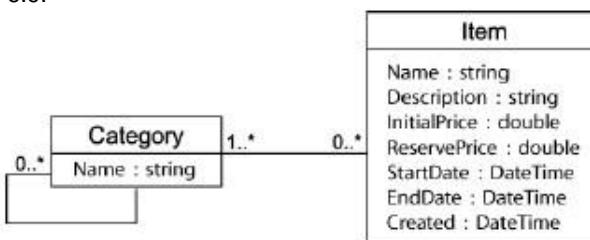


Figura 6.9 Uma associação muitos-para-muitos entre o valor de Categoria e Item

Como já explicado anteriormente nesta seção, evitamos o uso de muitos-para-muitos, pois há é quase sempre outras informações que devem ser anexados às ligações entre instâncias associadas, e a melhor maneira de representar esta informação é através de um intermediário classe de associação. No entanto, é o objetivo desta seção para implementar uma associação entidade real de muitos para muitos. Vamos começar com um exemplo unidirecional.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Uma associação unidirecional muitos-para-muitos associação

Se você só requerem a navegação unidirecional, o mapeamento é simples. Unidirecional many-to-muitas associações não são mais difíceis do que as coleções de instâncias tipo de valor que abrangeu anteriormente. Por exemplo, se o Categoria tem um conjunto de Itens, Podemos usar esse mapeamento:

```
<Nome do conjunto = "Items"
  table = "CATEGORY_ITEM"
  lazy = "true"
  cascade = "save-update">
  <key column="CATEGORY_ID"/>
  <many-to-many class="Item" column="ITEM_ID"/>
</ Set>
```

Apenas como uma coleção de instâncias de tipo de valor, uma associação muitos-para-muitos tem sua própria tabela, o link

tabela ou tabela de associação. Neste caso, a tabela de ligação tem duas colunas: as chaves estrangeiras da CATEGORIA e ITEM tabelas. A chave primária é composta de duas colunas. A estrutura completa da tabela é mostrado na figura 6.10.

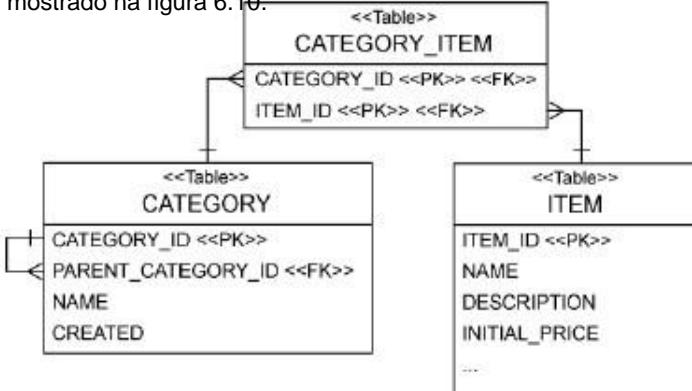


Figura 6.10 associação entidade Muitos-para-muitos mapeada para uma tabela de associação

Podemos também usar um saco com uma coluna principal separar-chave:

```
<Nome idbag = "Items"
  table = "CATEGORY_ITEM"
  lazy = "true"
  cascade = "save-update">
  <collection-id type="Int32" column="CATEGORY_ITEM_ID">
    <generator class="sequence"/>
  </ Recolha-id>
  <key column="CATEGORY_ID"/>
  <many-to-many class="Item" column="ITEM_ID"/>
<Idbag />
```

Como de costume, com uma `<idbag>` mapeamento, a chave primária é uma coluna de chave substituta, CATEGORY_ITEM_ID, e as ligações duplicadas são, portanto, permitido (o mesmo Item podem ser adicionados a um determinado Categoria duas vezes).

Outras variações que ainda pode usar são o mapa indexados ou coleções lista. O exemplo a seguir usa uma lista:

```
<Nome da lista = "Items"
  table = "CATEGORY_ITEM"
  lazy = "true"
  cascade = "save-update">
  <key column="CATEGORY_ID"/>
  <index column="DISPLAY_POSITION"/>
  <many-to-many class="Item" column="ITEM_ID"/>
</ List>
```

A chave primária consiste na Category_id e DISPLAY_POSITION colunas. Este mapeamento garante que todos os Item conhece a sua posição no Categoria.

. Criação de uma associação objeto no código NET é fácil:

```
usando (session.BeginTransaction ()) {
    Categoria cat = (Categoria) session.get (typeof (Categoria), categoryId);
    Item item = (Item) session.get (typeof (Item), itemId);

    cat.Items.Add (item);
    session.Transaction.Commit ();
}
```

Bidirecional de muitos para muitos associações são um pouco mais difícil.

A bidirecional muitos-para-muitos associação

Quando mapeamos uma associação bidirecional um-para-muitos no capítulo 3, seção 3.6, "Introducing associações ", explicou por que uma das extremidades da associação deve ser mapeado com inverse = "true". Sinta-se livre para rever essa seção, já que é relevante para comunicação de muitos para muitos associações também. Em

particular, cada linha da tabela de ligação é representada por dois elementos da coleção, um elemento em cada final da associação. Por exemplo, poderíamos criar um Item classe com uma coleção de Categoria casos, e uma Categoria classe com uma coleção de Item instâncias. Quando se trata de criar . relacionamentos em código NET, que poderia ser algo como isto:

```
cat.Items.Add (item);
item.Categories.Add (cat);
```

Independentemente da multiplicidade, uma associação bidirecional requer que você defina ambas as extremidades do associação.

Quando você mapear uma associação many-to-many bidirecional, você deve declarar um fim da associação com inverse = "true" para definir qual lado é usada para atualizar a tabela de link. Você pode escolher por si mesmo que fim que deve ser.

Lembre-se este mapeamento para o Items coleção da seção anterior:

```
<class name="Category" table="CATEGORY">
    ...
    <Nome do conjunto = "Items"
        table = "CATEGORY_ITEM"
        lazy = "true"
        cascade = "save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </ Set>
</ Class>
```

Podemos reutilizar esse mapeamento para o Categoria final da associação bidirecional. Nós o mapa Item final da seguinte forma:

```
<class name="Item" table="ITEM">
    ...
    <Nome do conjunto = "Categorias"
        table = "CATEGORY_ITEM"
        lazy = "true"
        inverse = "true"
        cascade = "save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </ Set>
</ Class>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note o uso de inverse = "true". Mais uma vez, essa configuração conta NHibernate para ignorar as alterações feitas no categorias coleta e uso a outra extremidade da associação - a itens coleção - como o representação que deve ser sincronizado com o banco de dados.

Nós escolhemos cascade = "save-update" para ambas as extremidades da coleção, que se adequou às nossas necessidades também. Note-se que cascade = "all",cascade = "delete" E cascade = "all-delete-órfãos" não são significativo para muitos-para-muitos, uma vez que uma instância com potencialmente muitos pais não devem ser excluído quando apenas um dos pais é excluído.

Outra coisa a considerar é que os tipos de coleções podem ser usadas para comunicação de muitos para muitos associações? Você precisa usar o mesmo tipo de coleta em cada extremidade? É razoável a utilização, por exemplo, não uma lista ao final marcada inverse = "true" (Ou explicitamente definido falso) E um saco no final que é marcado inverse = "true".

Você pode usar qualquer um dos mapeamentos nós mostramos para unidirecional muitos-para-muitos para o fim noninverse da associação bidirecional. <set>,<idbag>,<list>E <map> são todos possível, e os mapeamentos são idênticos aos mostrados anteriormente.

Para o fim inverso, <set> é aceitável, como é o mapeamento saco seguinte:

```
<class name="Item" table="ITEM">
    ...
    <Nome de saco = "Categorias"
        table = "CATEGORY_ITEM"
        lazy = "true"
        inverse = cascata "true" = "save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </Saco />
</ Class>
```

Esta é a primeira vez que nós mostramos a <bag> declaração: É semelhante a um <idbag> mapeamento, mas não envolve uma coluna chave substituta. Ele permite que você use um IList (Com a semântica saco) em um persistente classe em vez de um ISET. Assim, é preferível se o lado noninverse de uma associação many-to-many mapeamento está usando um mapa, uma lista ou saco (que duplica todas as licença). Lembre-se que um saco não preservar a ordem dos elementos.

Sem outros mapeamentos devem ser utilizados para o fim inverso de uma associação muitos-para-muitos. Indexados coleções como listas e mapas não pode ser usado, já que NHibernate não irá inicializar ou manter o coluna do índice se inverse = "true". Isso também é verdadeiro e importante lembrar a todos outra associação mapeamentos envolvendo coleções: uma coleção indexada, ou mesmo matrizes, não pode ser definida como inverse = "true".

Nós já franziu a testa para o uso de uma associação muitos-para-muitos e sugeriu o uso de compostos elemento mapeamentos como uma alternativa. Vamos ver como isso funciona.

Usando uma coleção de componentes para uma associação muitos-para-muitos

Suponha que precisa gravar alguma informação cada vez que adicionar um Item a um Categoria. Por exemplo, que talvez seja necessário armazenar a data eo nome do usuário que adicionou o item a esta categoria.

Precisamos

uma classe C # para representar esta informação:

```
public class CategorizedItem
{
    string username privado;
    DateTime dateAdded privada;
    item Item privado;
    categoria Categoria privado;
    ...
}
```

Você verá que nós omitimos as propriedades e Equals () e GetHashCode () métodos, mas eles ser necessário para esta classe de componente.

Nós o mapa Itens coleção em Categoria como mostrado abaixo. Se você preferir usar atributos de mapeamento no seu código, você deve ser capaz de deduzir facilmente o mapeamento usando atributos, basta ter cuidado ao ordenando-lhes:

```
<set name="Items" lazy="true" table="CATEGORY_ITEMS">
  <key column="CATEGORY_ID"/>
  class="CategorizedItem" > composite-element
    <parent name="Category"/>
    <Many-to-one name = "Item"
      class = "Item"
      coluna = "item_id"
      not-null = "true" />
      <property name="username" column="USERNAME" not-null="true"/>
      <property name="DateAdded" column="DATE_ADDED" not-null="true"/>
    </ Elemento composto->
</ Set>
```

Usamos o `<many-to-one>` elemento para declarar a associação para Item, E usamos o `<property>` mapeamentos para declarar as informações de associação relacionada extra. A tabela a ligação agora tem quatro colunas:

Category_id,Item_id,NOME DE USUÁRIOE DATE_ADDED. As colunas da CategorizedItem propriedades nunca deve ser nulo: Caso contrário, não pode identificar uma entrada única ligação, porque todos eles são parte do

chave primária tabela. Você pode ver a estrutura da tabela na figura 6.11.

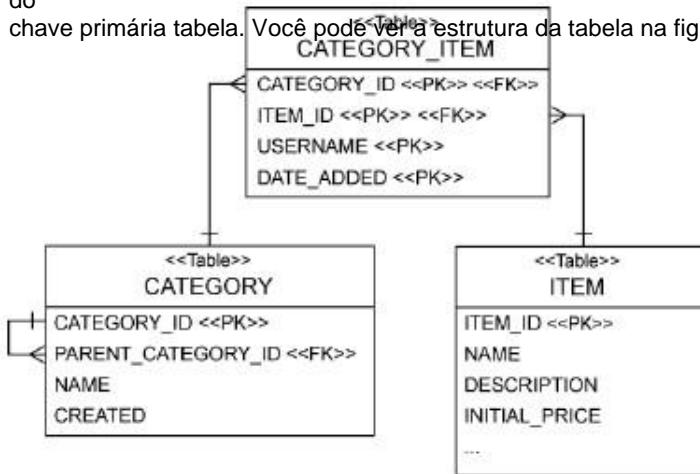


Figura 6.11 Muitos-para-muitos tabela de associação de entidade usando um componente

Na verdade, ao invés de apenas o mapeamento Nome de Usuário, Gostaríamos de manter uma referência real para o

Usuário objeto. Neste caso, temos o seguinte associação ternária mapeamento:

```
<set name="Items" lazy="true" table="CATEGORY_ITEMS">
  <key column="CATEGORY_ID"/>
  class="CategorizedItem" > composite-element
    <parent name="Category"/>
    <Many-to-one name = "Item"
      class = "Item"
      coluna = "item_id"
      not-null = "true" />
      <Many-to-one name = "User"
        class = "User"
        coluna = "USER_ID"
        not-null = "true" />
        <property name="DateAdded" column="DATE_ADDED" not-null="true"/>
      </ Elemento composto->
</ Set>
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Esta é uma besta bastante exóticos! Se você encontrar-se com um mapeamento de como isso, você deve perguntar se poderia ser melhor para mapear CategorizedItem como uma classe de entidade e usar dois um-para-muitos. Além disso, não há maneira de fazer esse mapeamento bidirecional: um componente, tais como CategorizedItem não pode, por definição, têm referências comuns. Você não pode navegar de Item para CategorizedItem.

Falamos sobre algumas limitações de muitos-para-muitos mapeamentos na seção anterior. Um deles, a restrição às coleções não indexada para o fim inverso de uma associação, também se aplica a one-to-muitas associações, se eles são bidirecionais. Vamos dar uma olhada em um-para-muitos e muitos-para-um novamente, para refrescar sua memória e elaborar sobre o que discutimos no capítulo 4.

Um-para-muitos

Você já sabe mais do que você precisa saber sobre associações um-para-muitos do capítulo 3.

Nós mapeamos uma relação pai / filho típico entre duas classes persistentes da entidade, Item e Oferta.

Esta era uma associação bidirecional, através de um <one-to-many> e um <many-to-one> mapeamento. O End "muitos" desta associação foi implementado em C # com um ISET, Tínhamos uma coleção de Lances em O Item classe. Vamos reconsiderar esse mapeamento e percorrer alguns casos especiais.

Usando um saco com semântica definida

Por exemplo, se você absolutamente precisa de uma IList de crianças em seus pais de classe C #, é possível usar um <bag> mapeamento no lugar de uma set. No nosso exemplo, primeiro temos que substituir o tipo de Lances coleção no Item classe persistente com um IList. O mapeamento para a associação entre Item e Oferta é então deixado essencialmente inalterada:

```
Class <
    name = "Oferta"
    table = "BID">
    ...
<Many-to-one
    name = "Item"
    coluna = "item_id"
    class = "Item"
    not-null = "true" />

</ Class>

Class <
    name = "Item"
    table = "ITEM">
    ...
Saco de <
    name = "Licitações"
    inverse = "true"
    cascade = "all-delete-orphan">

    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
<Saco />

</ Class>
```

Mudamos o nome da <set> elemento para <bag>, Fazendo com que nenhuma outra alteração. Note, no entanto, que mudar essa não é útil: a estrutura da tabela não suporta duplicatas, de modo que o <bag> resultados de mapeamento em uma associação com a semântica definida. Alguns gostos preferem o uso de ILists mesmo para associações com o conjunto semântica, mas a nossa não, por isso recomendamos o uso de <set> mapeamentos para pai típico / criança relacionamentos.

A solução óbvia, mas errado seria a utilização de um real <list> mapeamento para o Lances com um coluna adicional mantendo a posição dos elementos. Lembre-se que a limitação NHibernate introduzido no início deste capítulo: você não pode usar coleções indexadas em um inverso lado de um

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

associação. O inverse = "true" lado da associação não é considerado quando se salva o NHibernate estado do objeto, de modo NHibernate irá ignorar o índice dos elementos e não atualizar a coluna posição.

Entretanto, se sua relação pai / filho só será unidirecional, onde a navegação é apenas possível a partir de pai para filho, você poderia até usar um tipo de coleção indexada porque o fim "muitos" já não seria inversa. Bons usos para unidirecional um-para-muitos são incomuns em prática, e não temos um em nosso aplicativo de leilão. Você pode se lembrar que nós começamos com O Item e Oferta mapeamento no capítulo 4, tornando-o primeiro unidirecional, mas rapidamente introduziu o outro lado do mapeamento.

Vamos encontrar um exemplo diferente para implementar uma associação unidirecional um-para-muitos com uma coleção indexada.

Mapeamento unidirecional

Para os fins desta discussão, nós vamos supor agora que a associação entre Categoria e Item está a ser remodelado como uma associação um-para-muitos, um Item agora pertence a no máximo uma categoria e não possui uma referência para sua categoria atual. No código C #, modelo que esta como uma coleção nomeada

Itens no Categoria classe; não temos que mudar nada se não use uma coleção indexada.

Se Itens é implementado como um ISET, Usamos o seguinte mapeamento:

```
<set name="Items" lazy="true">
  <key column="CATEGORY_ID"/>
  <one-to-many class="Item"/>
</ Set>
```

Lembre-se que um-para-muitos mapeamentos associação não precisa declarar um nome de tabela. NHibernate já sabe que os nomes das colunas no mapeamento da coleção (neste caso, apenas Category_id) pertencem ao ITEM mesa. A estrutura da tabela é mostrado na figura 6.12.

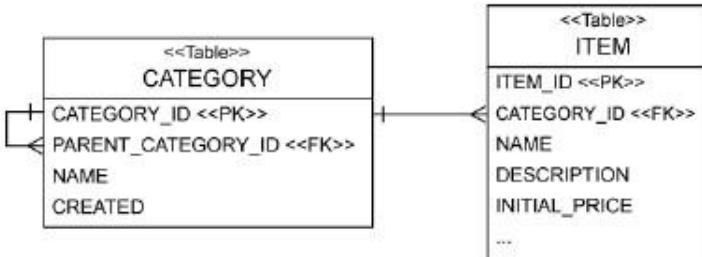


Figura 6.12 Um padrão um-para-muitos associação usando uma coluna de chave estrangeira

O outro lado da associação, o Item classe, não tem referência de mapeamento para Categoria. Podemos agora também usar uma coleção indexada na Categoria. Por exemplo, depois mudamos o Itens propriedade para Lista:

```
<enumerar name="Items" lazy="true">
  <key>
    <column name="CATEGORY_ID" not-null="false"/>
  </ Key>
  <index column="DISPLAY_POSITION"/>
  <one-to-many class="Item"/>
</ List>
```

Observe o novo DISPLAY_POSITION coluna na ITEM tabela, que ocupa a posição do Item elementos na coleção.

Há uma questão importante a considerar, que, em nossa experiência, puzzles muitos usuários NHibernate em primeiro lugar. Em uma associação unidirecional um-para-muitos, a coluna de chave estrangeira Category_id no ITEM devem ser anuláveis. Um Item poderiam ser salvas sem saber nada sobre um Categoria-It's a-pé

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

só entidade! Este é um modelo consistente e mapeamento, e você pode ter que pensar duas vezes se você lidar com uma não-nulo chave estrangeira e uma relação pai / filho. Usando uma associação bidirecional (E um Conjunto) É a solução correta.

Agora que você sabe sobre todas as técnicas de mapeamento de associação para entidades normal, você pode querer a considerar a herança, como é que todas estas associações, que trabalham entre os vários níveis de uma hierarquia? Claro, o que realmente queremos é polimórficos comportamento, então vamos ver como lida NHibernate com as associações entidade polimórficas.

6.4 Mapeamento de associações polimórficas

Polimorfismo é uma característica definidora de linguagens orientadas a objeto como C#. Portanto, o apoio à associações polimórficas e consultas é um requisito fundamental de uma solução de ORM como NHibernate. Surpreendentemente, conseguimos chegar até aqui sem a necessidade de falar muito sobre polimorfismo. Ainda mais surpreendente, não há muito a dizer sobre o tema polimorfismo é tão fácil de usar em NHibernate que não precisa gastar muito esforço explicando esse recurso.

Para lhe dar uma boa visão geral de como associações polimórficas são usados, vamos considerar primeiro um many-to-one associação a uma classe que pode ter subclasses. Neste caso, garante que NHibernate você pode criar links para qualquer instância da subclasse como faria para instâncias da classe base. Depois disso, nós também vamos guiá-lo através da criação de coleções polimórficas, e depois ir para explicar as questões específicas com a "tabela por classe concreta" de mapeamento.

6.4.1 Polymorphic many-to-one associações

Aassociação polimórfica é uma associação que pode se referir a instâncias de uma subclasse, onde o pai classe foi explicitamente especificado nos metadados de mapeamento. Para este exemplo, imagine que não temos

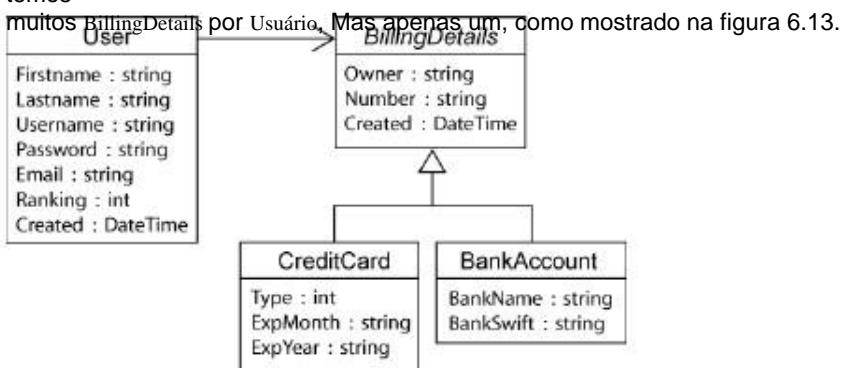


Figura 6.13 O usuário tem apenas um objeto informações de faturamento.

O usuário precisa de uma associação unidirecional para alguns BillingDetails, Que pode ser CreditCard detalhes ou BankAccount detalhes. Mapeamos esta associação para a classe abstrata BillingDetails como seguinte forma:

```

<Many-to-one name = "BillingDetails"
  class = "BillingDetails"
  coluna = "BILLING_DETAILS_ID"
  cascade = "save-update" />

```

Mas desde que BillingDetails é abstrata, a associação deve se referir a uma instância de um dos seus subclasses-CreditCard OU BankAccount-Em tempo de execução.

Todos os mapeamentos associação nós introduzimos até agora neste capítulo polimorfismo apoio. Você não precisa fazer nada especial para utilizar associações polimórficas em NHibernate - basta especificar o

nome de qualquer classe mapeada persistente em seu mapeamento da associação. Então, se essa classe declara qualquer

<subclass> OU <joined-subclass> elementos, a associação é naturalmente polimórficas.

O código a seguir demonstra a criação de uma associação para uma instância do CreditCard subclasse:

```
CreditCard cc = new CreditCard ();
cc.Number = ccNumber;
cc.Type = ccType;
cc.ExpiryDate = ccExpiryDate;

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction ()) {

    Usuário user = (User) session.get (typeof (User), uid);
    user.BillingDetails cc =;
    session.Transaction.Commit ();

}
```

Agora, quando navegar pela associação de uma segunda operação, NHibernate recupera automaticamente o CreditCard exemplo:

```
usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction ()) {

    Usuário user = (User) session.get (typeof (User), uid);
    user.BillingDetails.Pay (paymentAmount);
    session.Transaction.Commit ();

}
```

Note que, quando user.BillingDetails.Pay (paymentAmount), A chamada é contra o apropriado subclasse.

Há uma coisa que atente para: se BillingDetails foi mapeada com lazy = "true", NHibernate proxy teria o BillingDetails associação. Neste caso, não seria capaz de realizar um typecast para a classe de concreto CreditCard em tempo de execução, e até mesmo o é operador teria se comportar estranhamente:

```
Usuário user = (User) session.get (typeof (User), uid);
BillingDetails user.BillingDetails bd =;
Assert.IsFalse (bd é CreditCard);
CreditCard cc = (CreditCard) bd;
```

Neste código, o typecast na última linha falha porque bd é uma instância do proxy e quando criá-la, NHibernate ainda não sabe que é um CreditCard; Tudo o que sabe é que ele é um BillingDetails. Quando um método é chamado no proxy, a chamada é delegada a uma instância do CreditCard que é trazida de forma tardia. Para executar um typecast proxy-seguro, use Session.load ():

```
Usuário user = (User) session.get (typeof (User), uid);
BillingDetails user.BillingDetails bd =;
CreditCard cc =
    (CreditCard) Session.load (typeof (CreditCard), bd.Id);
expiryDate = cc.ExpiryDate;
```

Após a chamada para carregar, bd e cc referem-se a duas instâncias diferentes de proxy, que ambos delegar o mesmo subjacente CreditCard exemplo. Além disso, como casos de proxy foram criados, não acerto banco de dados foi incorridos ainda.

Note que você pode evitar esses problemas, evitando a busca preguiçosa, como no seguinte código, usando um

consulta técnica discutida no próximo capítulo:

```
Usuário user = (User) session.CreateCriteria (typeof (User))
    . Add (Expression.Expression.Eq ("id", uid))
    . SetFetchMode ("BillingDetails", FetchMode.Eager)
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        . UniqueResult ();
CreditCard cc = (CreditCard) user.BillingDetails;
expiryDate = cc.ExpiryDate;

```

Assim, BillingDetails terá sido buscado ansiosamente, neste caso, evitando assim a carga preguiçoso. Verdadeiramente orientada a objetos de código não deve usar é ou typecasts numerosos. Se você se encontra em execução problemas com proxies, você deve questionar seu projeto, perguntando se existe um mais polimórficos abordagem.

One-to-one associações são tratados da mesma maneira. E sobre associações de muitos valores?

6.4.2 coleções polimórfica

Vamos refatorar o exemplo anterior para sua forma original, introduzido em nosso CaveatEmptor aplicação. Se Usuário possui muitos BillingDetails, Usamos uma bidirecional um-para-muitos. Em BillingDetails, Temos o seguinte:

```

<Many-to-one name = "User"
  class = "User"
  coluna = "USER_ID" />

```

No Usuário mapeamento s, temos o seguinte:

```

<Nome do conjunto = "BillingDetails"
  lazy = "true"
  cascade = "save-update"
  inverse = "true">
<key column="USER_ID"/>
<one-to-many class="BillingDetails"/>
</ Set>

```

Adicionando um CreditCard é fácil:

```

CreditCard cc = new CreditCard ();
cc.Number = ccNumber;
cc.Type = ccType;
cc.ExpiryDate = ccExpiryDate;

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction ()) {

    Usuário user = (User) session.get (typeof (User), uid);
    user.AddBillingDetails (cc);
    session.Transaction.Commit ();

}

```

Como de costume, o nosso user.AddBillingDetails (cc) função irá garantir que a associação está fixado em ambas as extremidades chamando BillingDetails.Add (cc) e cc.User = esta.

Podemos iterar sobre a recolha e lidar com casos de CreditCard e BankAccount:

```

usando (sessão ISession sessionFactory.openSession = ())
usando (session.BeginTransaction ()) {

    Usuário user = (User) session.get (typeof (User), uid);
    foreach (BillingDetails bd em user.BillingDetails) {
        bd.Pay (ccPaymentAmount);
        session.Transaction.Commit ();

}

```

Note-se que, bd.Pay (...) será um convite a adequada BillingDetails exemplo subclasse. No exemplos até agora, temos assumido que BillingDetails é uma classe mapeada explicitamente no NHibernate documento de mapeamento, e que a estratégia de mapeamento de herança é table-per-hierarquia ou table-per-subclasse. Ainda não considerou o caso de uma estratégia de mapeamento table-per-concrete-class, onde BillingDetails não seria mencionado explicitamente no arquivo de mapeamento, mas somente na definição # C das subclasses.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

6.4.3 associações polimórfica e table-per-concrete-class

Na seção 3.7.1, "Tabela por classe concreta", definimos a table-per-concrete-class estratégia de mapeamento e observaram que esta estratégia de mapeamento torna difícil para representar uma associação polimórfica, porque você não pode mapear uma relação de chave estrangeira para a tabela da classe base abstrata. Não há tabela para a classe base com esta estratégia, você só tem mesas para as classes de concreto.

Suponhamos que queremos representar uma associação de muitos-para-um polimórfico de Usuário para BillingDetails, onde o BillingDetails hierarquia de classe é mapeada usando esta table-per-concrete-estratégia de classe. Há um CREDIT_CARD mesa e uma BANK_ACCOUNT tabela, mas não BILLING_DETAILS mesa. Precisamos de dois pedaços de informação no USUÁRIO tabela para identificar o associado CreditCard ou BankAccount:

- O nome da tabela na qual a instância associada reside
- O identificador da instância associada

O USUÁRIO tabela requer a adição de um BILLING_DETAILS_TYPE coluna, além da BILLING_DETAILS_ID. Nós usamos um NHibernate <any> elemento para mapear esta associação:

```
<Qualquer nome = "BillingDetails"
  meta-type = "String"
  id-type = "Int32"
  cascade = "save-update">
  <meta-value value="CRÉDIT CARD" class="CreditCard"/>
  <meta-value value="BANK ACCOUNT" class="BankAccount"/>
  <column name="BILLING DETAILS TYPE"/>
  <column name="BILLING DETAILS ID"/>
</ Qualquer>
```

O meta-type atributo especifica o tipo do NHibernate BILLING_DETAILS_TYPE coluna, o id-type atributo especifica o tipo de BILLING_DETAILS_ID coluna (CreditCard e BankAccount deve ter o tipo mesmo identificador). Note que a ordem do <column> elementos é importantes: primeiro o tipo, em seguida, o identificador.

O <meta-value> elementos dizer NHibernate como interpretar o valor da BILLING_DETAILS_TYPE coluna. Podemos usar qualquer valor que gostamos como um tipo discriminador. Por exemplo, podemos codificar as informações em dois personagens:

```
<Qualquer nome = "BillingDetails"
  meta-type = "String"
  id-type = "Int32"
  cascade = "save-update">
  <meta-value value="CC" class="CreditCard"/>
  <meta-value value="CA" class="BankAccount"/>
  <column name="BILLING DETAILS TYPE"/>
  <column name="BILLING DETAILS ID"/>
</ Qualquer>
```

Na verdade, a <meta-value> elementos são opcionais, se você omitti-los, irá utilizar o NHibernate totalmente nomes qualificados das classes. Um exemplo desta estrutura de tabela é mostrado na figura 6.14.

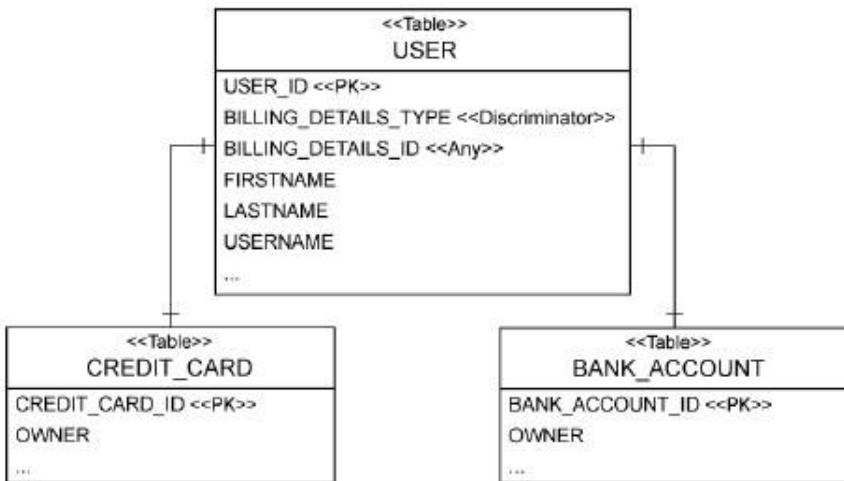


Figura 6.14 Usando uma coluna discriminadora com um qualquer associação

Aqui está o primeiro grande problema com esse tipo de associação: não podemos adicionar uma restrição de chave estrangeira para o `BILLING_DETAILS_ID` coluna, uma vez que alguns valores referem-se ao `BANK_ACCOUNT` mesa e outros para o `CREDIT_CARD` mesa. Assim, precisamos de chegar a alguma outra forma para garantir a integridade. Um banco de dados gatilho pode ser uma maneira de conseguir isso.

Além disso, é difícil escrever SQL tabela junta para esta associação. Em particular, o NHibernate facilidades de consulta não suportam este tipo de mapeamento da associação, nem pode esta associação ser buscado usando uma associação externa. Nós desencorajar o uso de `<any>` associações para todos, mas os casos mais especiais.

Como você pode ver, o polimorfismo é mais confusa no caso de uma herança table-per-concrete-class mapeamento de estratégia. Não costumo usar esta estratégia de mapeamento quando as associações polimórficas são necessário. Enquanto se mantiver no outro herança mapeamento de estratégias, polimorfismo é simples, e você geralmente não precisa pensar sobre isso.

Este capítulo abordou os pontos mais delicados da ORM e técnicas, por vezes, necessários para resolver o problema de incompatibilidade estrutural. Podemos agora totalmente mapa todas as entidades e associações na CaveatEmptor modelo de domínio.

Discussimos também o sistema de tipo NHibernate, que distingue entidades a partir de tipos de valor. Um instância da entidade tem o seu próprio ciclo de vida e identidade persistente, uma instância de um tipo de valor é completamente dependente de uma entidade proprietária.

NHibernate define uma rica variedade de tipos internos mapeamento NET valor que a ponte entre os dois tipos e tipos SQL. Quando estes tipos predefinidos não forem suficientes, você pode facilmente estender-los usando tipos personalizados ou mapeamentos de componentes e até mesmo implementar conversões arbitrárias de .NET para SQL tipos de dados.

Coleção de valores de propriedades são consideradas de tipo de valor. A coleção não tem a sua própria identidade persistente e pertence a uma única entidade proprietária. Você viu como mapear coleções, incluindo coleções de valor digitado instâncias e associações de muitos valores entidade.

NHibernate suporta um-para-um, um-para-muitos e muitos-para-muitos associação entre entidades. Em prática, recomendamos contra o uso excessivo de muitos-para-muitos. Associações em NHibernate são naturalmente polimórficas. Também falamos sobre o comportamento bidirecional de tal relacionamentos.

Tendo coberto técnicas de mapeamento em grande detalhe, o próximo capítulo vai agora afastar-se do tópico de mapeamento, e vire para o tema da recuperação de objetos. Esta baseia-se nos conceitos introduzidos em capítulo 4, apenas dando informações mais detalhadas que lhe permite construir mais eficiente e flexível

consultas. Também vamos cobrir alguns dos tópicos mais avançados que envolvem recuperação de objetos, tais como consultas de relatório, buscar associações e cache.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



Recuperação de objetos de forma eficiente

As consultas são a parte mais interessante de escrever código bom acesso de dados. A consulta complexa pode exigir uma tempo para acertar, e seu impacto sobre o desempenho de um aplicativo pode ser enorme. Tal como acontece com SQL regular, escrever consultas NHibernate se torna muito mais fácil com a experiência.

Se você estiver usando SQL manuscrita para um número de anos, você pode estar preocupado que ORM vai tirar um pouco da expressividade e flexibilidade que você está acostumado. Este é raramente o caso; Instalações poderosa consulta do NHibernate permitem fazer quase qualquer coisa que você faria em SQL, e em alguns casos mais. Para os casos raros em que você não pode fazer instalações NHibernates 'própria consulta do exatamente o que você quer, NHibernate permite recuperar objetos usando o dialeto SQL nativo de seu banco de dados.

No capítulo 4, secção 4.4, mencionamos que existem três maneiras de expressar consultas em NHibernate. Primeira é a HQL:

```
session.CreateQuery ("da categoria c, onde c.Name como 'Laptop %'");
```

Em seguida é a ICriteria API:

```
session.CreateCriteria (typeof (Categoria))
    . Add (Expression.Like ("Nome", "% Laptop"));
```

Finalmente, há SQL direto, que mapeia automaticamente conjuntos de resultados para objetos:

```
session.CreateSQLQuery (
    "Select {*} de c. CATEGORIA {c} onde NAME como 'Laptop%', 
    "C",
    typeof (Categoria));
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Este capítulo aborda em profundidade utilizando técnicas de todos os três métodos. Você também pode usar este capítulo como um referência, portanto, algumas seções são escritos em um estilo menos detalhada, mas mostram muitos exemplos de código pequena para casos de uso diferentes.

Vamos começar nossa explorarion mostrando como as consultas são realmente executado com NHibernate.

Ou

dito de outra forma, ao invés de focar-se as consultas, vamos focar as várias técnicas para realmente criar e executar consultas usando NHibernate. Depois disso, nós vamos passar a discutir as indicações de como as consultas são compostas.

7.1 consultas Executora

O IQuery e ICriteria interfaces tanto define vários métodos para controlar a execução de um consulta. Para executar uma consulta em seu aplicativo, você precisa obter uma instância de um desses consulta interfaces usando o ISession. Vamos dar uma rápida olhada em como você pode fazer isso.

7.1.1 As interfaces de consulta

Para criar um novo IQuery exemplo, chamar o CreateQuery () ou CreateSQLQuery ().

IQuery pode ser

usados para preparar uma consulta HQL como segue:

```
EuConsulta hqlQuery session.CreateQuery = ("from User");
```

Esta consulta está agora configurado para buscar todos os objetos de usuário em bancos de dados. Nós também podemos conseguir a mesma coisa usando o CreateSQLQuery ()método usando o dialeto SQL nativo do banco de dados subjacente:

```
EuSQLQuery consulta session.CreateSQLQuery = (
    "Select {*} de u. USUÁRIOS {u}", "u",
    typeof (User));
```

Você aprenderá mais sobre como executar consultas SQL na seção 8.5.4, "consultas SQL Native". Finalmente, aqui está

como nós usaria a rigidez ICriteria interface para fazer a mesma coisa de uma maneira diferente:

```
EuCriterios crit = session.CreateCriteria (typeof (User));
```

Este último exemplo usa CreateCriteria ()para obter uma lista de objetos de volta. Você vai notar que o raiz entidade tipo que nós queremos que a consulta de retorno é especificado como Usuário. Vamos estudar consultas critérios em pormenor mais tarde

, e agora continuar a nossa discussão sobre a criação e execução de consultas, olhando para outro útil conceito - a paginação.

Paginação do resultado

Paginação é uma técnica comumente usada, e você provavelmente já viu em ação várias vezes. Para exemplo, um site de comércio eletrônico pode exibir listas de produtos ao longo de um número de páginas, cada uma mostrando

apenas 10 ou 20 produtos de cada vez. Normalmente, os usuários navegar para páginas seguinte ou anterior clicando

links apropriados na página. Ao escrever código de acesso de dados para este cenário, os desenvolvedores precisavam

trabalhar para fora como para mostrar a página correta de registros em um determinado momento - e é isso que a paginação é tudo sobre.

Em NHibernate, tanto o IQuery e ICriteria interfaces de fazer a paginação simples, como demonstrado

```
EuQuery =
    session.CreateQuery ("da ordem do usuário u por u.Name asc");
query.SetFirstResult (0);
query.SetMaxResults (10);
```

A chamada para SetMaxResults (10) limita o resultado da consulta definida para os primeiros 10 objetos selecionados pelo banco de dados. E se quiséssemos obter alguns resultados para a próxima página?

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

ICriteria crit = session.CreateCriteria (typeof (User));
crit.AddOrder (Order.Asc ("Nome"));
crit.SetFirstResult (10);
crit.SetMaxResults (10);
IList <User> results = crit.List <User> ();

```

A partir do objeto 10, nós recuperamos os próximos 10 objetos. Note que não há nenhuma maneira padrão de expressar a paginação no SQL, e cada fornecedor de banco de dados muitas vezes fornece uma sintaxe diferente e abordagem.

Felizmente, NHibernate sabe os truques para cada fornecedor, de modo de paginação é muito fácil de fazer, independentemente da seu banco de dados o seu particular.

IQuery e ICriteria também expõe um fluent interface que permite método de encadeamento. Para demonstrar, Reescrevemos os dois exemplos anteriores para tirar proveito disso:

```

<User> Resultados IList =
    session.CreateQuery ("da ordem do usuário u por u.Name asc")
        . SetFirstResult (0)
        . SetMaxResults (10)
        . <User> List ();

<User> Resultados IList =
    session.CreateCriteria (typeof (User))
        . AddOrder (Order.Asc ("Nome"))
        . SetFirstResult (40)
        . SetMaxResults (20)
        . <User> List ();

```

Chamadas encadeamento de método desta maneira é considerado menos detalhado e mais fácil de escrever, e é possível

a ver com muitas das APIs NHibernate's .

Agora que nós criamos consultas e configurar a paginação, veremos como você ir sobre começar a resultados de uma consulta.

Listagem e iterando resultados

O List () método executa a consulta e retorna os resultados como uma lista:

```
IList <User> resultado = session.CreateQuery ("from User") <User> List ();
```

Ao escrever consultas, às vezes queremos apenas uma única instância a ser retornado. Por exemplo, se nós queria encontrar o lance mais alto, podemos obter essa instância lendo-o da lista de resultados pelo índice: resultado [0]. Alternativamente, nós poderíamos usar SetMaxResults (1)E executar a consulta com o UniqueResult () método:

```

Licitação maxBid =
    (Bid) session.CreateQuery ("da ordem de b Licitação por b.Amount desc")
        . SetMaxResults (1)
        . UniqueResult ();

Licitação Licitação = (Bid) session.CreateCriteria (typeof (Bid))
    . Add (Expression.Eq ("Id", id))
    . UniqueResult ();

```

Você precisa ter certeza de que sua consulta retorna apenas um objeto, caso contrário uma exceção será lançada.

O IQuery e ISession interfaces também fornecer uma Enumeráveis () método, que retorna a mesmo resultado que List () ou Find (), mas que usa uma estratégia diferente para recuperar os resultados. Quando você usa Enumeráveis () para executar uma consulta, NHibernate recupera somente a chave primária (identificador) valores nas primeiras SQL select, que tenta encontrar o resto do estado dos objetos no cache, antes de consultando novamente para o resto dos valores de propriedade. Esta técnica pode ser usada para otimizar o carregamento em casos específicos, como discutido na seção 7.7, "Otimizando a recuperação do objeto."

Por que não usar ISession.Find () em vez de IQuery.List ()?

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O ISession API fornece métodos de atalho para consultas simples. Em vez de criar uma IQuery exemplo, você também pode ligar para ISession.Find ("from User"). O resultado é o mesmo que a partir de IQuery.List (). O mesmo é verdadeiro para Enumeráveis (). No entanto, essa consulta métodos de atalho na ISession API será removidas no futuro para reduzir o inchaço dos métodos de sessão. Nós recomendamos sempre usando o IQuery API.

Finalmente, outro fator importante de construção de consultas é o de parâmetros de ligação. O IQuery interface permite que você conseguir isso de uma forma flexível, como vamos agora discutir.

7.1.2 parâmetros Binding

Permitindo que os desenvolvedores para ligar valores para consultas é uma característica importante para qualquer biblioteca de acesso a dados porque lhes permite construir consultas que são de fácil manutenção e seguro. Vamos demonstrar os tipos de associação de parâmetros disponíveis em NHibernate abaixo, mas primeiro vamos olhar a os problemas potenciais de não parâmetros de ligação. O problema de ataques de injeção SQL Considerere o seguinte código:

```
cadeia queryString =
    "Do item i, onde i.Description como '" + searchString + "'";
IList resultado = session.CreateQuery (queryString) Lista ();
```

Este código é pura e simplesmente MAU! Você pode saber por quê? Simplificando, o código seria potencialmente deixe o seu aplicativo aberto para Sql Injection ataques. Este é o lugar onde um usuário mal-intencionado tenta enganar sua aplicação em execução o seu próprio SQL contra o banco de dados, para que eles possam causar danos ou circunavegar segurança do aplicativo. Se o usuário digitou o searchString abaixo:

O queryString enviadas ao banco de dados seria

```
do item i, onde i.Description como 'foo' e CallSomeStoredProcedure () e 'bar' =
    'Bar'
```

Como você pode ver, o original queryString já não seria uma simples busca por uma corda, mas também executar um procedimento armazenado no banco de dados! Um dos principais problemas aqui é que a nossa aplicação não está verificando os valores passados a partir do usuário interface. Devido a isso, as aspas não são escapou e, portanto, o usuário pode injetar sua própria SQL. Os usuários podem mesmo accidentalmente travar o aplicativo apenas colocando uma aspa simples no busca string. A regra de ouro é "Nunca passar valores unchecked da entrada do usuário ao banco de dados"! Felizmente, esse problema é facilmente evitado pelo uso de parâmetros. Com os parâmetros, a nossa consulta pode parecido com este:

```
cadeia queryString =
    "Do incisos I, onde i.Description como: searchString"
```

Quando usamos parámetros, as consultas e os parâmetros são enviados para o banco de dados separadamente, de modo que o banco de dados pode garantir que eles sejam tratados de forma segura e eficiente.

Outro motivo para usar parâmetros é que ele ajuda NHibernate para ser mais eficiente, isto é porque NHibernate acompanha as consultas que você executar. Quando os parâmetros são usados ele só precisa mantém

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

um cópia da consulta na memória, mesmo que seja executado milhares de vezes com parâmetros diferentes, cada tempo.

Então, agora nós entendemos a importância de parâmetros, a próxima pergunta é como podemos usá-los em nossas consultas NHibernate? Existem duas abordagens para parâmetro de ligação: parâmetros nomeados e parâmetros posicionais. Discutimos estes, por sua vez abaixo.

Usando os parâmetros nomeados

Usando parâmetros nomeados, podemos reescrever a consulta como

```
cadeia queryString =  
    "A partir do item Item onde item.Description como: searchString";
```

Os dois pontos seguido por um nome de parâmetro indica um parâmetro nomeado. Então, podemos usar o IQuery interface para ligar um valor para o searchString parâmetro:

```
Resultado IList session.CreateQuery = (queryString)  
    . SetString ("searchString", searchString)  
    . List ();
```

Porque searchString é um user-supplied variável string, usamos o SetString () método da IQuery interface para vinculá-lo ao parâmetro nomeado (searchString). Este código é mais limpo, mais seguro, e executa melhor, porque uma única instrução SQL compilada pode ser reutilizado se vinculam apenas os parâmetros mudar.

Muitas vezes, você vai precisar de vários parâmetros:

```
cadeia queryString = @ "do item Item  
    onde item.Description como: searchString  
    e item.Date >: minDate";
```

```
Resultado IList session.CreateQuery = (queryString)  
    . SetString ("searchString", searchString)  
    . SetDate ("minDate", minDate)  
    . List ();
```

Usando parâmetros de posição

Se preferir, você pode usar parâmetros de posição:

```
cadeia queryString = @ "do item Item  
    onde item.Description como?  
    e item.Date > ?";
```

```
Resultado IList session.createQuery = (queryString)  
    . SetString (0, searchString)  
    . SetDate (1, minDate)  
    . List ();
```

Não só isto é menos código auto-documentado que a alternativa com parâmetros nomeados, também é muito mais vulneráveis à quebra fácil se nós mudamos um pouco a string de consulta:

```
cadeia queryString = @ "do item Item  
    onde item.Date >?  
    e item.Description como ?";
```

Cada mudança de posição dos parâmetros bind requer uma mudança para o código do parâmetro de ligação. Isto leva a código frágil e manutenção intensiva. Recomendamos que você evite posicional parâmetros.

Passado, um parâmetro nomeado pode aparecer várias vezes na cadeia de consulta:

```
cadeia userSearch =  
    @ "A partir de u usuário onde u.Username como: searchString  
    ou u.Email como: searchString ";
```

```
Resultado IList session.CreateQuery = (userSearch)
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
    . SetString ("searchString", searchString)
    . List ();
```

Vinculativo argumentos arbitrários

Nós usamos `SetString()` e `SetDate()` para vincular argumentos para parâmetros de consulta. O `IQuery` interface fornece métodos de conveniência argumentos semelhantes para ligação da maioria dos NHibernate built-in tipos: desde `SetInt32()` para `SetTimestamp()` e `SetEnum()`.

Um método é particularmente útil `SetEntity()`, Que lhe permite ligar uma entidade persistente:

```
session.CreateQuery ("from Item item = onde item.Seller: vendedor")
    . SetEntity ("vendedor", vendedora)
    . List ();
```

No entanto, há também um método genérico que permite ligar um argumento de qualquer tipo NHibernate:

```
cadeia queryString = @ "do item Item
    onde item.Seller =: vendedor e
        item.Description como: desc ";

session.CreateQuery (queryString)
    . SetParameter ("vendedor", vendedor,
        NHibernateUtil.Entity (typeof (User)))
    . SetParameter ("desc", descrição, NHibernateUtil.String)
    . List ();
```

Isso funciona mesmo para o costume tipos definidos pelo usuário como

```
MonetaryAmount:
IQuery q =
    session.CreateQuery ("Oferta de lance onde> bid.Amount: quantidade");
q.setParameter ("quantidade",
    givenAmount,
    NHibernateUtil.Custom (typeof (MonetaryAmountUserType)));
IList <Bid> resultado = <Bid> q.List ();
```

Para alguns tipos de parâmetros, é possível adivinhar o tipo NHibernate da classe do parâmetro valor. Neste caso, você não precisa especificar o tipo NHibernate explicitamente:

```
cadeia queryString = @ "do item Item
    onde item.Seller =: vendedor e
        item.Description como: desc ";

session.CreateQuery (queryString)
    . SetParameter ("vendedor", vendedora)
    . SetParameter ("desc", descrição)
    . List ();
```

Como você pode ver, ele ainda trabalha com entidades, tais como `vendedor`. Esta abordagem funciona muito bem para corda, `int` ou `bool` parâmetros, por exemplo, mas não tão bem para `DateTime`, Onde o tipo NHibernate pode ser `Timestamp` ou `DateTime`. Nesse caso, você tem que usar o método adequado de ligação ou explicitamente usar `NHibernateUtil.DateTime` (Ou qualquer tipo NHibernate outros) como o terceiro argumento para `SetParameter()`.

Se tivéssemos um POCO com `Vendedor` e `Descrição` propriedades, podemos usar a `SetProperties()` método para vincular os parâmetros de consulta. Por exemplo, poderíamos passar os parâmetros de consulta em uma instância do

O Item classe:

```
Item Item item = new ();
item.Seller seller =;
item.Description = descrição;
```

```
cadeia queryString = @ "do item Item
    onde item.Seller =: vendedor e
        item.Description como: desc ";
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
. session.CreateQuery (queryString) SetProperties (item) List () ;
```

SetProperties () partidas, os nomes das propriedades POCO a parâmetros nomeados na cadeia de consulta, utilização SetParameter () para adivinhar o tipo NHibernate e vincular o valor. Na prática, isso acaba por ser menos útil do que parece, uma vez que alguns tipos comuns NHibernate não são fáceis de adivinhar (DateTime, Em particular).

Os métodos de parâmetro obrigatório de IQuery são nulos de segurança, fazendo com que este código legal:

```
session.CreateQuery ("do usuário como u = onde u.Email: e-mail")
    . SetString ("email", null)
    . List ();
```

No entanto, o resultado deste código quase certamente não é o que pretendemos. O SQL resultante será conter uma comparação como nome = null, Que sempre é avaliada como nula no SQL lógica ternária. Em vez disso, devemos usar o é nulo operador:

```
session.CreateQuery ("do usuário como u onde u.Email é nulo") List () ;
```

Até agora, os exemplos de código HQL nós mostramos todo o uso integrado strings literais HQL query. Isto não é razoável para consultas simples, mas assim que começar a considerar consultas complexas que devem ser dividido em várias linhas, ele começa a ficar um pouco pesado.

7.1.3 Usando consultas nomeadas

Nós não gostamos de ver strings literais HQL espalhados por todo o código C # a menos que forem necessárias. NHibernate permite armazenar seqüências de consulta fora do seu código, uma técnica que é chamado consultas nomeadas.

Isto permite que você armazene todas as consultas relacionadas a uma determinada classe persistente junto com as outras metadados da classe em um arquivo de mapeamento XML. O nome da consulta é utilizada para chamá-lo a partir do aplicação.

```
O GetNamedQuery método obtém um IQuery exemplo, para uma consulta nomeada:
```

Neste exemplo, nós executamos a consulta nomeada FindItemsByDescription após a ligação uma string argumento para um parâmetro nomeado. A consulta nomeada é definido nos metadados de mapeamento, por exemplo, em

Item.hbm.xml, Usando o <query> elemento:

```
<query name="FindItemsByDescription"> <! [CDATA [
    do item Item onde item.Description como: descrição
    Consulta ]]></>
```

Consultas nomeadas não têm que ser strings HQL, eles podem até ser consultas SQL nativas e sua C # código não precisa saber a diferença:

```

<sql-query name="FindItemsByDescription"> <! [CDATA [
    selecione {*} de i. ITEM {i}, onde DESCRIÇÃO como: descrição
  ]]>
  <Voltar class="Item"/> alias="i"
</ Sql-query>

```

Isto é útil se você pensa que você pode querer otimizar suas consultas depois por o ajuste fino do SQL. É também é uma boa solução se você tiver a porta de um aplicativo de legado para NHibernate, onde o código SQL foi isolado das rotinas handcoded ADO.NET. Com consultas nomeadas, você pode facilmente porto as consultas um por um para arquivos de mapeamento.

7.1.4 Usando substituições consulta

Muitas vezes é necessário, ou pelo menos útil, para usar uma palavra diferente para nomear um objeto em uma consulta. Para exemplo, com a propriedade de um boolean como User.IsAdmin, Você vai escrever:
 Usuário de u = 1, onde u.IsAdmin

Mas, adicionando essa propriedade para seu arquivo de configuração:

```

<property name="hibernate.query.substitutions">
  true 1, 0 false
</ Property>

```

Você pode escrever:

```
Usuário de u onde u.IsAdmin = true
```

Note-se que, este recurso também pode ser usado para renomear funções SQL. Nós agora embrulhado nossa discussão sobre a criação e execução de consultas, então agora é hora de realmente focar as consultas si. A próxima seção cobre HQL, começando com consultas simples e de passar para alguns tópicos muito mais avançado.

7.2 consultas básicas para objetos

Vamos começar com perguntas simples para se familiarizar com a sintaxe e semântica HQL. Embora nós mostrar a alternativa critérios para a maioria das consultas HQL, tenha em mente que HQL é a abordagem preferida para consultas complexas. Normalmente, os critérios podem ser derivados se você sabe o equivalente HQL, é muito mais difícil o contrário.

NOTA
 Testes de consultas-NHibernate Você pode usar o open source ferramenta Query Analyzer NHibernate NHibernate para executar consultas ad hoc. Ele permite que você selecione documentos de mapeamento NHibernate (ou escrevê-los), definição da configuração NHibernate, e depois ver o resultado de consultas HQL você tipo de forma interativa. Mais detalhes são fornecidos na seção 7.6.4.

7.2.1 A simples consulta

A mais simples consulta recupera todas as instâncias de uma determinada classe persistente. Em HQL, é parecido com este:
 de Licitação

Usando a interface ICriteria, é parecido com este:

```
ICriteria c = session.CreateCriteria (typeof (Bid));
```

Ambos geraria o seguinte SQL por trás das cenas:

```
selecionar B. BID ID, O VALOR B., B. item id, B. CRIADO de B BID
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Mesmo para este caso simples, você pode ver que HQL é menos detalhado do que SQL.

7.2.2 Usando aliases

Quando você consulta uma classe usando HQL muitas vezes você precisa atribuir um pseudônimo para a classe consultados, que você

usar como referência em outras partes da consulta:

```
de Licitacao como  
licitacao
```

O como palavra-chave é sempre opcional. A seguir é equivalente:

```
Oferta de lance
```

Pense nisso como sendo um pouco como a declaração de variável temporária no seguinte código C #:

```
for (int i = 0; i < allQueriedBids.Count; i++) {  
    Licitacao Licitacao = (Bid) allQueriedBids [i];  
    ...  
}
```

Vamos atribuir o alias `oferta` a instâncias consultadas da `Oferta` classe, permitindo-nos a referir a sua propriedade valores no final do código (ou consulta). Para lembrar-se da semelhança, recomendamos que você use a mesma convenção para aliases que você usa para variáveis temporárias (camelCase, geralmente).

No entanto, usamos apelidos mais curtos em alguns dos exemplos deste livro (por exemplo, `Euem` vez de `item`) para manter o código impresso legível.

NOTA

Nós nunca escrever palavras-chave HQL em letras maiúsculas; nós nunca escrever palavras-chave SQL em maiúsculas

quer. Parece feio e antiquado, a maioria dos terminais modernos pode exibir maiúsculas e caracteres minúsculos. No entanto, HQL não diferencia maiúsculas de minúsculas para palavras-chave, assim você pode escrever

`FROM Bid AS lance se você gosta de gritar.`

Pelo contrário, uma consulta define um alias critérios implícitos. A entidade raiz em uma consulta de critérios é sempre

atribuído o alias `este`. Discutimos esse assunto em mais detalhes posteriormente, quando estamos juntando associações

com consultas critérios. Você não tem que pensar muito sobre aliases ao usar o `ICriteria API`.

7.2.3 consultas polimórfica

Descrevemos HQL como uma linguagem de consulta orientada a objeto, por isso devem apoiar consultas polimórficas

isto é, as consultas para instâncias de uma classe e todas as instâncias de suas subclasses, respectivamente. Você já

sabe o suficiente HQL que podemos demonstrar isso. Considere a seguinte consulta:

Esta consulta retorna objetos do tipo `BillingDetails`, Que é uma classe abstrata. Assim, neste caso, os objetos concretos são dos subtipos de `BillingDetails:CreditCard` e `BankAccount`. Se apenas quer instâncias de uma subclasse particular, podemos usar

```
de cartao de credito
```

A classe chamada no a partir de cláusula não precisa ser uma classe mapeada persistente; qualquer classe vai fazer.

A consulta a seguir retorna todos os objetos persistentes no banco de dados inteiro:

```
de System.Object
```

Claro, isso também funciona para interfaces-esta consulta retorna todos os objetos serializable persistente (Na verdade, apenas aqueles que implementa a interface `ISerializable`):

```
de System.ISerializable
```

Consultas critérios também suportam polimorfismo:

```
session.CreateCriteria (typeof (BillingDetails)) Lista (..);
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Esta consulta retorna instâncias de BillingDetails e suas subclasses. Da mesma forma, os seguintes critérios consulta retorna todos os objetos persistentes:

```
session.CreateCriteria (typeof (System.Object)) Lista () .;
```

Polimorfismo não se aplica apenas às classes chamado explicitamente no a partir de cláusula, mas também para polimórficos

associações, como você verá mais tarde.

Agora que nós discutimos a a partir de cláusula, vamos passar para as outras partes do HQL.

7.2.4 Restrição

Nós normalmente não deseja recuperar todas as instâncias de uma classe quando executar uma consulta. Em vez disso, nós queremos expressar algumas restrições sobre os valores de propriedade de nossos objetos, portanto, apenas um subconjunto de objetos é recuperados. Isso é chamado restrição, e em ambos os HQL e SQL, a restrição é conseguido usando a onde cláusula. A onde cláusula pode ser simples ou complexa, mas vamos começar com um exemplo simples HQL:

Observe que a restrição é expressa em termos de uma propriedade, E-mail, Da Usuário classe, e que nós use uma noção orientada a objeto: Assim como em C #, u.Email não pode ser abreviado para simples E-mail.

Critérios para uma consulta, nós devemos construir um ICriterion objeto para expressar a restrição. O Expressão classe fornece métodos para a fábrica de embutidos ICriterion tipos. Vamos criar a mesma consulta utilizando critérios e imediatamente executá-lo:

```
ICriterion emailEq = Expression.Eq ("Email", "foo@hibernate.org");
ICriteria crit = session.CreateCriteria (typeof (User));
crit.add (emailEq);
Usuário user = (User) crit.UniqueResult ();
```

Criamos um ICriterion exemplo, segurando a simples Expressão para uma comparação de igualdade e adicioná-lo à ICriteria. O UniqueResult () método executa a consulta e retorna exatamente um objeto como um resultado.

Geralmente, nós iria escrever isso um pouco menos com detalhes, usando encadeamento de método:

```
Usuário user = (User) session.CreateCriteria (typeof (User))
    . Add (Expression.Eq ("Email", "foo@hibernate.org"))
    . UniqueResult ();
```

O SQL gerado por essas consultas é

```
selecionar U. USER_ID, U. FIRSTNAME, U. LASTNAME, U. USERNAME, U. EMAIL
a partir de U USUÁRIO
U. onde EMAIL = 'foo@hibernate.org'
```

É comum ter uma restrição que deve sempre ser usado; na maioria das vezes, é usado para ignorar obsoleto dados. Você pode, por exemplo, têm uma Ativo propriedade e escrever:

```
seleccione Usuário u onde u.Email = 'foo@hibernate.org' e u.Active = 1
```

Mas isto é perigoso porque você pode esquecer essa restrição, uma melhor solução, neste caso, é alterar o mapeamento de:

```
<class name="User" where="ACTIVE=1">
```

Agora, você pode simplesmente escrever:
do Usuário u onde u.Email = 'foo@hibernate.org'

Esta consulta irá gerar a consulta SQL:

```
selecionar U. USER_ID, U. FIRSTNAME, U. LASTNAME, U. USERNAME, U. EMAIL
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
a partir de U USUÁRIO
U. onde EMAIL ACTIVE = 'foo@hibernate.org' e U. = 1
```

Note que a coluna ACTIVE não tem de ser mapeado. E, desde NHibernate 1.2.0, este atributo também é usado ao chamar `ISession.Load()` e `ISession.Get()`. Este recurso também está disponível para coleções:

```
<bag name="Users" where="ACTIVE=1">
```

Aqui, a coleção Usuários irá conter apenas os usuários cujos ACTIVE valor é 1. Esta abordagem pode ser útil, mas recomendamos considerando filtros para a maioria dos cenários (leia a seção 7.5.2, "Coleção filtros").

Você pode, é claro, use vários outros operadores de comparação para a restrição.

7.2.5 Os operadores de comparação

A restrição é expressa usando a lógica ternária. O onde cláusula é uma expressão lógica que avalia para true, false, null ou para cada tupla de objetos. Você construir expressões lógicas, comparando propriedades de objetos para outras propriedades ou valores literais usando HQL operadores de built-in comparação.

FAQ

Qual é a lógica ternária? Uma linha é incluída em um conjunto de resultados SQL se e somente se o onde cláusula avaliada como verdadeira. Em C #, `notNullObject == null` avaliada como falsa e `null == null` avaliada como verdadeira. Em SQL, `NOT_NULL_COLUMN = null` e `= null` nulo tanto avaliar como nula, não é verdade. Assim, SQL precisa de um operador especial, `IS NULL`, Para testar se um valor é nulo. Este lógica ternária é uma maneira de lidar com expressões que podem ser aplicados a valores nulos coluna. É A (discutível) de extensão SQL para a lógica binária familiar do modelo relacional e do típico linguagens de programação como C #.

HQL suporta os mesmos operadores básicos como SQL: `=, <>, <, >, >=, <=, entre, não entre, em` E não em. Por exemplo:

```
Oferta de lance em que bid.Amount entre 1 e 10
Oferta de lance em que bid.Amount > 100
do Usuário u onde u.Email in ('foo@hibernate.org', 'bar@hibernate.org')
```

No caso de consultas critérios, todos os mesmos operadores estão disponíveis através do Expressão classe:

```
session.CreateCriteria (typeof (Bid))
    . Add (Expression.Between ("Valor", 1, 10))
    . List ();

session.CreateCriteria (typeof (Bid))
    . Add (Expression.Gt ("Valor", 100))
    . List ();

string [] = {e-mails "foo@NHibernate.org", "bar@NHibernate.org"};
session.CreateCriteria (typeof (User))
    . Add (Expression.In ("Email", e-mails))
    . List ();
```

Porque o banco de dados subjacente implementa a lógica ternária, o teste para nulo valores requer alguns cuidados.

Lembre-se que `= null` nulo não avalia a verdadeiro no banco de dados, mas para nulo. Todas as comparações que usam o nulo operador de fato avaliar a nulo. Ambos HQL eo ICriteria Fornecer uma API-SQL estilo é nulo operador:

```
do Usuário u onde u.Email é nulo
```

Esta consulta retorna todos os usuários com nenhum endereço de e-mail. A semântica mesma disponível nos ICriteria API:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
session.CreateCriteria (typeof (User))
    . Add (Expression.IsNotNull ("Email"))
    . List ();
```

Nós também precisamos ser capazes de encontrar usuários que fazer ter um endereço de e-mail:
do Usuário u onde u.Email não é nulo

```
session.CreateCriteria (typeof (User))
    . Add (Expression.IsNotNull ("Email"))
    . List ();
```

Finalmente, o HQL onde cláusula suporta expressões aritméticas (mas o ICriteria API não):

```
Oferta de lance em que (bid.Amount / 0,71) - 100,0 > 0,0
```

Para pesquisas baseado em texto, você precisa ser capaz de realizar case-insensitive correspondência e correspondências em fragmentos de cordas em expressões restrição.

7.2.6 strings

O como operador permite pesquisas curinga, onde os símbolos são curinga %_ e, assim como em SQL:

```
Usuário de u onde u.Firstname como "% S"
```

Esta expressão restringe o resultado para os usuários com um primeiro nome que começa com um capital S. Você também pode

nega a como operador, por exemplo, usando uma expressão de correspondência substring:
a partir de u usuário onde não u.Firstname como "%% Foo S"

Critérios para consultas, pesquisas curinga pode usar os mesmos símbolos ou especificar um curinga MatchMode. NHibernate fornece a MatchMode como parte do ICriteria consulta API, nós usá-lo para escrevendo expressões de correspondência seqüência de caracteres sem manipulação de string. Essas duas consultas são equivalentes:

```
session.CreateCriteria (typeof (User))
    . Add (Expression.Like ("Nome", "% S"))
    . List ();

session.CreateCriteria (typeof (User))
    . Add (Expression.Like ("Nome", "S", MatchMode.Start))
    . List ();
```

O permitido MatchModes são Começar,Final,Em qualquer lugarE Exato.

Um recurso extremamente poderoso do HQL é a capacidade de chamar funções SQL arbitrário na onde cláusula. Se o seu banco de dados suporta funções definidas pelo usuário (a maioria), você pode colocar essa funcionalidade para todos os tipos de usos, bem ou mal. Por enquanto, vamos considerar a utilidade do padrão ANSI SQL funções upper () e inferior (). Eles podem ser usados para case-insensitive de pesquisa:

```
Usuário de u onde inferiores (u.Email) = 'foo@hibernate.org'
```

O ICriteria API não suporta chamadas de função SQL. Ele faz, contudo, fornecer um especial facilidade para case-insensitive de pesquisa:

```
session.CreateCriteria (typeof (User))
    . Add (Expression.Eq ("Email", "foo@hibernate.org"). IgnoreCase ())
    . List ();
```

Infelizmente, HQL não fornece um operador de concatenação-padrão, em vez disso, ele suporta o que quer que seu banco de dados fornece sintaxe. Aqui está um exemplo para SQL Server:

```
de usuário do usuário
onde (user.Firstname + '' user.Lastname +) like '% S% K'
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Voltaremos a alguns recursos mais exóticos do HQL onde cláusula mais adiante neste capítulo. Nós só usamos expressões simples para as restrições desta seção, vamos combinar vários com operadores lógicos.

7.2.7 Os operadores lógicos

Operadores lógicos (e parênteses para agrupamento) são usados para combinar expressões:

```
de usuário do usuário
    onde user.Firstname como "% S" e user.Lastname como "% K"

de usuário do usuário
    onde (user.Firstname como "% S" e user.Lastname como "% K")
    ou user.email in ('foo@hibernate.org', 'bar@hibernate.org')
```

Se você adicionar vários ICriterion instâncias para a ICriteria exemplo, eles são aplicados conjunctively (isto é, usando e):

```
session.CreateCriteria (typeof (User))
    . Add (Expression.Like ("Nome", "% S"))
    . Add (Expression.Like ("Sobrenome", "% K"))
```

Se você precisa de disjunção (ou), Você tem duas opções. A primeira é usar Expression.Or () juntamente com Expression.And ():

```
ICriteria crit = session.CreateCriteria (typeof (User))
    . Add (
        Expression.Or (
            Expression.And (
                Expression.Like ("Nome", "% S"),
                Expression.Like ("Sobrenome", "% K")
            ),
            Expression.In ("Email", e-mails)
        )
    );
```

A segunda opção é usar Expression.Disjunction () juntamente com Expression.Conjunction ():

```
ICriteria crit = session.CreateCriteria (typeof (User))
    . Add (Expression.Disjunction ()
        . Add (Expression.Conjunction ()
            . Add (Expression.Like ("Nome", "% S"))
            . Add (Expression.Like ("Sobrenome", "% K"))
        )
        . Add (Expression.In ("Email", e-mails))
    );
```

Nós pensamos que as duas opções são feios, mesmo depois de passar cinco minutos tentando formatá-los para o máximo legibilidade. Então, a menos que você está construindo uma consulta on the fly, a seqüência de HQL é muito mais fácil entender. Consultas complexas critérios são úteis apenas quando eles são criados através de programação, pois exemplo, no caso de uma tela de busca complexo com vários critérios de pesquisa opcional, que pode ter um CriteriaBuilder que traduz restrições usuário ICriteria instâncias.

7.2.8 Ordenação resultados da consulta

Todas as linguagens de consulta fornecem um mecanismo para ordenar os resultados da consulta. HQL fornece uma por fim cláusula, semelhante à SQL.

Esta consulta retorna todos os usuários, ordenados por nome de usuário:

```
desde o pedido do usuário u por u.Username
```

Você especificar ordem ascendente e descendente utilizando asc ou desc:

```
desde o pedido do usuário u por u.Username desc
```

Finalmente, você pode ordenar por múltiplas propriedades:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
desde o pedido do usuário u por u.Lastname asc, u.Firstname asc
```

O ICriteria API fornece uma facilidade similar:

```
Resultados IList session.CreateCriteria = (typeof (User))
    . AddOrder (Order.Asc ("Apelido"))
    . AddOrder (Order.Asc ("Nome"))
    . List ();
```

Até agora, nós discutimos apenas os conceitos básicos de consultas HQL e critérios. Você já aprendeu a escrever um simples a partir de cláusula e usar apelidos para as aulas. Nós combinamos expressões restrição vários com operadores lógicos. No entanto, nosso foco é único classes-que é persistente, nós só referenciada uma única classe no a partir de cláusula. Uma técnica de consulta importante que nós não discutimos ainda é o união de associações em tempo de execução.

7.3 associações Unir

Ao consultar bases de dados, às vezes queremos combinar dados de duas ou mais relações. Isto é alcançado através de um participar. Por exemplo, poderíamos juntar os dados no ITEM e BID tabelas, como mostrado na

figura 7.1. Note que nem todas as colunas e linhas possíveis são mostradas, daí as linhas pontilhadas.

ITEM_ID	NAME	INITIAL_PRICE	BID	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00		1	1	10.00
2	Bar	50.00		2	1	20.00
3	Baz	1.00		3	2	55.50

Figura 7.1 A ITEM e BID tabelas são candidatos óbvios para uma operação de junção.

O que a maioria das pessoas pensam quando ouvem a palavra juntar no contexto de bancos de dados SQL é um interior participar. Uma junção interna é um dos vários tipos de associações, e é o mais fácil de entender. Considere o SQL declaração e resultado na figura 7.2. Esta declaração SQL é um ANSI estilo join.

```
from ITEM I inner join BID B on I.ITEM_ID = B.ITEM_ID
```

ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
1	Foo	2.00	2	1	20.00
2	Bar	50.00	3	2	55.50

Figura 7.2 A tabela resultado de uma junção ANSI-style interior de duas tabelas

Se unirmos tabelas ITEM e BID com uma junção interna, usando seus atributos comuns (o Item_id coluna), temos todos os itens e suas propostas em uma tabela novo resultado. Observe que o resultado desta operação contém apenas os itens que têm propostas. Se queremos que todos os itens e valores nulos em vez de dados licitação quando há

é sem licitação correspondente, usamos um (Esquerda) de junção externa, como mostrado na figura 7.3.

ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
1	Foo	2.00	2	1	20.00
2	Bar	50.00	3	2	55.50
3	Baz	1.00	null	null	null

Figura 7.3 O resultado de um estilo exterior ANSI-left join de duas tabelas

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Você pode pensar em uma tabela de junção como trabalhar como segue. Primeiro, você recebe um produto cartesiano dos dois tabelas, tomando todas as combinações possíveis de ITEM linhas com BID linhas. Segundo, você filtrar estes juntou-se linhas usando um condição de junção. Note-se que o banco de dados muito mais algoritmos sofisticados para avaliar uma junção, que normalmente não construir um produto que consome memória e em seguida, filtrar todas as linhas. A junção condição é apenas uma expressão booleana que é avaliada como true se a linha juntada é para ser incluído no resultado. No caso da junção externa esquerda, cada linha na (à esquerda) ITEM tabela que nunca satisfaz a junção condição também é incluído no resultado, com valores nulos retornados para todas as colunas de BID. (A direito exterior juntar-se recuperaria todos os lances e nulo se a oferta não tem não-item certamente uma consulta sensível em nosso situação.)

Em SQL, a condição de junção é normalmente especificada explicitamente, não é possível simplesmente usar o nome de

7.3.1 NHibernate não precisa de estrangeira para especificar como duas tabelas devem ser unidas. Em vez disso, temos que especificar a junção. Em consultas NHibernate, você não costuma especificar uma condição de junção explícita. Em vez disso, você especificar o nome da cláusula de uma junção ANSI-style ou no onde cláusula para a chamada teta de estilo join, onde Item.id = B.item.id.

nome de uma associação de classe mapeada para que NHibernate pode trabalhar fora da junção para você. Por exemplo,

O Item classe tem uma associação chamada licitações com o Oferta classe. Se o nome desta associação em nosso consulta, NHibernate tem informação suficiente no documento de mapeamento para depois deduzir a juntar-se expressão. Isso ajuda a tornar as consultas menos detalhada e mais legível.

HQL oferece quatro maneiras de expressar junta interna e externa:

- Um ordinário juntar-se à a partir de cláusula
- Abuscar juntar-se à a partir de cláusula
- Atheta-style juntar-se à onde cláusula
- Um implícito associação juntar

Vamos discutir todas essas opções neste capítulo, mas porque o "normal" e "fetch" a partir de cláusula juntar-se ter a clara sintaxe, vamos discutir estes primeiros.

Ao trabalhar com NHibernate, geralmente há várias razões pelas quais você pode querer usar um juntar-se, e é importante notar que NHibernate realmente permite diferenciar entre os fins para aderir. Vamos colocar isso no contexto de um pequeno exemplo. Suponha que está consultando Items, há três possíveis razões que possam estar interessados em aderir ao Ofertas.

Em primeiro lugar, podemos querer recuperar Items voltou com base em algum critério que deve ser aplicado a seus Ofertas. Por exemplo, você pode desejar que todas Itens que têm uma oferta de mais de US \$ 100, daí isto requer um junção interna.

Alternativamente, podemos estar executando uma consulta em que estamos interessados principalmente em somente os Items sem qualquer critério especial para Ofertas. Nós pode ou não querer acessar o Lances para um item, mas queremos a opção para NHibernate para preguiçosamente carregá-los quando nós primeiro acessar a coleção.

Antera razão para juntar a Lances é que a gente pode querer executar uma junção externa para carregar todos os Items

juntamente com os seus licitações na escolha mesma, algo que chamamos de busca ansiosa anteriormente.

Lembre-se que

nós preferimos para mapear todas as associações de preguiçoso por padrão, portanto, um ansioso, outer-join fetch consulta pode ser usado para

7.3.2 Busca ansiosa

Suponha essa estratégia de busca padrão em tempo de execução. Vamos discutir este primeiro cenário. Em HQL, você pode especificar que uma associação deve ser ansiosamente buscada por uma associação externa usando o

buscar palavra-chave na a partir de cláusula:

```
de item Item
left join fetch item.Bids
    onde item.Description like '% parte%'
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Esta consulta retorna todos os itens com uma descrição que contém a string parte, E todos os seus lances, em um única de seleção. Quando executado, ele retorna uma lista de Item casos, com os seus licitações coleções totalmente inicializado. Nós chamamos isso de a partir de cláusula fetch join. O objetivo da associação é buscar um desempenho otimização: Usamos esta sintaxe só porque queremos inicialização ansiosos do licitações coleções em um SQL única selecionar.

Podemos fazer a mesma coisa usando o ICriteria API:

```
session.CreateCriteria (typeof (ponto))
    . SetFetchMode ("Licitações", FetchMode.Eager)
    . Add (Expression.Like ("Descrição", "parte", MatchMode.Anywhere))
    . List ();
```

Ambos estes resultado consultas no SQL seguinte:

```
selecionar DESCRIÇÃO I., I. CRIADO, I. SUCCESSFUL_BID, B. BID_ID,
B. QUANTIDADE, B. item_id, B. CRIADO
ITEM I da
LEFT OUTER JOIN B BID em I. item_id = B. item_id
onde I. DESCRIÇÃO like '% parte%'
```

Nós também podemos prefetch many-to-one ou one-to-one associações usando a mesma sintaxe:

```
Oferta de lance
left join fetch bid.Item
left join fetch bid.Bidder
onde bid.Amount > 100
```

```
session.CreateCriteria (typeof (Bid))
    . SetFetchMode ("Item", FetchMode.Eager)
    . SetFetchMode ("Concorrente", FetchMode.Eager)
    . Add (Expression.Gt ("Valor", 100))
    . List ();
```

Essas consultas execute o seguinte SQL:

```
selecionar DESCRIÇÃO I., I. CRIADO, I. SUCCESSFUL_BID,
B. BID_ID, O VALOR B., B. item_id, B. CRIADO,
U. USERNAME, PASSWORD U., U. FIRSTNAME, U. LASTNAME
de B BID
ITEM junção externa à esquerda na I I. item_id = B. item_id
deixou U USUÁRIO junção externa na U. USER_ID = B. BIDDER_ID
onde VALOR B.> 100
```

Note que o à esquerda palavra-chave é opcional no HQL, para que pudéssemos reescrever os exemplos anteriores usando juntar buscar. Embora isto pareça simples de usar, há um par de coisas a considerar e lembre-se:

- HQL sempre ignora o documento de mapeamento configuração fetch (outer join) ansioso. Se você mapeou algumas associações de ser obtida pela junção externa, através da criação outer-join = "true" ou fetch = "join" sobre o mapeamento da associação, qualquer consulta HQL irá ignorar essa preferência. Com HQL, se você quiser busca ansiosa que você precisa perguntar para ele na cadeia de consulta. HQL é projetado para ser tão **possível quanto** pode completamente (re) definir a estratégia de busca que deve ser usado durante a execução. Em comparação, os critérios vão tomar conhecimento pleno de seus mapeamentos! Se você especificar outer-join = "true" no arquivo de mapeamento, a consulta vai buscar critérios que a associação de junção externa assim como ISession.Get () ou ISession.Load () para recuperação por identificador. Critérios para uma consulta, você pode desabilitar explicitamente outer join fetching chamando SetFetchMode ("Licitações", FetchMode.Lazy).

NHibernate atualmente limita a busca de uma única coleção ansiosamente. Esta é uma razoável restrição, uma vez buscar mais de uma coleção em uma única consulta seria um cartesiano resultado do produto. Essa restrição pode ser relaxada em uma versão futura do NHibernate, mas nós

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

incentivá-lo a pensar sobre o tamanho do conjunto de resultados, se mais de uma coleção é buscada no uma associação externa. A quantidade de dados que teria que ser transportados entre banco de dados e aplicação pode facilmente crescer na faixa de megabyte, e mais do que seria jogado fora imediatamente (NHibernate achata tabular o resultado conjunto para construir o gráfico do objeto). Você pode buscar o maior número um-para-um ou muitos-para-um como você gosta.

- n Se você pegar uma coleção, NHibernate não retorna uma lista de resultados distintos. Por exemplo, um individual Item pode aparecer várias vezes no resultado IList. Se você outer-join fetch os lances. Você provavelmente vai precisar para fazer a resultados distintos si mesmo usando, por exemplo: distinctResults = new HashSet(resultList);. Um ISET não permite elementos duplicados.

Isto é como NHibernate implementa o que chamamos runtime buscar estratégias de associação , Um poderoso característica que é essencial para alcançar alta performance em ORM. Vamos continuar com o outro juntar-se operações.

7.3.3 Usando aliases com junta

Já discutimos o papel do onde cláusula expressa restrição. Muitas vezes, você vai precisar aplicar os critérios de restrição para múltiplas classes associados (tabelas associadas). Se quisermos fazer isso usando um

HQL a partir de cláusula de junção, é preciso atribuir um alias para a classe de cadastro:

```
de item Item
juntar lance item.Bids
    onde item.Description like '% parte%'
    e bid.Amount > 100
```

Esta consulta atribui o alias item para a classe Item eo alias oferta juntou-se à Item'S lances. Em seguida, usar os dois aliases para expressar nossos critérios de restrição na onde cláusula.

O SQL resultante é a seguinte:

```
selecionar DESCRIÇÃO I., I. CRIADO, I. SUCCESSFUL_BID,
B. BID_ID, O VALOR B., B. item_id, B. CRIADO
ITEM I`da
B BID junção interna em I. item_id = B. item_id
onde I. DESCRIÇÃO like '% parte%'
e B. VALOR > 100
```

A consulta retorna todas as combinações de associados Lances e Items. Mas, ao contrário de um fetch join, o Lances cobrança da Item não é inicializado pela consulta! Então o que queremos dizer por um combinação aqui? Nós significa um par ordenado: (Item lance). No resultado da consulta, NHibernate representa um par ordenado como um array. Vamos discutir um exemplo de código completo com o resultado de tal consulta:

```
IQuery session.CreateQuery q = "from Item item juntar lance item.Bids";
foreach (object [] par em q.List ()) {
    Item item = par [0];
    Licitação Licitação = (Bid) par [1];
}
```

Em vez de um IList de Itens, Esta consulta retorna um IList de object [] matrizes. No índice 0é o Item, e no índice 1é o Oferta. A especial Item pode aparecer várias vezes, uma para cada associado Oferta.

Isso tudo é diferente do caso de uma consulta com uma busca ansiosa de junção. A consulta com o fetch juntar retornou um IList de Items, com inicializado Lances coleções.

Se não deseja que o Ofertas no resultado da consulta, pode-se especificar um selecionar cláusula no HQL. Esta cláusula é opcional (não é em SQL), então só temos que usá-lo quando não estamos satisfeitos com o resultado retornado por padrão. Usamos o alias em um selecionar cláusula para recuperar apenas os objetos selecionados:

```
selecionar o item
de item Item
juntar lance item.Bids
    onde item.Description like '% parte%'
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
e bid.Amount > 100
```

Agora o SQL gerado parecido com este:

```
selecionar DESCRIÇÃO I., I. CRIADO, I. SUCCESSFUL_BID,  
ITEM I da  
B BID juncão interna em I. item_id = B. item_id  
onde I. DESCRIÇÃO like '% parte%'  
e B. VALOR > 100
```

O resultado da consulta contém apenas Items, e porque é um juncão interna, apenas Items que têm Ofertas:

```
IQuery session.CreateQuery q = ("select i do item i juntar i.Bids b");  
foreach (item Item em <Item> q.List () {  
    ...  
}
```

Como você pode ver, usando aliases no HQL é a mesma para ambas as classes diretos e associações unidas. Nós

atribuir aliases na a partir de cláusula e usá-los no onde e no opcional selecionar cláusula. O selecionar cláusula no HQL é muito mais poderoso, nós discutir isso em detalhes mais adiante neste capítulo.

Há duas maneiras de expressar uma juncão na ICriteria API, daí existem duas maneiras de usar aliases para a restrição. O primeiro é o CreateCriteria () método da Critérios interface. Isso significa que você pode aninhar chamadas para CreateCriteria ():

```
ICriteria itemCriteria session.CreateCriteria = (typeof (Item));  
itemCriteria.Add (Expression.Like ("Descrição",  
    "Parte",  
    MatchMode.Anywhere));  
ICriteria bidCriteria = itemCriteria.CreateCriteria ("Propostas");  
bidCriteria.Add (Expression.Gt ("Valor", 100));
```

```
Resultados IList itemCriteria.List = ();
```

Nós geralmente escrever a consulta da seguinte forma, usando encadeamento de método:

```
Resultados IList =  
    session.CreateCriteria (typeof (ponto))  
    . Add (Expression.Like ("Descrição", "parte", MatchMode.Anywhere))  
    . CreateCriteria ("Propostas")  
    . Add (Expression.Gt ("Valor", 100))  
    . List ();
```

A criação de um ICriteria instância para o Lances do Item resulta em uma juncão interna entre os tabelas das duas classes. Note que podemos chamar de List () em ambos ICriteria instância sem alterando os resultados da consulta.

A segunda maneira de expressar esta consulta usando o ICriteria API é atribuir um alias para o juntou entidade:

```
Resultados IList =  
    session.CreateCriteria (typeof (ponto))  
    . CreateAlias ("Propostas", "bid")  
    . Add (Expression.Like ("Descrição", "parte%"))  
    . Add (Expression.Gt ("bid.Amount", 100))  
    . List ();
```

Esta abordagem não usa uma segunda instância ICriteria. Assim, as propriedades da entidade juntou deve ser qualificada pelo apelido atribuído em CreateAlias (). Propriedades do entidade raiz (Item) Podem ser referidos para o alias sem qualificação ou usando o alias "This". Assim, a seguir é equivalente:

```
Resultados IList =  
    session.CreateCriteria (typeof (ponto))  
    . CreateAlias ("Propostas", "bid")
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    . Add (Expression.Like ("this.Description", "parte%"))
    . Add (Expression.Gt ("bid.Amount", 100)
    . List ();

```

Por padrão, uma consulta critérios retorna apenas a raiz da entidade, neste caso, o `Items`-no resultado da consulta.

Vamos resumir com um exemplo completo:

```

IList resultados <Item> =
    session.CreateCriteria (typeof (ponto))
        . CreateAlias ("Propostas", "bid")
        . Add (Expression.Like ("this.Description", "parte%"))
        . Add (Expression.Gt ("bid.Amount", 100))
        . <Item> List ();

foreach (item item no resultado) {
    / / Faz alguma coisa
}

```

Tenha em mente que o `Lances` coleção de cada `Item` não é inicializado. A limitação de consultas critérios é que você não pode combinar uma `CreateAlias` com um modo de buscar ansiosos, por exemplo, `SetFetchMode ("Licitações", FetchMode.Eager)` não é válido.

Às vezes você gostaria de uma maneira menos detalhada para expressar um join. Em NHibernate, você pode usar um implícito associação de junção.

7.3.4 Usando implícita junta

Até agora, usamos simples nomes de propriedade qualificado como `bid.Amount` e `item.Description` em nosso Consultas HQL. HQL suporta expressões de propriedade multipart caminho para dois propósitos:

- Consultando componentes
- Expressando associação implícita junta

O primeiro uso é simples:

```
do Usuário u onde u.Address.City = 'Bangkok'
```

Nós expressamos as partes do componente mapeado Endereço com a notação de ponto. Este uso também é apoiado pela `ICriteria` API:

```
session.CreateCriteria (typeof (User))
    . Add (Expression.Eq ("Address.City", "Bangkok"));
```

O segundo uso, a associação implícita adesão, está disponível apenas no HQL. Por exemplo:

```
Oferta de lance em que bid.Item.Description like '% parte%'
```

Isto resulta em uma junção implícita sobre as associações many-to-one de `Oferta` para `Item`. Implícita a junção é sempre dirigida ao longo de muitos-para-um ou um-para-um, nunca através de um conjunto de valores associação (você não pode escrever `item.Bids.Amount`).

Várias associações são possíveis em uma única expressão propriedade de caminho. Se a associação de `Item` para

`Categoria` seria many-to-one (em vez dos actuais muitos-para-muitos), podemos escrever

```
Oferta de lance em que bid.Item.Category.Name like '% Laptop'
```

Nós desaprovar o uso deste açúcar sintático para consultas mais complexas. Junta são importantes, e especialmente quando otimização de consultas, você precisa ser capaz de ver de relance quantos deles há são. Considere a seguinte consulta (novamente, usando um many-to-one de `Item` para `Categoria`):

```
Oferta de lance
onde bid.Item.Category.Name like '% Laptop'
e bid.Item.SuccessfulBid.Amount > 100
```

Quantas associações são necessárias para expressar isso em SQL? Mesmo se você receber a resposta certa, nós apostamos que leva mais do que alguns segundos. A resposta é três; o SQL gerado é algo como isto:

```
selecione ...
de B BID
inner join ITEM I em B. item_id = I. item_id
CATEGORIA interior juntar C em I. category_id = C. category_id
interior BID participar na SB I. SUCCESSFUL_BID_ID = SB.BID_ID
onde C. NOME like '% Laptop'
e SB.AMOUNT > 100
```

É mais evidente se expressar a mesma consulta como esta:

```
Oferta de lance
juntar-se item de bid.Item
onde item.Category.Name like '% Laptop'
e item.SuccessfulBid.Amount > 100
```

Podemos até ser mais detalhado:

```
a partir da proposta como lance
junta bid.Item como item
junta item.Category como gato
junta item.SuccessfulBid como winningBid
onde cat.Name like '% Laptop'
e winningBid.Amount > 100
```

Vamos continuar com juntar condições usando atributos arbitrários, expresso em theta estilo.

7.3.5 Theta em estilo junta

Um produto cartesiano permite recuperar todas as combinações possíveis de ocorrências de duas ou mais classes. Esta consulta retorna todos os pares ordenados de Usuários e Categoria objetos:

```
da categoria do usuário,
```

Obviamente, isso geralmente não é útil. Há um caso em que é comumente usados: theta-style junta.

Em SQL tradicional, uma junção theta-estilo é um produto cartesiano, juntamente com uma condição de junção na

onde cláusula, que é aplicado sobre o produto para restringir o resultado.

Em HQL, a sintaxe theta estilo é útil quando o seu não é uma condição de junção relação de chave estrangeira mapeado para uma associação de classe. Por exemplo, suponha que armazenamos as UsuárioName's em registros de log, em vez

mapeamento de uma associação de LogRecord para Usuário. As aulas não "sabe" nada sobre cada outros, porque eles não estão associados. Podemos, então, encontrar todos os Usuários e seus LogRecords com o seguintes theta de estilos join:

```
de usuário do usuário, LogRecord log onde user.username log.Username =
```

A condição de junção aqui é o nome de usuário, Apresentado como um atributo em ambas as classes. Se ambas as entidades têm a mesma coisa nome de usuário, Eles juntou-se (com uma junção interna) no resultado. O resultado da consulta consiste em

```
pares ordenados:List<Object> resultados = session.CreateQuery =
    @ "Do usuário do usuário, LogRecord log
        onde user.username log.Username = "
    )
    . List ();

foreach (Object [par] em resultados)
    Usuário user = par (Usuário) [0];
    LogRecord log = (LogRecord) par [1];
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Nós podemos mudar o resultado, adicionando um selecionar cláusula.

Você provavelmente não vai precisar usar theta-style junta com freqüência. Note que o ICriteria API não fornecer todos os meios para expressar produtos cartesianos ou teta-style junta. É também atualmente não possível em NHibernate para outer-join duas tabelas que não têm uma associação mapeada.

7.3.6 Comparando identificadores

É extremamente comum para executar consultas que comparam chave primária ou estrangeira valores-chave para qualquer parâmetros de consulta ou outros primária ou valores de chaves estrangeiras. Se você pensar sobre isso com mais objetos termos orientados, o que você está fazendo é comparar referências de objeto. HQL suporta o seguinte:

```
    onde i.Seller = u e u.Username = 'steve'
```

Nesta consulta, i.Seller refere-se a chave estrangeira para a USUÁRIO tabela no ITEM tabela (no SELLER_ID coluna), e usuário refere-se à chave primária da USUÁRIO tabela (no USER_ID coluna). Esta próxima consulta usa um estilo de theta-join e é equivalente ao estilo ANSI muito preferido:

```
do item i juntar i.Seller u
    onde u.Username = 'steve'
```

Por outro lado, o seguinte estilo theta-join não pode ser re-expressa como uma a partir de cláusula de junção:

```
do item i, b Bid
    onde i.Seller = b.Bidder
```

Neste caso, i.Seller e b.Bidder são chaves estrangeiras da USUÁRIO mesa. Note que este é um consulta importante na nossa aplicação, nós usá-lo para identificar as pessoas de licitação para seus próprios itens.

Podemos também gosto de comparar um valor de chave estrangeira para um parâmetro para consulta de exemplo, para localizar todos os

Comentários de um Usuário:

```
USUÁRIOS givenUser
IQuery q =
    session.CreateQuery ("de onde c Comentário c.FromUser =: user");
q.SetEntity ("user", givenUser);
Resultados IList q.List = ();
```

Alternativamente, algumas vezes nós preferimos expressar esses tipos de consultas em termos de valores identificador ao invés de referências de objeto. Você pode se referir a um valor de identificador por um ou outro nome do identificador propriedade (se houver) ou o nome da propriedade especial id. Todas as classes persistentes da entidade tem este especial HQL propriedade, mesmo se você não implementar uma propriedade identificador da classe (ver capítulo 3, seção 3.5.2, "Identidade banco de dados com NHibernate").

Essas consultas são exatamente equivalente a consultas anteriores:

```
    onde i.Seller.id = u.id e u.Username = 'steve'
    a partir do item Licitação
        onde i.Seller.id = b.Bidder.id
```

No entanto, agora podemos usar o valor do identificador como um parâmetro de consulta:

```
userId long = ...
IQuery q =
    session.CreateQuery ("de onde c Comentário c.FromUser.id =: id"); q.SetInt64 ("id",
userId);
Resultados IList q.List = ();
```

Você deve ter notado que há um mundo de diferença entre as consultas a seguir:

```
de Licitação, onde b = 1 b.Item.id
de Licitação, onde b b.Item.Description like '% parte%'
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

A segunda consulta utiliza uma tabela união implícita, o primeiro não tem junta-se em tudo.

Nós agora cobria a maior parte das características das instalações de consulta do NHibernate que normalmente são necessárias

para recuperar objetos para manipulação de lógica de negócios. Na próxima seção, vamos mudar o nosso foco e discutir recursos do HQL que são usados principalmente para análise e emissão de relatórios.

7.4 relatório consultas Writing

Consultas de relatórios tirar vantagem da capacidade do banco de dados para executar agrupamento eficiente e agregação de dados.

Eles são mais de natureza relacional, pois eles nem sempre retornam entidades. Por exemplo, em vez de recuperação completa Item entidades, a consulta do relatório só poderá recuperar os seus nomes e preços. Se este for

a única informação que precisamos para uma tela de relatório, não precisamos de entidades transacional e pode salvar o

pequena sobrecarga das automáticas suja controlo e armazenamento em cache no ISession.

Vamos considerar a estrutura de uma consulta HQL novamente.:

```
[Select ...] a partir de ... [Onde ...]  
[Grupo por ... [Ter fim ...]] [by ...]
```

A cláusula só é obrigatória de uma consulta HQL é a a partir de cláusula, por isso todas as outras cláusulas são opcionais. Assim

agora, temos discutido o a partir de, ondeE por fim cláusulas. Foi utilizado também o selecionar cláusula Declaro que as entidades devem ser retornados em uma consulta de junção.

Em consultas de relatórios, você usa o selecionar cláusula de projeção eo grupo por e ter cláusulas para agregação. Vejamos o que entendemos por projeção.

7.4.1 Projeção

O selecionar cláusula realiza projeção. Ele permite que você especificar quais objetos ou propriedades de objetos que você

quer nos resultados da consulta. Por exemplo, como você já viu, a consulta retorna seguinte ordem pares de ItemS e OfertaS:

```
do item item juntar item.Bids lance onde bid.Amount > 100
```

Se apenas queria que o Items, devemos usar esta consulta:

```
seleccione o item a partir do item item juntar item.Bids lance onde bid.Amount > 100
```

Ou, se estivéssemos apenas exibindo uma página da lista para o usuário, que poderia ser adiquate apenas recuperar alguns

propriedades desses objetos necessários para essa página:

```
selecionar item.id, item.Description, bid.Amount  
do item item juntar lance item.Bids  
onde bid.Amount > 100
```

Esta consulta retorna uma matriz de objetos para cada linha. Porque há três itens na cláusula select, cada object []terá 3 elementos. Também, porque é uma consulta do relatório, os objetos não são o resultado Entidades NHibernate e não são, portanto, não são transacionais. Vamos executar a consulta com algum código:

```
Resultados IList session.CreateQuery =  
@ "Select item.id, item.Description, bid.Amount  
do item item juntar lance item.Bids  
onde bid.Amount > 100 "  
> . List ();  
  
foreach (Object [linha] no resultado) {  
  
    id = linha longa (long) [0];  
    descrição string = linha (string) [1];  
    dobro = (double) row [2];  
  
    / / ... mostrar os valores em uma tela de  
    relatório  
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Se você está acostumado a trabalhar com objetos de domínio, este exemplo vai parecer muito feio e detalhado. NHibernate nos dá uma outra abordagem a este, chamado instanciação dinâmica.

Usando instanciação dinâmica

Se você não encontrar a trabalhar com arrays de valores um pouco complicado, nós vamos usar NHibernate dinâmica instanciação e definir uma classe para representar cada linha de resultados. Podemos fazer isso usando a HQL selecionar

novo Construir:

```
selecionar novos ItemRow (item.id, item.Description, bid.Amount)
    do item item juntar lance item.Bids
    onde bid.Amount > 100
```

O ItemRow classe é aquela que ia escrever só para a nossa tela de relatório, e notar que também temos de dar-lhe um construtor apropriado. Essa consulta retorna uma nova instanciada (mas transitórios) casos de ItemRow, Como você pode ver no exemplo a seguir:

```
Resultados IList session.CreateQuery =
    @ "Select nova ItemRow (item.id, item.Description, bid.Amount)
        do item item juntar lance item.Bids
        onde bid.Amount > 100 "
)
.List ();

foreach (ItemRow linha no resultado) {
    / / Faz alguma coisa
}
```

O costume ItemRow classe não tem que ser uma classe persistente que tem seu próprio arquivo de mapeamento, mas em

Para NHibernate para "ver" isso, precisamos importá-lo usando:

```
<hibernate-mapping>
    <import class="ItemRow" />
</hibernate-mapping>
```

ItemRow é, portanto, apenas uma classe de transferência de dados, útil na geração de relatório.

Obtenção de resultados distintos

Quando você usa um selecionar cláusula, os elementos do resultado não são mais a garantia de ser único. Para exemplo, ItemS descrições não são únicos, então a seguinte consulta pode retornar a mesma descrição mais de uma vez:

```
selecionar item.Description de item Item
```

É difícil ver como isso poderia ser significativo para ter duas linhas idênticas no resultado de uma consulta, assim se você acha que as duplicatas são provavelmente, você deve usar o distinto palavra-chave:

```
selecionar item.Description distinta item Item
```

Isso elimina duplicados da lista retornada de Item descrições.

Chamar funções SQL

Você deve se lembrar que você pode chamar funções de banco de dados SQL específicos na onde cláusula. É também possível, pelo menos para alguns dialetos SQL NHibernate, para chamar funções de banco de dados específicos do SQL selecionar cláusula. Por exemplo, a consulta a seguir recupera a data ea hora atuais do banco de dados servidor (a sintaxe SQL Server), juntamente com uma propriedade de Item:
selecionar item.StartDate, getdate () do item Item

A técnica de funções de banco de dados no selecionar cláusula é, naturalmente, não se limitando a banco de dados funções dependentes, mas para outro, mais genérico (ou padronizado) funções SQL, assim:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
selecionar item.StartDate, item.EndDate, superior (item.Name)
de item Item
```

Essa consulta retorna uma `object []` com a data de início e término de um leilão item, eo nome do o item tudo em maiúsculas.

Vamos agora olhar para chamar SQL funções de agregação.

7.4.2 Usando agregação

NHibernate reconhece as seguintes funções de agregação: `count()`,`min()`,`max()`,`sum()`E `avg()`.
Esta consulta conta todos os `Item`s:

```
select count (*) do item
```

O resultado é retornado como um Número

inteiro:

```
int count =
(Int) session.CreateQuery ("select count (*) from Item")
. UniqueResult ();
```

Observe como usamos `*`, Que tem a mesma semântica como em SQL.

A variação próxima da consulta conta todos `Item`s que têm uma `successfulBid`:

```
select count (item.SuccessfulBid) de item Item
```

Esta consulta calcula o total de todos os bem-sucedida Ofertas:

```
select sum (item.SuccessfulBid.Amount) de item Item
```

A consulta retorna um valor do mesmo tipo como os elementos somados, neste caso duplo. Observe o uso de uma junção implícita na `selecionar` cláusula: Nós navegar pela associação (`SuccessfulBid`) a partir de `Item` para `Oferta` referenciando-a com um ponto.

A próxima consulta retorna os valores de lance mínimo e máximo para um determinado `Item`:

```
selecionar min (bid.Amount), max (bid.Amount)
Oferta de lance em que bid.Item.id = 1
```

O resultado é um par ordenado de duplos (duas instâncias de duplo em um `object [] array`).

O especial `count (distintos)` função ignora duplicatas:

```
select count (item.Description distintas) de item Item
```

Quando você chama uma função de agregação na `selecionar` cláusula, sem especificar qualquer agrupamento em um grupo

por cláusula, você recolhe o resultado para baixo para uma única linha que contém o valor agregado (s). Este significa (na ausência de um grupo por cláusula) qualquer `selecionar` cláusula que contém uma função agregada deve conter apenas funções agregadas.

Assim, para as estatísticas mais avançadas e elaboração de relatórios, você precisa ser capaz de realizar agrupamento.

7.4.3 Agrupamento

Assim como em SQL, qualquer propriedade ou apelido que aparece na HQL fora de uma função agregada na `selecionar` cláusula também deve aparecer na `grupo por` cláusula.

Considere a próxima consulta, que conta o número de usuários com o nome de cada especial passada:

```
selecionar u.Lastname, count (u) de usuário u
grupo por u.Lastname
```

Agora olhe para o SQL gerado:

```
selecionar U. LAST_NAME, count (U. USER_ID)
a partir de U. USUÁRIO
grupo por U. LAST_NAME
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Neste exemplo, o `u.Lastname` não é dentro de uma função agregada, nós usá-lo para o grupo o resultado. Nós também não precisa especificar a propriedade que nós gostaríamos de contar em HQL. O SQL gerado utilizar automaticamente a chave primária, se usarmos um alias que foi definido na a partir de cláusula.

A próxima consulta encontra o valor do lance médio para cada item:

```
selecionar bid.Item.id, avg (bid.Amount) de lance Licitação
grupo por bid.Item.id
```

Esta consulta retorna pares ordenados de Item identificadores e quantidade da oferta média. Observe como usamos a `id` propriedade especial para se referir ao identificador de uma classe persistente não importa o que o identificador é real nome da propriedade é.

A próxima consulta conta o número de licitações e calcula a oferta média por item não vendidos:

```
selecionar bid.Item.id, count (bid), avg (bid.Amount)
Oferta de lance
onde bid.Item.SuccessfulBid é nulo
grupo por bid.Item.id
```

Esta consulta usa uma associação implícita participar. Para uma junção explícita comuns no a partir de cláusula (`não um fetch join`), podemos re-expressá-la da seguinte forma:

```
selecionar bidItem.id, count (bid), avg (bid.Amount)
Oferta de lance
    juntar bid.Item bidItem
onde bidItem.SuccessfulBid é nulo
grupo por bidItem.id
```

Para inicializar o licitações cobrança da Items, podemos usar uma busca juntar-se e referem-se às associações a partir do outro lado:

```
selecionar item.id, count (bid), avg (bid.Amount)
de item Item
    buscar juntar lance item.Bids
onde item.SuccessfulBid é nulo
grupo por item.id
```

Às vezes, você vai querer restringir ainda mais o resultado selecionando apenas os valores particulares de um grupo.

7.4.4 Restringindo grupos com ter

O `onde` cláusula é usada para executar a operação relacional de restrição de linhas. O `ter` cláusula executa restrição grupos.

Por exemplo, a próxima consulta conta com usuários cada sobrenome que começa com K:

```
selecionar user.Lastname, count (usuário)
de usuário do usuário
grupo por user.Lastname
ter user.Lastname como 'K%'
```

As mesmas regras governam o `selecionar` e `ter` cláusulas: Somente propriedades agrupadas podem aparecer fora uma função agregada. A próxima consulta conta o número de lances por item não vendidos, apresentar os resultados de apenas os itens que têm mais de 10 lances:

```
selecionar item.id, count (bid), avg (bid.Amount)
de item Item
    juntar lance item.Bids
onde item.SuccessfulBid é nulo
grupo por item.id
ter count (bid)> 10
```

Consultas mais usar um relatório `selecionar` cláusula de escolher uma lista de propriedades projetadas ou agregados. Você

visto que, quando mais de uma propriedade ou alias é listada na `selecionar` cláusula, retorna o NHibernate resultados da consulta como tuplas: Cada linha da lista de resultado da consulta é uma instância de `object` [].

Por favor, [Todas](#) comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

inconveniente e não typesafe, assim NHibernate fornece a selecionar novos construtor, como mencionado anteriormente. Você pode criar novos objetos dinamicamente com esta técnica e também usá-lo em combinação com agregação e agrupamento.

Se nós definimos uma classe chamada ItemBidSummary com um construtor que leva um longo, Um corda, E um int, Podemos usar a seguinte consulta:

```
selecionar novos ItemBidSummary (bid.Item.id, count (bid), avg (bid.Amount))
  Oferta de lance
  onde bid.item.SuccessfulBid é nulo
  grupo por bid.Item.id
```

No resultado desta consulta, cada elemento é uma instância de ItemBidSummary, Que é um resumo de um Item, O número de propostas para esse item, eo valor do lance média. Esta abordagem é typesafe, e um transferência de dados de classe, tais como ItemBidSummary pode ser facilmente estendido para impressão formatado especial de valores nos relatórios.

7.4.5 Melhorar o desempenho com consultas de relatório

Consultas de relatório pode ter um impacto sobre o desempenho de sua aplicação. Vamos explorar esta questão em mais profundidade.

A única vez que vi qualquer sobrecarga significativa no código em relação ao NHibernate direta Consultas e ADO.NET, em seguida, apenas para teste irrealisticamente simples casos é no caso especial de leitura consultas apenas contra um banco de dados local. É possível para um banco de dados completamente cache de resultados da consulta em memória e responder rapidamente, de modo benchmarks geralmente são inúteis se o conjunto de dados é pequena: SQL Plain ADO.NET e será sempre a opção mais rápida.

Por outro lado, mesmo com um resultado pequeno, NHibernate ainda deve fazer o trabalho de adicionar os objetos resultantes de uma consulta ao ISession cache (talvez também o cache de segundo nível) e gerenciar singularidade, e assim por diante. Consultas de relatório dar-lhe uma forma de evitar a sobrecarga de gestão do ISession cache. A sobrecarga de uma consulta do relatório em relação ao NHibernate direta SQL / ADO.NET não é geralmente mensuráveis, mesmo em casos extremos irrealista como carregar um milhão de objetos de um local banco de dados sem a latência da rede.

Consultas de relatório usando projeção em HQL permitem especificar exatamente quais as propriedades que deseja recuperar. Para consultas de relatório, você não está selecionando entidades, mas apenas propriedades ou

valores agregados:
de usuário do usuário
grupo por user.Lastname

Esta consulta não retorna uma entidade persistente, de modo NHibernate não adiciona um objeto transacional para o

ISession cache. Além disso, NHibernate não vai monitorar alterações a estes objetos retornados.

Relatórios de resultados em consultas mais rápidas liberação de memória alocada, uma vez que os objetos não são mantidos na

ISession cache até que a ISession está fechado, eles podem ser de coleta de lixo, logo que eles são dereferenced pela aplicação, após a execução do relatório.

Estas considerações são quase sempre muito menor, por isso não vá para fora e reescrever todas as suas linhas apenas as operações de usar consultas de relatório em vez de objetos transacional, em cache, e monitorados. Relatório

as consultas são mais detalhado e (discutivelmente) menos orientada a objetos. Eles também fazem uso menos eficiente de

Caches NHibernate, que é muito mais importante uma vez que você considerar a sobrecarga de controle remoto comunicação com o banco de dados em sistemas de produção. Nós seguimos o "não otimizar prematuramente" Wisdom, e recomendo que você espere até que você encontre um caso real onde você tem um desempenho real problema antes de usar essa otimização.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

7.4.6 Obtenção de DataSets

Pode acontecer que você tem que interagir com um componente usando DataSets. Muitos motores de relatório, como

Crystal Reports, tem suporte limitado para POCO (mas que pode ser o suficiente). Sua fonte de dados comum ou é o banco de dados diretamente ou um DataSet. Mas, como NHibernate não retornam DataSets, temos que encontrar uma solução.

O trabalho em torno de mais comum é usar diretamente ADO.NET para obter o DataSet; esta solução pode caber

em muitas situações, mas não tirar proveito dos recursos NHibernate e requer cuidado monitoramento de possíveis mudanças como eles podem fazer caches NHibernate obsoleto.

Outra solução é usar NHibernate para consultar dados e preencher um DataSet com o resultado. Esta operação

pode ser feito manualmente por escrever um código semelhante ao exigido para DTOs, mas vai se tornar tédio nós quando se lida com numerosas entidades. Neste caso, a geração de código pode ajudar a simplificar bastante a processo.

Agora, vamos voltar para consultas de entidade regular. Há ainda características NHibernate muitos à espera de ser descoberto.

Você vai usar técnicas avançadas de consulta com menos freqüência com NHibernate, mas será útil saber sobre eles. Nesta seção, falamos de programação de construção com critérios de "objetos exemplo", um tópico que brevemente apresentado anteriormente.

Filtragem coleções também é uma técnica acessível: Você pode usar o banco de dados em vez de objetos de filtragem na memória. Subconsultas e as consultas em SQL nativo irá arredondar para fora seu conhecimento da consulta NHibernate técnicas.

Consultas 7.5.1 Dinâmica

É comum para consultas a ser construído programaticamente através da combinação de vários critérios de consulta opcional dependendo da entrada do usuário. Por exemplo, um administrador de sistema pode querer procurar usuários por qualquer combinação de nome ou sobrenome, e para recuperar o resultado ordenado por nome de usuário. Usando HQL, nós

poderia construir a consulta usando manipulações string:

```
StringBuilder queryString = new StringBuilder ();
bool conditionFound = false;

if (firstname != null) {
    queryString.Append ("inferior (u.Firstname) como: nome");
    conditionFound = true;
}
if (sobrenome != null) {
    if (conditionFound) queryString.Append ("e");
    queryString.Append ("inferior (u.Lastname) como: sobrenome");
    conditionFound = true;
}

cadeia fromClause conditionFound ==?
    "A partir de u usuário onde";
    "A partir de u usuário";

. queryString.Insert (0, fromClause) Append ("order by u.username");

Consulta IQuery getSession = () CreateQuery (queryString.ToString ());

if (firstname != null)
    query.SetString ("nome",
        '%' + Firstname.ToLower () + '%');
if (sobrenome != null)
    query.SetString ("sobrenome",
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        '%' + Lastname.ToLower () + '%');

    retorno <User> query.List ();
}

Este código é tedioso e barulhenta, então vamos tentar uma abordagem diferente. O ICriteria API parece
promissor:
    público IList findusers <User> (nome string,
                                         lastname string) {

        ICriteria crit = getSession () CreateCriteria (typeof (User)).;

        if (firstname! = null) {
            crit.Add (Expression.InsensitiveLike ("Nome",
                                                 firstname,
                                                 MatchMode.Anywhere));
        }
        if (sobrenome! = null) {
            crit.Add (Expression.InsensitiveLike ("sobrenome",
                                                 sobrenome,
                                                 MatchMode.Anywhere));
        }

        crit.AddOrder (Order.Asc ("Username"));

        retorno <User> crit.List ();
}

```

Este código é muito menor e mais legível. Note que o `InsensitiveLike()` operador executa uma case-insensitive partida. Parece não haver dúvida de que esta é uma abordagem melhor. No entanto, para pesquisas com muitos critérios de pesquisa opcional, há uma maneira ainda melhor.

Em primeiro lugar, observar que à medida que acrescentamos novos critérios de pesquisa, a lista de parâmetros de `Findusers()` cresce. Seria melhor para capturar as propriedades pesquisáveis como um objeto. Uma vez que todas as propriedades pertencem à pesquisa

Usuário classe, por que não usar uma instância de Usuário?

QBE usa essa idéia, você fornecer uma instância da classe consultado com algumas propriedades inicializado, ea consulta retorna todas as instâncias persistentes com correspondência de valores de propriedade. NHibernate implementa

```

    publica <User> IList findusers (User u) {
        QBE como parte do ICriteria consulta API:
        = Exemplo exampleUser
          Example.Create (u) IgnoreCase () EnableLike (MatchMode.Anywhere)...;

        getSession return () . CreateCriteria (typeof (User))
          . Add (exampleUser)
          . <User> List ();
    }

```

A chamada para `Create()` retorna uma nova instância `Exemplo` para a instância de dado Usuário. O `IgnoreCase()` método coloca a consulta de exemplo em um modo de maiúsculas e minúsculas para todos os cadeias de valor

propriedades. A chamada para `EnableLike()` especifica que o SQL como operador deve ser usado para todos os cadeias de valor de propriedades, e especifica um `MatchMode`.

Nós significativamente simplificado o código novamente. A melhor coisa sobre NHibernate Exemplo consultas é que um `Exemplo` é apenas um simples `ICriterion`. Assim, você pode misturar e combinar livremente com QBE QBC.

Vamos ver como isso funciona restringindo ainda mais os resultados da pesquisa para usuários com uns `Items`. Para este fim, somamos um `ICriteria` para o usuário exemplo, restringir o resultado usando a sua `Items` coleção de `Items`:

```

    publica <User> IList findusers (User u) {

        = Exemplo exampleUser
          Example.Create (u) IgnoreCase () EnableLike (MatchMode.Anywhere)...;

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

getSession return () . CreateCriteria (typeof (User))
    . Add (exampleUser)
        . CreateCriteria ("Itens")
            . Add (Expression.IsNotNull ("SuccessfulBid"))
    . <User> List ();
}

}

```

Melhor ainda, podemos combinar Usuário propriedades e Item propriedades na mesma busca:

```

público IList findusers <User> (User u, ponto i) {

    = Exemplo exampleUser
    Example.Create (u) IgnoreCase () EnableLike (MatchMode.Anywhere)...;

    Exemplo exampleItem =
    Example.Create (i) IgnoreCase () EnableLike (MatchMode.Anywhere)...;

    getSession return () . CreateCriteria (typeof (User))
        . Add (exampleUser)
            . CreateCriteria ("Itens")
                . Add (exampleItem)
    . <User> List ();
}

}

```

Neste ponto, nós convidamos você a voltar atrás e considerar o quanto código seria necessário para implementar Nesta tela de pesquisa usando SQL handcoded / ADO.NET. É sim um monte, se listamos aqui que se estenderia para páginas.

7.5.2 Coleta de filtros

Você geralmente deseja executar uma consulta contra todos os elementos de uma coleção particular. Por exemplo, poderíamos ter uma Item e gostaria de recuperar todas as propostas para esse item em particular, ordenados por quantidade de

a oferta. Nós já sabemos uma boa maneira de escrever essa consulta:

```

session.CreateQuery (@ "b Oferta de onde b.Item =: item
                        por fim asc b.Amount ")
    . SetEntity ("item", item)
    . List ();

```

Esta consulta funciona perfeitamente, já que a associação entre as propostas e os itens é bidirecional e cada Oferta conhece a sua Item. Imagine que esta associação era unidirecional: Item tem uma coleção de Ofertas, mas não existe uma associação inversa de Oferta para Item.

Poderíamos tentar a seguinte consulta:

```

query string = @ bid "selecionar item Item juntar lance item.Bids
                        onde = item: item por ordem asc bid.Amount ";

Resultados IList session.CreateQuery = (query)
    . SetEntity ("item", item)
    . List ();

```

Esta consulta é ineficiente, ele usa uma junção desnecessários. Uma solução melhor, mais elegante é usar um coleta de filtro: uma consulta especial que pode ser aplicado a uma coleção persistente ou array. É comumente usado para restringir ainda mais ou ordem de um resultado. Vamos utilizá-lo em um já carregado Item e sua coleção de lances:

```

Resultados IList = session.CreateFilter (item.Bids,
                                         "Order by asc this.Amount")
    . List ();

```

Este filtro é equivalente à primeira consulta mostrada anteriormente e resulta em SQL idênticas. Filtros de recolha de ter uma implícita a partir de cláusula e uma implícita onde condição. O alias este refere-se implicitamente a elementos da coleção de propostas.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Filtros de recolha de NHibernate são não executado na memória. A coleção de propostas pode ser inicializado quando o filtro é chamado e, em caso afirmativo, permanecerão não inicializado. Além disso, os filtros não se aplicam às coleções transitória ou resultados da consulta, pois eles só podem ser aplicadas a uma coleção persistente atualmente referenciado por um objeto associado com a sessão NHibernate.

A cláusula necessária apenas de uma consulta HQL é a partir de. Desde um filtro de coleção tem um implícito a partir de

```
Resultados IList = session.CreateFilter (item.Bids, "") List ();
```

cláusula, o seguinte é um filtro válido:

Para a grande surpresa de todos (incluindo o designer desse recurso), este filtro trivial acaba por ser útil! Você pode usá-lo para paginar elementos da coleção:

```
Resultados IList = session.CreateFilter (item.Bids, "")
    . SetFirstResult (50)
    . SetMaxResults (25)
    . List ();
```

Normalmente, no entanto, usaria uma por fim com consultas paginado.

Mesmo que você não precisa de um a partir de cláusula em um filtro de coleção, você pode incluir um se quiser. A coleta de filtro não precisa mesmo de retornar elementos da coleção que está sendo filtrado. A próxima consulta retorna qualquer Categoria com o mesmo nome como uma categoria na coleção de dados:

```
cadeia filterString =
    "Selecionar outras de outra categoria, onde this.Name = other.Name";

Resultados IList =
    session.CreateFilter (cat.ChildCategories, filterString)
    . List ();
```

A seguinte consulta retorna uma coleção de Usuários que lance no item:

```
Resultados IList =
    session.CreateFilter (item.Bids,
        "This.Bidder selecionar")
    . List ();
```

A próxima consulta retorna todos esses lances dos usuários (incluindo os de outros itens):

```
Resultados IList session.CreateFilter = (
    item.Bids,
    "Elementos select (this.Bidder.Bids)")
    . List ();
```

Note que a consulta usa a HQL especial `elements()` função (explicado mais tarde) para selecionar todos os elementos de uma coleção.

A razão mais importante para a existência de filtros de recolha é permitir que o aplicativo recuperar alguns elementos de uma coleção sem inicializar a coleção inteira. No caso da muito grandes coleções, isto é importante para obter um desempenho aceitável. A consulta a seguir recupera todas as propostas feitas por um usuário na última semana:

```
Resultados IList =
    session.CreateFilter (user.Bids,
        "Onde this.Created >: oneWeekAgo")
    . SetDateTime ("oneWeekAgo", oneWeekAgo)
    . List ();
```

Novamente, esta consulta não inicializar o Lances cobrança da Usuário.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

7.5.3 Subconsultas

Subselects são uma característica importante e poderosa de SQL. Um subselect é uma consulta seleção embutidos em outra consulta, geralmente na cláusula `SELECT` ou `WHERE`.

HQL suporta subqueries na cláusula `WHERE`. Não podemos pensar de muitos bons usos para subconsultas em HQL. Apesar de serem possíveis, usar subqueries na cláusula `WHERE` pode ser uma futura extensão agradável. (Você pode lembrar-se do capítulo 3, seção 3.4.2, que um bens provenientes mapeamento é de fato um subselect.) Note que algumas plataformas suportadas pelo NHibernate não implementam subselects. Se você deseja portabilidade entre muitos bancos de dados diferentes, você não deve usar este recurso.

O resultado de uma subconsulta pode conter uma única linha ou várias linhas. Tipicamente, subqueries que retornam uma única linha executar agregação. A subconsulta a seguir retorna o número total de itens vendidos por um usuário, a consulta externa retorna todos os usuários que já venderam mais de 10 itens:

```
usuário de u onde 10 <(
    select count (i) de i u.Items onde i.SuccessfulBid não é nulo
)
```

Este é um exemplo de correlacionada subconsulta-it refere-se a um alias (`u`) A partir da consulta externa. A subconsulta seguinte é uma subconsulta não correlacionada:

```
Oferta de lance em que bid.Amount + 1 > =
    select max (b.Amount) de b Licitação
)
```

A subconsulta neste exemplo retorna o valor do lance máximo em todo o sistema, a consulta externa retorna todas as propostas cujo valor está dentro de um (dólar) desse montante.

Note que em ambos os casos, a subconsulta é colocada entre parênteses. Este é sempre necessário.

Subqueries correlacionadas são inofensivos, não há razão para não usá-las quando conveniente, embora sempre pode ser reescrita como duas consultas (afinal, eles não fazem referência entre si). Você deve pensar mais cuidadosamente sobre o impacto no desempenho de subconsultas correlacionadas. Em um banco de dados, o custo de desempenho de uma subconsulta correlacionada simples é semelhante ao custo de uma junção.

No entanto, não é necessariamente possível reescrever uma subconsulta correlacionada com várias consultas em separado.

Se uma subconsulta retorna várias linhas, é combinado com quantificação. ANSI SQL (e HQL) define os quantificadores seguintes:

- todos
- alguns (Um sinônimo para qualquer)
- em (Um sinônimo para = Qualquer)

Por exemplo, a seguinte consulta retorna itens onde todas as propostas estão a menos de

100:
do item item onde 100 > all (select b.Amount de item.Bids b)

A próxima consulta retorna todos os itens com lances superiores a 100:
do item item onde 100 <qualquer (select b.Amount de item.Bids b)

Essa consulta retorna itens com um lance de exatamente 100:
do item item onde 100 = alguns (selecionar b.Amount de item.Bids b)

Assim como esta:

```
do item item onde 100 em (select b.Amount de item.Bids b)
```

HQL suporta uma sintaxe de atalho para subqueries que operam em elementos ou índices de uma coleção. O consulta a seguir usa o HQL especial `elements()` função:

```
IList list = session.CreateQuery (@ "da categoria c
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        onde: item em elementos (c.Items) "
    . SetEntity ("item", item)
    . List ();

```

A consulta retorna todas as categorias ao qual o item pertence e é equivalente ao HQL seguinte, onde a subconsulta é mais explícito:

```

Resultados IList = session.CreateQuery (@ "da categoria c
                                         onde: item (a partir de c.Items) ")
    . SetEntity ("item", item)
    . List ();

```

Juntamente com elements (), Fornece HQL índices (), maxelement (), minelement (), maxindex () , minindex () E size (), Cada um dos quais é equivalente a uma subconsulta correlacionada certa contra o passou coleção. Consulte a documentação do NHibernate para mais informações sobre esses especiais funções, eles são raramente utilizados.

Subconsultas são uma técnica avançada, você deve questionar a sua utilização frequente, uma vez que consultas com subqueries muitas vezes pode ser reescrito usando somente junta e agregação. No entanto, eles são poderosos e úteis de vez em quando.

Por agora, esperamos que você está convencido de que as instalações de consulta do NHibernate são flexíveis, poderosos e fácil de usar. HQL fornece quase todas as funcionalidades do padrão ANSI SQL. É claro que, em raras ocasiões você fazer necessário recorrer a handcrafted SQL, especialmente quando se pretende aproveitar características do banco de dados que vão além da funcionalidade especificada pelo padrão ANSI.

7.6 SQL Native

Podemos pensar em alguns bons exemplos porque você pode usar SQL nativas no NHibernate: HQL não prevê qualquer mecanismo para especificar dicas de consulta SQL, ele também não suporta consultas hierárquicas

(Como o Oracle CONNECT BY cláusula) e pode ser necessário para rapidamente porto código SQL para o seu aplicação. Supomos que você vai tropeçar em outros exemplos.

Nestes casos relativamente raros, você está livre para recorrer ao uso da API ADO.NET diretamente. No entanto, isso significa escrever o código tedioso com a mão para transformar o resultado da consulta a um objeto gráfico. Você pode evitar todo esse trabalho usando apoio do NHibernate built-in para consultas SQL nativas.

NHibernate permite que você execute consultas arbitrary SQL para recuperar valores escalares ou mesmo entidades. Essas consultas podem ser escritos em seu código C # ou em seus arquivos de mapeamento. Neste último caso, também é possível chamar procedimentos armazenados. Você pode até mesmo substituir os comandos SQL que

NHibernate
7.6.1 Usando a API ISQLQuery
gera para as operações de CRUD. Todas estas técnicas wil ser abordados nas páginas seguintes.
ISQLQuery instâncias são criadas, chamando o método ISession.CreateSQLQuery (), Passando em um String de consulta SQL. Depois, você pode usar os métodos de ISQLQuery para fornecer mais detalhes sobre o seu consulta.

Uma consulta SQL pode retornar valores escalares das colunas individuais, uma entidade completa (juntamente com o seu associações e coleções) ou várias entidades. Ele também suporta todos os recursos de consultas HQL que significa que você pode usar parâmetros e paginação, etc

Consultas escalares e entidade

O mais simples consultas nativas são consultas escalar. Por exemplo:

```

Resultados IList session.CreateSQLQuery = ("SELECT * FROM ITEM")
    . AddScalar ("item_id", NHibernateUtil.Int64)
    . AddScalar ("NOME", NHibernateUtil.String)
    . AddScalar ("CRIADO", NHibernateUtil.Date)
    . List ();

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Esta consulta não retornará objetos Item, mas ele irá retornar as colunas especificado de todos os itens como arrays de objetos (object []). Estas colunas substituir o * no SELECIONE.

Se você quiser recuperar entidades, você pode fazer isso:

```
Resultados IList <Item> session.CreateSQLQuery = ("SELECT * FROM ITEM")
    . AddEntity (typeof (ponto))
    . <Item> List ();
```

Você também pode gerenciar coleções de associações e juntando -los. Vamos ver como podemos ansiosamente carga os itens com os seus vendedores:

```
Resultados IList <Item> session.CreateSQLQuery = (
    @ "SELECT item.id, NAME, CRIADO, SELLER_ID, ...
        USER_ID, FIRSTNAME, ...
        ITEM DE i, u USER,
        ONDE i.SELLER_ID = u.USER_ID ")
    . AddEntity ("item", typeof (Item))
    . AddJoin ("item.Seller")
    . <Item> List ();
```

Neste caso, a consulta é mais complexo porque temos de especificar as colunas das duas tabelas. Devemos também especificar o alias "item" em AddEntity (), a fim de juntar-se a sua Vendedor.

Aqui está como você pode tentar se juntar ao Lances coleta dos itens:

```
Resultados IList <Item> session.CreateSQLQuery = (
    @ "SELECT i.ITEM_ID, NAME, criado, ...
        BID_ID, CRIADO, b.ITEM_ID, ...
        ITEM DE i, b BID,
        ONDE i.ITEM_ID = b.ITEM_ID ")
    . AddEntity ("item", typeof (Item))
    . AddJoin ("item.Bids")
    . <Item> List ();
```

Esta consulta é semelhante ao anterior. Um bom conhecimento de SQL foi o suficiente para escrevê-lo e tomar cuidado com o nome da coluna ambígua Item_id. No entanto, infelizmente isso não vai funcionar no NHibernate, ele não reconhecerá i.ITEM_ID porque não é especificado como aquele no arquivo de mapeamento ou atributos.

É hora de introduzir um novo truque que irá resolver este problema. Demonstramos isso na próxima seção que explica como recuperar muitos tipos de entidade em uma única consulta.

Várias consultas entidades

Consultando mais de uma entidade aumenta a chance de ter duplicações nome da coluna. Felizmente, este problema é simples de resolver usando espaços reservados.

Espaços reservados são necessárias porque um resultado da consulta SQL pode retornar ao estado de entidade múltipla casos em cada linha, ou até mesmo o estado de várias instâncias da mesma entidade. O que precisamos é de uma forma de distinguir entre as diferentes entidades. NHibernate usa é esquema de nomenclatura própria, onde aliases de coluna são colocados no SQL mapear corretamente os valores da coluna para as propriedades de particulares instâncias. No entanto, ao usar nosso próprio SQL não queremos que o usuário tem que compreender tudo isto, Em vez disso, as consultas SQL nativas são especificados com espaços reservados para os aliases de coluna, que são muito mais simples.

A seguinte consulta SQL nativo mostra o que esses espaços reservados, os nomes entre parêntesis-parcer:

```
Resultados IList <Item> session.CreateSQLQuery = (
    @ "SELECT i.ITEM_ID como {} item.id,
        i.NAME como {} item.Name,
        i.CREATED como {} item.Created,
        ...
        FROM ITEM i ")
    . AddEntity ("item", typeof (Item))
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
. <Item> List ();
```

Cada espaço reservado especifica um nome de propriedade HQL-style. E, devemos fornecer a classe de entidade que é referido por item nos espaços reservados.

Aqui está um atalho, se você não quiser especificar todas as colunas explicitamente:

```
Resultados IList <Item> session.CreateSQLQuery = (
    @ "SELECT {item} .*
        FROM ITEM ")
    . AddEntity ("item", typeof (Item))
    . <Item> List ();
```

O {Item} .* espaço reservado é substituído por uma lista de nomes de colunas mapeadas e coluna correta aliases para todas as propriedades do Item entidade.

Agora, vamos ver como podemos voltar várias entidades:

```
Resultados IList session.CreateSQLQuery = (
    @ "SELECT {item}.*, {user}.*
        FROM ITEM i INNER JOIN USUÁRIO u
        ON = i.SELLER_ID u.USER_ID ")
    . AddEntity ("item", typeof (Item))
    . AddEntity ("user", typeof (User))
    . List ();
```

Esta consulta irá retornar tuplas de entidades, como de costume, NHibernate representa uma tupla como uma instância de

object [].

Tal como acontece com HQL, é geralmente recomendado para manter essas consultas fora do seu código, escrevê-los em

seus mapeamentos como discutido a seguir.

7.6.2 Named consultas SQL

Consultas nomeadas são consultas SQL definido nos arquivos de mapeamento do NHibernate. Aqui está como você pode reescrever o exemplo anterior:

```
name="FindItemsAndSellers"> <sql-query
    <Voltar class="Item"/> alias="item"
    <Voltar alias="user" class="User"/>
    <! [CDATA [
        SELECLONE {item}.* , {user}.*
        FROM ITEM i INNER JOIN USUÁRIO u
        ON = i.SELLER_ID u.USER_ID
    ]]>
</ Sql-query>
```

Esta consulta nomeada pode ser executado a partir do código da seguinte forma:

```
Resultados IList = session.GetNamedQuery ("FindItemAndSellers")
    . List ();
```

Comparando a consulta nomeada para a versão em linha discutido anteriormente, podemos ver <return> elemento substitui o método AddEntity ().<return-join> é utilizado para associações e coleções, e <return-scalar> para retornar valores escalares. Você também pode carregar uma coleção usando apenas <Load-coleção>.

Se você costuma retornar as mesmas informações, você pode exteriorizar-lo usando <resultset>. Aqui está uma exemplo completo:

```
<resultset name="FullItem">
    <Voltar class="Item"/> alias="item"
    <return-join alias="user" property="item.Seller"/>
    <return-join alias="bid" property="item.Bids"/>
    type="int"/> <return-scalar column="diff"
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

</ Resultset>

<sql-query name="FindItemsWithSellersAndBids" resultset-ref="FullItem">
    <! [CDATA [
        SELECCIONE {item.*}, {user.*}, {lance.*},
                    i.RESERVE_PRICE-i.INITIAL_PRICE como diff
        FROM ITEM i
        U USER INNER JOIN ON i.SELLER_ID = u.USER_ID
        LEFT OUTER JOIN b BID ON i.ITEM_ID = b.ITEM_ID
    ]]>
</ Sql-query>

```

Quando executado, essa consulta retornará tuplas contendo um item com o seu vendedor e lances e um calculado o valor escalar (diff).

Note-se que <resultset> elementos também podem ser referidos no código usando o método ISQLQuery setResultSetMapping () .

Consultas nomeadas SQL permitem evitar a sintaxe {} e definir a sua própria coluna aliases. Aqui está um exemplo simples:

```

name="FindItems"> <sql-query
    <Voltar alias="item" class="Item">
        <return-property name="nome" column="MY_NAME"/>
        <return-property name="InitialPrice">
            <return-column name="MY_ITEM_PRICE_VALUE"/>
            <return-column name="MY_ITEM_PRICE_CURRENCY"/>
        </ Retorno propriedade>
    <Voltar />
    <! [CDATA [
        SELECCIONE i.ITEM_ID como {} item.id, ...
                    i.NAME como my_name,
                    i.INITIAL_PRICE como MY_ITEM_PRICE_VALUE,
                    i.INITIAL_PRICE_CURRENCY como MY_ITEM_PRICE_CURRENCY,
        FROM ITEM i
    ]]>
</ Sql-query>

```

Neste exemplo, também usamos o {} sintaxe para colunas que não deseja personalizar. No capítulo 6, seção 6.1.2, definimos preço inicial item como um tipo de usuário composto, de modo que este exemplo mostra como carregá-lo.

Desde o SQL nativo é fortemente acoplados às mesas reais mapeados e colunas, é altamente Recomendamos que você define todas as consultas SQL nativo no documento de mapeamento em vez de incorporar -los no código C #.

Usando procedimentos armazenados

Consultas NHibernate chamado SQL pode chamar procedimentos armazenados e funções desde a versão 1.2.0.

Se criar um procedimento armazenado como (usando um banco de dados SQL Server):

```

CRIAR FindItems_SP procedimento como
    SELECCIONE item_id, NAME, INITIAL_PRICE, INITIAL_PRICE_CURRENCY, ...
    ITEM DE

```

Podemos chamá-lo usando a consulta

assim:

```

name="FindItems"> <sql-query
    <Voltar alias="item" class="Item">
        <return-property name="id" column="ITEM_ID "/>
        <return-property name="nome" column="NAME"/>
        <return-property name="InitialPrice">
            <return-column name="INITIAL_PRICE "/>
            <return-column name="INITIAL_PRICE_CURRENCY"/>
        </ Retorno propriedade>
    ...

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<Voltar />
exec FindItems_SP
</ Sql-query>
```

Ao contrário do exemplo anterior, temos que mapear todas as propriedades aqui, e isso é óbvio que não pode NHibernate

injetar seus próprios aliases colunas.

Note-se que o procedimento armazenado deve retornar um resultset para ser capaz de trabalhar com NHibernate. Outro

limitação é que as associações e coleções não são suportadas; uma consulta SQL chamar uma armazenados procedimento só pode retornar valores escalares e entidades.

Finalmente, os procedimentos armazenados são banco de dados-dependentes. Portanto, o mapeamento para um servidor SQL

banco de dados não pode ser o mesmo que para um banco de dados Oracle.

Se, em alguns casos especiais, você precisa ainda mais controle sobre o SQL que é executado, ou se você quiser

para chamar um procedimento armazenado que não é suportado, NHibernate oferece-lhe uma maneira de obter um ADO.NET

conexão. A propriedade `session.Connection` retorna o ADO.NET atualmente ativa `IDbConnection`

a partir do `ISession`. Não é sua a responsabilidade de fechar essa conexão, apenas para executar qualquer SQL declarações que você gosta e depois continuar usando o `ISession` (E, finalmente, fechar o `ISession`). A mesma coisa é verdade para as transações, você não deve confirmar ou reverter essa conexão você mesmo (a menos que

7.6.3 Personalizando Create, Retrieve, Update, Delete

gerenciar completamente a conexão para NHibernate).

Na maioria dos casos, os comandos gerados pelo NHibernate para salvar as entidades são aceitáveis. No entanto,

pode acontecer que você precisa para executar uma operação específica e, portanto, substituir NHibernate SQL gerados. NHibernate permite que você especifique as instruções SQL para criar, atualizar e excluir operações.

Os comandos personalizados são escritos no mapeamento da classe em questão. Por exemplo:

```

name="Item"> <class
  ...
  <sql-insert>
    INSERT INTO produto (nome, ..., item_id) VALUES (UPPER (?), ...,?)
  </ Sql-insert>
  <sql-update> ATUALIZAÇÃO NAME SET ITEM = UPPER (?), ... ONDE item_id =? </ Sql-update>
  <sql-delete> exec DeleteItem_SP? </ sql-delete>
</ Class>
```

Neste exemplo, `<sql-insert>` e `<sql-update>` respectivamente, guardar e atualizar um item com um costume lógica (a conversão de nomes em maiúsculas). E, como você pode ver na `<sql-delete>`, Estas personalizado comandos também podem chamar procedimentos armazenados. Neste último caso, a ordem dos parâmetros de posição

deve ser respeitado (como você pode ver aqui, o identificador é geralmente o último parâmetro). A ordem é definido por NHibernate e você pode lê-lo, permitindo que o log de depuração e lendo o SQL estática comandos que são gerados pelo NHibernate (lembre-se de fazer isso antes de escrever este o costume comandos).

Note-se que o costume `<sql-insert>` será ignorado se você usa identidade para gerar identificador valores para a classe. Seus comandos personalizados são necessários para afetar o mesmo número de linhas como

NHibernate gerado SQL faria. Você pode desativar esta verificação, adicionando `check = "none"` para o seu comandos.

Recuperar os comandos são definidos como consultas nomeadas. Por exemplo, aqui está uma consulta para carregar um item com bloqueio pessimista:

```

<sql-query name="LoadItem">
  <Voltar alias="item" class="Item" lock-mode="upgrade"/>
  SELECT {ponto} .*
  ITEM DE
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
    ONDE item_id =?
    FOR UPDATE
</ Sql-query>
```

Então, ele deve ser referenciado no mapeamento:

```
name="Item"> <class
  ...
    <loader query-ref="LoadItem"/>
</ Class>
```

Também é possível personalizar a forma como uma coleção deve ser carregado. Neste caso, a consulta será chamado

usar o `<load-collection>` tag. Aqui está um exemplo para o Lances coleção de Item:

```
name="LoadItemBids"> <sql-query
  <load-collection role="Item.Bids"/> alias="bid"
  SELECT {lance .}
  FROM BID
  ONDE item_id =: id
</ Sql-query>
```

Aqui está como ele é referenciado:

```
<bag name="Bids" ...>
  ...
    <loader query-ref="LoadItemBids"/>
<Saco />
```

Quando você está escrevendo consultas e testá-las em seu aplicativo, você pode encontrar um dos comuns problemas de desempenho com ORM. Felizmente, nós sabemos como evitar (ou, pelo menos, limitar) os seus impacto. Este processo é chamado otimizar a recuperação do objeto. Vamos examinar os problemas mais comuns.

7.7 recuperação de objetos Otimização

Desempenho tuning seu aplicativo deve primeiro incluir as configurações mais óbvias, como o melhor estratégias de busca e uso de proxies, como mostrado no capítulo 4. Note que consideramos permitindo que o cache de segundo nível a ser a otimização última vez que você costuma fazer.

O buscar junta, parte do tempo de execução de estratégias buscando, introduzido neste capítulo, merecem alguma atenção extra. No entanto, alguns problemas de design não podem ser resolvidos por meio do ajuste, mas só pode ser evitado se possível.

7.7.1 Resolvendo o problema de um n selecciona

O assassino maior desempenho em aplicações que persistir objetos para bancos de dados SQL é o n +1 seleciona problema . Quando você ajustar o desempenho de uma aplicação NHibernate, este problema é a primeira coisa você geralmente precisa resolver.

É normal (e recomendável) para mapear quase todas as associações para a inicialização lenta. Isso significa que você geralmente definido todas as coleções de `lazy = "true"` e até mesmo alterar algumas das one-to-one e muitos-to-one associações para não usar associações externas por padrão. Esta é a única maneira de evitar a recuperação de todos os objetos no banco de dados em cada transação. Infelizmente, esta decisão expõe a um n seleccióna problema. É fácil entender este problema, considerando uma consulta simples que recupera todos os Items para um usuário em particular:

```
Resultados IList<Item> session.CreateCriteria = (typeof (ponto))
  . Add (Expression.Eq ("item.Seller", user))
  . <Item> List ();
```

Essa consulta retorna uma lista de itens, onde cada conjunto de propostas é um wrapper coleção não inicializada. Suponha que agora querem encontrar o lance máximo para cada item. O código a seguir seria uma maneira de fazer isso:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

IList maxAmounts = new ArrayList ();
foreach (item item no resultado) {
    dupla maxAmount = 0;
    foreach (bid Bid em item.Bids) {
        if (maxAmount < bid.Amount)
            maxAmount = bid.Amount;
    }
    maxAmounts.Add (novo MaxAmount (item.Id, maxAmount));
}

```

Mas há um enorme problema com esta solução (além do fato de que esta seria muito melhor executada no banco de dados usando funções de agregação): Cada vez que acessar a coleção de licitações, NHibernate deve buscar esta coleção preguiçoso do banco de dados para cada item. Se a consulta inicial retorna 20 itens, toda a transação exige um inicial selecionar que recupera os itens mais 20 adicionais selecionarS para carregar o licitações coleções de cada item. Isso pode facilmente resultar em latência inaceitável numa sistema que acessa o banco de dados em uma rede. Normalmente você não criar explicitamente tais operações, porque você deve ver rapidamente fazê-lo é de qualidade inferior. No entanto, a um n seleciona problema muitas vezes é escondido na lógica mais complexa aplicação e você não pode reconhecê-lo, olhando para um único rotina.

A primeira tentativa para resolver este problema poderia ser a de permitir busca em lote. Nós mudamos nosso mapeamento para o licitações coleção para ficar assim:

Com busca em lote ativado, carregando o NHibernate próximos 10 itens quando a coleta é o primeiro acessado. Isto reduz o problema de $n + 1$ seleciona para $N/10 + 1$ seleciona. Para muitas aplicações, este pode ser suficiente para atingir a latência aceitável. Por outro lado, isso também significa que em alguns outros transações, as coleções são buscadas desnecessariamente. Não é o melhor que podemos fazer em termos de redução do número de idas ao banco de dados.

Uma solução muito melhor é aproveitar de agregação HQL e executar a obra de calcular o lance máximo no banco de dados. Assim evitamos o problema:

```

query string = @ "select nova MaxAmount (item.id, max (bid.Amount))
                  do item item juntar lance item.Bids "
                  onde item.Seller =: grupo de usuários por item.id ";

MaxAmounts IList session.CreateQuery = (query)
    . SetEntity ("user", usuário)
    . List ();

```

Infelizmente, esta solução não está completa para o problema genérico. Em geral, podemos precisar de fazer mais processamento complexo sobre as propostas do que simplesmente calcular o montante máximo. Nós preferimos fazer isso processamento na aplicação. NET.

Podemos tentar buscar permitindo ansiosos no nível do documento de mapeamento:

O outer-join atributo está disponível para coleções e outras associações. Ele força para NHibernate carregar a associação inteira, usando um SQL outer join. Você também pode usar o buscar atributo; fetch = "select" é equivalente a outer-join = "false" E fetch = "join" é equivalente a outer-join = "true".

Note que, como mencionado anteriormente, as consultas HQL ignorar o outer-join atributo, mas poderíamos estar usando uma consulta critérios.

Este mapeamento evita o problema na medida em que essa transação está em causa, estamos agora em condições de carregar todos os licitações na escolha inicial. Infelizmente, qualquer outra operação que recupera itens usando Get (), Load (), ou uma consulta critérios também vai recuperar todos os lances de uma vez. Recuperando dados desnecessários impõe extras

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

carga em ambos os banco de dados do servidor e do servidor de aplicativos e também pode reduzir a simultaneidade da sistema, criando muitos bloqueios de leitura desnecessária ao nível do banco de dados.

Por isso consideramos a busca ansiosa ao nível do arquivo de mapeamento a ser quase sempre um mau abordagem. O outer-join atributo de mapeamento de coleção é sem dúvida um dos misfeature NHibernate (Felizmente, é desativado por padrão). Ocasionalmente, faz sentido permitir que outer-join para um <Many-to-one> ou <one-to-one> associação (o padrão é automático, Ver capítulo 4, seção 4.4.6, "Ponto único associações "), mas nunca faria isso no caso de uma coleção.

Nossa solução recomendada para este problema é aproveitar o suporte para NHibernate tempo de execução (nível de código) as declarações de estratégias de associação busca. O exemplo pode ser implementado como este:

```
Resultados IList <Item> session.CreateCriteria = (typeof (ponto))
    . Add (Expression.Eq ("item.Seller", user))
    . SetFetchMode ("Licitações", FetchMode.Eager)
    . <Item> List ();

// Faz resultados distintos
ISet <Item> distinctResults = new HashSet <Item> (resultados);

IList maxAmounts = new ArrayList ();
foreach (item Item em distinctResults) {
    dupla maxAmount = 0;
    foreach (bid Bid em item.Bids) {
        if (maxAmount < bid.Amount)
            maxAmount = bid.Amount;
    }
    maxAmounts.Add (novo MaxAmount (item.Id, maxAmount));
}
```

Nós desabilitado busca em lote e buscar ansiosos no nível de mapeamento, a coleção é preguiçoso por padrão. Em vez disso, permitir que a busca ansiosa por esta consulta só chamando SetFetchMode (). Como discutido anteriormente neste capítulo, isto é equivalente a um buscar unir no a partir de cláusula de uma consulta HQL.

O exemplo de código anterior tenha uma complicação extra: A lista de resultados retornados pelo NHibernate consulta de critérios não é garantido que ser distinto. No caso de uma consulta que busca uma coleção com exterior

join, ele irá conter itens duplicados. É responsabilidade do aplicativo para que os resultados distintos se que é necessário. Nós implementamos isso adicionando os resultados para um HashSet (Da biblioteca Iesi.Collections) e depois iterar o conjunto.

Então, nós estabelecemos uma solução geral para o problema n seleciona um. Em vez de recuperar apenas os objetos de nível superior na consulta inicial e, em seguida, buscar associações necessários, como a aplicação navega o objeto gráfico, seguimos um processo de duas etapas:

- 1 Buscar todos os dados necessários na consulta inicial, especificando exatamente quais serão as associações acessada na unidade de trabalho seguinte.
- 2 Navegar no gráfico do objeto, que será composto inteiramente de objetos que já foram obtido a partir da base de dados.

Esta é a única verdadeira solução para a incompatibilidade entre o mundo orientado a objetos, onde os dados são acessados por navegação, e para o mundo relacional, onde os dados são acessados por aderir.

Outra solução eficiente, para gráficos de profundidade dos objetos, é a emissão de uma consulta por nível e deixar

NHibernate resolver as referências entre os objetos. Por exemplo, podemos consulta categorias, pedindo NHibernate para buscar seus itens. Então, podemos consultar esses itens, pedindo NHibernate para buscar suas propostas.

Finalmente, há uma outra solução para o problema n seleciona um. Para algumas classes ou coleções com um número suficientemente pequeno de casos, é possível manter todas as instâncias no segundo nível cache, evitando a necessidade de acesso ao banco. Obviamente, esta solução é o preferido onde e quando é possível (não é possível no caso do Lances de um Item, Porque não permitiria o cache para este tipo de dados).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O problema n +1 selecione podem aparecer sempre que usamos a List () método de IQuery para recuperar o resultado. Como mencionamos anteriormente, esta questão pode ser escondida em uma lógica mais complexa, é altamente

recomendar as estratégias de otimização mencionada no capítulo 4, seção 4.4.7, "Tuning recuperação de objeto" para encontrar tais cenários. Também é possível gerar muitos selecione usando Find (), O atalho para consultas sobre O ISession API, ou Load () e Get ().

Existe um método API terceira consulta não discutimos ainda. É extremamente importante entender quando é aplicável, porque produz n +1 selecione!

7.7.2 Usando Enumerable () consultas

O Enumeráveis () método da ISession e IQuery interfaces de comporta de forma diferente do que o Find () e List () métodos. É fornecido especificamente para deixá-lo aproveitar ao máximo a segunda cache de nível.

Considere o seguinte código:

```
IQuery categoryByName =  
    session.CreateQuery ("da categoria c, onde c.Name como: nome");  
categoryByName.SetString ("nome", categoryNamePattern);  
IList categorias categoryByName.List = ();
```

Este resultados da consulta em execução de uma SQL selecionar, Com todas as colunas do CATEGORIA tabela incluída no

O selecionar cláusula:

```
selecionar category id, NAME, parent id da categoria onde NAME como?
```

Se esperamos que as categorias já estão em cache na sessão ou cache de segundo nível, então nós só precisamos

o valor do identificador (a chave para o cache). Isto irá reduzir a quantidade de dados que temos para buscar a partir de

banco de dados: O SQL seguinte seria um pouco mais eficiente:

Podemos usar o Enumeráveis () método:

```
IQuery categoryByName =  
    session.CreateQuery ("da categoria c, onde c.Name como: nome");  
categoryByName.SetString ("nome", categoryNamePattern);  
IEnumerable <category> categorias = categoryByName.Enumerable <category> ();
```

A consulta inicial apenas recupera a categoria valores de chave primária. Em seguida, percorrer o resultado, NHibernate e olha para cima cada Categoria na sessão atual e no cache de segundo nível. Se um cache miss ocorre, NHibernate executa um adicional selecionar, Recuperando a categoria de seu principal chave da base de dados.

Na maioria dos casos, esta é uma otimização de menores. É geralmente muito mais importante para minimizar linha lê

do que para minimizar coluna lê. Ainda assim, se o objeto tem campos cadeia de grandes dimensões, esta técnica pode ser

úteis para minimizar os pacotes de dados na rede e, portanto, a latência.

Vamos falar sobre outra otimização, que também não é aplicável em cada caso. Até agora, nós só discutido cache os resultados de uma pesquisa por identificador (incluindo pesquisas implícita, tais como o carregamento de um

associação preguiçoso) no capítulo 5. Também é possível armazenar em cache os resultados das consultas

7.7.3 consultas caching

NHibernate

Para aplicações que executam muitas consultas e inserções poucos, exclusões ou atualizações, o cache de consultas pode

ter um impacto no desempenho. No entanto, se o aplicativo executa muitas gravações, a cache de consultas não poderão ser utilizadas de forma eficiente. NHibernate expira um conjunto de resultados em cache de consulta quando há qualquer inserir,

atualização ou exclusão de qualquer linha de uma tabela que aparece na consulta.

Assim como nem todas as classes ou coleções devem ser armazenados em cache, nem todas as pesquisas deve ser armazenado em cache ou será

beneficiar do cache. Por exemplo, se uma tela de pesquisa tem muitos critérios de pesquisa diferentes, então ele vai Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Não acontece muitas vezes o suficiente para que o usuário escolhe o mesmo critério muitas vezes. Neste caso, o cache

resultados da consulta serão subutilizadas, e que estaríamos melhor se não habilitar o cache para essa consulta.

Note que o cache de consultas não cache de entidades retornadas no conjunto de resultados da consulta, apenas o

valores identificador. NHibernate, no entanto, totalmente de cache os dados do valor digitado retornado por uma projeção

consulta. Por exemplo, a consulta de projeção "Select u, u.b.Created do Usuário, Bid b onde b.Bidder = U " resultará em cache dos identificadores dos usuários eo objeto data em que fez seus lances.

É da responsabilidade do cache de segundo nível (em conjunto com o cache de sessão) para o cache estado real das entidades. Assim, se a consulta em cache você acabou de ver é executado novamente, NHibernate terá

as datas de criação de oferta na cache de consulta, mas executar uma pesquisa na sessão e cache de segundo nível

(Ou até mesmo executar SQL novamente) para cada usuário que estava no resultado. Isto é similar à estratégia de pesquisa "hibernate.cache.use query cache" value="true" />

de Enumeráveis () . Como explicado na seção anterior.

No entanto, essa configuração não é suficiente para NHibernate para armazenar em cache os resultados da consulta. Por padrão,

Consultas NHibernate sempre ignorar o cache. Para habilitar o cache de consulta para uma consulta particular (para permitir

seus resultados para ser adicionado ao cache, e para permitir que ele desenhe seus resultados a partir de o cache), você usa o

```
IQuery categoryByName =
    IQuery interface:CreateQuery ("da categoria c, onde c.Name =: nome");
    categoryByName.SetString ("nome", categoryName);
    categoryByName.SetCacheable (true);
```

Mesmo isso não lhe dá granularidade suficiente, no entanto. Consultas diferentes podem exigir diferentes consulta políticas de expiração. NHibernate permite que você especifique um cache diferentes nomeado região para cada consulta:

```
IQuery userByName =
    session.CreateQuery ("de onde u usuário u.Username =: uname");
    userByName.SetString ("uname", nome de usuário);
    userByName.SetCacheable (true);
    userByName.SetCacheRegion ("UserQueries");
```

Agora você pode configurar as políticas de cache de validade usando o nome da região. Quando o cache de consulta é

habilitado, as regiões de cache são os seguintes:

- „ O padrão região cache de consultas, nulo
- „ Cada região chamada
- „ O cache de timestamp, NHibernate.Cache.UpdateTimestampsCache, Que é um especial região que detém timestamps das atualizações mais recentes para cada tabela

NHibernate usa o cache de timestamp para decidir se um conjunto de resultados em cache de consulta é obsoleto. NHibernate parece

no cache timestamp para o timestamp da inserção mais recente, atualização ou exclusão feita ao consultado mesa. Se for mais tarde do que o timestamp dos resultados da consulta em cache, então os resultados são armazenados em cache

descartada e uma nova consulta é emitida. Para melhores resultados, você deve configurar o cache para timestamp que o timestamp atualização para uma tabela não expira a partir do cache, enquanto consultas na tabela são ainda em cache em uma das outras regiões. A maneira mais fácil é desligar o termo para o cache timestamp.

Algumas palavras finais sobre otimização de desempenho: Lembre-se que questões como a um n seleciona problema pode atrasar a sua aplicação para o desempenho inaceitável. Tente evitar o problema usando a melhor estratégia de busca. Verifique se o seu objeto de recuperação técnica é o melhor para seu caso de uso antes de olhar para o cache de nada.

Do nosso ponto de vista, o cache de segundo nível é uma característica importante, mas não é o primeiro opção quando otimizando o desempenho. Erros no projeto de consultas ou uma parte desnecessariamente complexa

do seu modelo de objeto não pode ser melhorada com um "cache tudo" abordagem. Se um aplicativo executa apenas

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

a um nível aceitável com uma cache de hot (Um cache inteiro) depois de várias horas ou dias de tempo de execução, você deve verificar se há erros de projeto sério, consultas unperformant e problemas n +1 seleciona.

7.7.4 Utilizando profilers e Query Analyzer NHibernate

Na maioria dos casos, existem muitas maneiras de escrever uma consulta, e pode ser difícil para selecionar o ideal. Profiler

ferramentas podem ajudar a testar o desempenho destas opções. Então, use-os o mais rápido possível.

Ao trabalhar com consultas HQL, você também pode saber se o SQL gerado é ótima. Há um ferramenta chamada Query Analyzer NHibernate que permite que você dinamicamente escrever e executar consultas em seu modelo de domínio. Ele exibe a consulta SQL gerada em tempo real e exibir o resultado de seu consulta quando você executá-lo. Por isso, é também muito útil quando aprender HQL.

Formoredetails, refertothedocumentationonitswebsite:

<http://www.ayende.com/projects/nhibernate-query-analyzer.aspx>

Note-se que algumas consultas HQL neste capítulo não irá funcionar com a versão lançada do CaveatEmptor porque assumimos um tipo diferente de mapeamento foi utilizado em seus casos específicos, a fim de ilustrar funcionalidades específicas.

7.8. Sumário

Nós não esperamos que você sabe tudo sobre HQL e critérios depois de ler este capítulo uma vez. No entanto, o capítulo será útil como uma referência em seu trabalho diário com NHibernate, e nós incentivá-lo a voltar e reler seções sempre que você precisar.

Os exemplos de código neste capítulo mostram as três técnicas básicas de consulta NHibernate: HQL, Query por critérios (incluindo um mecanismo de consulta por exemplo), execução e direta de banco de dados específico SQL consultas.

Consideramos HQL o método mais poderoso. Consultas HQL são fáceis de entender, e eles usam classe persistente e nomes de propriedade em vez de nomes de tabela e coluna. HQL é polimórfico: Você pode recuperar todos os objetos com uma determinada interface por meio de consulta para essa interface. Com HQL, você tem a plena

poder de restrições arbitrárias e projeção de resultados, com operadores lógicos e chamadas de função apenas como em SQL, mas sempre no nível do objeto usando nomes de classe e propriedade. Você pode usar o nome parâmetros para ligar argumentos de consulta em uma forma segura e type-safe. Relatório de estilo de consultas também são

suportados, e esta é uma área importante em que as soluções ORM geralmente não possuem outras características.

A maioria deles é também verdade para os critérios com base em consultas, mas em vez de usar uma seqüência de consulta, você usa um typesafe API para construir a consulta. Se você usar ferramentas de refatoração como ReSharper, você também se beneficiam suas consultas de critérios a ser considerados em refactorings. Chamado objetos exemplo podem ser combinados

com critérios, por exemplo, para recuperar "todos os itens que se parecem com o exemplo dado."

A parte mais importante da recuperação de objetos é o carregamento de objetos associados, isto é, como você define a parte do objeto gráfico que você gostaria de carga do banco de dados em uma única operação.

NHibernate fornece preguiçoso, ansioso, e as estratégias de busca em lote, em metadados de mapeamento e dinamicamente em tempo de execução. Você pode usar associação se juntar e iteração resultado para evitar problemas comuns, tais como o n +1 seleciona problema. Seu objetivo é minimizar roundtrips banco de dados com muitas consultas pequenas, mas no mesmo tempo, você também tentar minimizar a quantidade de dados carregados em uma consulta.

A melhor estratégia de consulta e recuperação de objeto ideal depender do seu caso de uso, mas você deve ser bem preparados com os exemplos neste capítulo e tempo de execução excelente NHibernate busca estratégias.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

8

Desenvolvimento de Aplicações NHibernate

Este capítulo aborda

- Implementação de aplicações em camadas
- Resolução de problemas durante a configuração. NET usando o NHibernate
- Alcançar as metas de design
- Resolução de problemas de depuração e desempenho
- Serviços de integração como o log de auditoria
-
-

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Neste ponto, você pode estar pensando "Eu sei tudo sobre os recursos do NHibernate, mas como faço para colocá-los todos juntos para construir uma aplicação completa NHibernate? "É hora de nós responder a essa pergunta, e aplicar todo o conhecimento que adquirimos para implementar aplicativos como parte de um desenvolvimento do mundo real processo.

Se você lembrar, nós discutimos a arquitetura de uma aplicação desejada NHibernate no capítulo 1, secção 1.1.3, "Arquitetura em camadas". Isto proporcionou uma visão panorâmica das coisas grandes, mas é claro que precisamos de alguma forma obter a partir desse ponto de código executável de trabalho.

Discutiremos a camada por camada processo de desenvolvimento, mostrando a parte interna de cada camada, e como cada um deve lidar com as suas responsabilidades específicas, e como eles se comunicam uns com os outros.

Porque este livro centra-se em NHibernate, o modelo de domínio e camadas de persistência a maior parte de nossa atenção. No entanto, usando NHibernate exigirá decisões de projeto em toda a aplicação camadas, por isso vamos ter a certeza de dar detalhes nestes onde achamos que vai ajudar.

A complexidade de definir e construir sua aplicação em camadas depende da complexidade do problema em questão. Por exemplo, o nosso "Olá Mundo" no capítulo 2 seria relativamente trivial para você. No entanto, a construção de uma aplicação tão complexo como o exemplo CaveatEmptor pode revelar um pouco mais desafiador! Seguindo o conselho dado neste capítulo, você deve ser capaz de encontrar seu caminho através de todos os patches difícil. Note que vamos apresentar muitas idéias e padrões, e é importante que você entenda seus prós e contras para que você possa tomar as decisões corretas no direito lugares.

A primeira parte deste capítulo centra-se na implementação de uma aplicação NHibernate. Primeiro, nós vai redescobrir a arquitetura clássica de um aplicativo de NHibernate. Vamos falar sobre as suas camadas, suas propósitos e brevemente sobre suas implementações. Depois disso, vamos logo falar sobre o desenvolvimento de algum tipo de aplicações .NET; vemos lidar com as questões que você pode encontrar quando começar a escrever a aplicação ou quando implantá-lo. Finalmente, vamos discutir sobre o problema resolver no contexto de aplicações NHibernate. Esta parte também serve como um mapa para os próximos dois capítulos. Ele irá fornecer referências às seções próximo para mais detalhes.

A segunda parte deste capítulo é sobre os serviços: Como integrar componentes fracamente acoplados a um Aplicação NHibernate. Você vai aprender como usar o `IInterceptor` interface para integrar eficientemente serviços como registro de auditoria. Também vamos falar brevemente sobre algumas outras alternativas.

Vamos começar no início com a arquitetura de um aplicativo NHibernate e suas implementação.

8.1 Dentro das camadas de uma aplicação NHibernate

Capítulo 1 apresentou uma visão geral da arquitetura em camadas de uma aplicação NHibernate, mas você provavelmente ainda tem muitas perguntas sobre a sua implementação. Esses capítulos finais devem dar-lhe o respostas que precisa. Note que esta seção, muitas vezes, direcioná-lo para os capítulos seguintes para obter mais explicações detalhadas.

Vamos rever a importância da aplicação de camadas disciplinado que enfatizamos no primeiro capítulos deste livro. Camadas ajuda a alcançar separação de interesses, tornando o código mais legível pelo código do agrupamento que faz coisas semelhantes. Por outro lado, camadas tem um preço: Cada camada extra aumenta a quantidade de código necessário para implementar um simples pedaço de código de funcionalidade e muito mais faz com que a funcionalidade em si mais difícil de mudar.

Não vamos tentar formar qualquer conclusão sobre o número correto de camadas de usar (e certamente não sobre o que essas camadas deve ser) uma vez que o "melhor" design varia de aplicação para aplicação e uma discussão completa de arquitetura de aplicação é bem fora do escopo deste livro. Nós apenas observar que, em nossa opinião, uma camada só deve existir se for necessário, uma vez que aumenta a complexidade e os custos do desenvolvimento.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Nesta seção, nós vamos através das camadas introduzido no capítulo 1, explicando suas funções e discutindo suas implementações. Dessa forma, você vai aprender como desenvolver progressivamente um NHibernate aplicação.

Estas camadas são:

- A camada de negócios (com o modelo de domínio)
- A camada de persistência
- A camada de apresentação

Olhando para estas camadas, percebemos que apenas dois deles importa para o usuário final: A camada de negócios

é importante porque incorpora o problema da aplicação é suposto resolver. A camada de apresentação importa para o usuário, pois permite-lhes emitir comandos para a aplicação e ver os resultados.

A camada de persistência é algo que o usuário não se importa muito com diretamente, mas que é obviamente, essencial para a maioria das aplicações. É útil lembrar que a persistência é um serviço que é utilizados pela aplicação (como os apresentados na seção 9.4), e que está lá para permitir o carregamento e economia de um dado aplicações. Mantendo este ponto de vista em mente irá ajudá-lo a alcançar um bom separação de preocupações.

Se você pensar com cuidado sobre a implementação destas camadas, você verá que todos eles existem para aumentar o modelo de domínio de alguma forma. É por isso que dizemos que o processo de desenvolvimento de um

Aplicação NHibernate é domínio-centric, e esta abordagem é chamado Domain-Driven Development (DDD). Porque o teste é parte do processo de desenvolvimento, também falamos sobre o teste essas camadas e ver como podemos aplicar Test-Driven Development (TDD).

Antes de prosseguir, uma introdução aos padrões, DDD e TDD é necessária.

8.1.1 Padrões e metodologias

A amplitude deste capítulo nos obriga a mencionar muitos padrões e práticas relacionadas com software de desenvolvimento. Vamos dar uma breve introdução a cada um como nós discuti-los, mas se você achar que está em território desconhecido com nós encorajamos você a seguir todas as referências que damos para obter uma mais profunda compreensão dos temas.

No caso de você não estudou os padrões de antes, aqui é a idéia geral. Muitos problemas, embora diferentes em sua formulação, são resolvidos de uma maneira similar. Assim, é possível formular um resumo solução que pode ser aplicado para resolver estes problemas. Portanto, um padrão é uma descrição ou modelo recomendar uma solução para um problema recorrente. Centenas de padrões foram descobertos e documentado ao longo dos anos, e nós vamos dar referências aos livros e artigos que consideramos ser mais importante.

Um dos padrões mais famosa e popular é o Padrão Singleton, você já ouviu falar dele? Este padrão resolve problemas onde apenas uma instância de uma classe deve estar disponível para um sistema, e como essa instância pode ser globalmente acessível. Outro padrão que temos discutido já é "modelo de domínio" padrão, que padrão descreve uma maneira de capturar o comportamento e os dados de um sistema usando a colaborar classes, em vez de conjuntos de registro ou algum outro dispositivo.

Há muitos benefícios em padrões de aprendizagem: Eles estão bem testados, os métodos vez comprovada que permitem aproveitar a experiência de outros profissionais. Eles também nos dão vocabulário comum para eficientemente comunicar aspectos de nosso software com os outros. Vamos agora olhar para um padrão que está ganhando popularidade crescente no desenvolvimento de software mainstream, e que é altamente aplicável para NHibernate - o padrão Model View Controller.

O padrão MVC

Model / View / Controller (MVC) é um padrão populares comumente utilizados em ambos os web e desktop aplicações. Ela prescreve separa as preocupações aplicação em três categorias: Model View, e Controlador. Esta abordagem é ilustrada na figura 8.1.

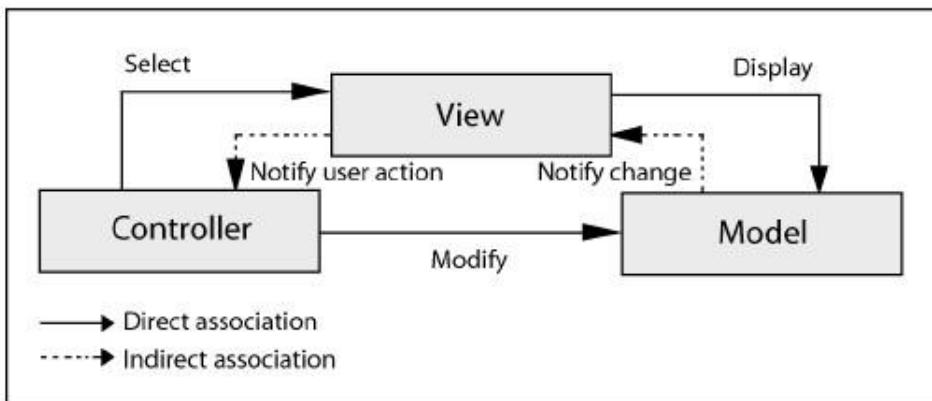


Figura 8.1 Modelo / View / Controller padrão

Vamos olhar mais de perto as peças-chave desta figura, começando com a View. The View tem tudo a ver apresentação de dados para o usuário, e recolher o seu de entrada (mouse cliques, etc entrada de dados). Ver o poderia

ser uma página web, uma forma de janelas, ou um menu-driven discurso de áudio como os encontrados em sistemas de telefonia.

Uma visão obtém seus dados de um ou mais modelos, assim o propósito de, essencialmente, uma visão na vida é para mostrar modelos e receber a entrada do usuário. Em muitos casos, estes modelos são os objetos de domínio em seu sistema,

que nós falamos muitas vezes ao longo deste livro. Então onde é que o controlador se encaixa em todos os isso? Bem, quando o usuário interage com a visão, a visão teria de notificar o controlador sobre aqueles interações. Poderia dizer o controlador sobre um clique do mouse, uma seleção drop-down list, ou alguns dados entrada. O controlador é o cérebro de coordenação entre o usuário eo modelo de domínio. Em muitos casos, o controlador poderia simplesmente emitir atualizações para o modelo, definir propriedades e chamar métodos.

As associações diretas definir as dependências entre esses componentes, o controlador sabe sobre a Vista eo Modelo, o Ver apenas diretamente sabe sobre o modelo, eo modelo é totalmente independente - que não sabe sobre os controladores ou exibições. O (linha pontilhada) indireta associações indicam que um componente pode comunicar-se com outro componente sem diretamente dependendo dela. Em. NET, eventos nos dá essa capacidade, e isso nos dá um sabor do Observador padrão. Para mais detalhes do que, ler o capítulo 10, seção 10.3.1.

Ao aplicar o padrão MVC para uma aplicação NHibernate, o modelo é representado pela entidades, o Ver é representado pela interface com o usuário mostrando o Model (em uma aplicação Web, é a página web). E, o controlador é o cérebro; ele usa entradas do usuário para processar a modelo e pedir ao Ver para mostrar o resultado.

Em. NET, é comum para espalhar o controlador em muitas camadas (apresentação e a camada de negócio). Ela exige uma certa disciplina para evitar colocar tudo no código por trás do página web ou windows form, no entanto, vale a pena o esforço, porque aumenta a capacidade de reutilização e testabilidade do seu código. A camada de apresentação deve conter apenas o código relacionado à formatação, exibir e recuperar informações. Se você está ciente do projeto MVC Microsoft, você pode perceber que visa proporcionar muitos desses benefícios.

Domain-Driven Development

Domain-Driven Development (DDD) é uma abordagem popular para evoluir modelos de domínio rico.

O objetivo do DDD é permitir que os clientes e desenvolvedores para trabalhar em conjunto com um mínimo de atrito.

Desenvolvedores e clientes estabelecer uma linguagem comum que eles usam para se comunicar conceitos em negócios em nível de conversas, e, além disso esta mesma linguagem é usada para retratar conceitos em o código do software. Portanto, a linguagem do software e os negócios são os mesmos, deixando pouco espaço para ambigüidade.

Tendo o usuário final e o desenvolvedor trabalhando em conjunto a velocidades de ida e volta entre a decisão decisões e na implementação.

Em termos práticos, o processo de desenvolvimento está centrada no modelo de domínio. Está escrito primeiro a certifique-se que o negócio é corretamente compreendido e representado. Os outros componentes e camadas são implementadas e adaptadas com base na evolução do modelo de domínio.

Todas as aplicações NHibernate de alguma forma seguir esta abordagem, até mesmo o "Olá Mundo", em capítulo 2, faz. Portanto, recomendamos que você para saber mais sobre essa metodologia, porque vai ajudá-lo a construir aplicações NHibernate. Para saber mais sobre DDD, referem-se aos livros em domínio Driven Design [Evans 2004] e aplicando Domain-Driven Design Patterns e [Nilsson 2006].

Test-Driven Development

Embora haja uma tendência a concentrar-se na lógica de negócios e implementação, teste também deve ser parte do processo de desenvolvimento. Test-Driven Development (TDD) vai tão longe a ponto de dar o primeiro passo, antes mesmo a implementação. Aplicando TDD significa pensar em uma solução para um problema, escrever os testes deste solução, e então implementar a solução, tornando a passar por testes. TDD não é popular com novas comers, no entanto, uma vez que são utilizados para o processo, você vê os benefícios no curto prazo e longo prazo.

Você só pode estar familiarizado com os testes de aplicativos clássico para garantia de qualidade onde você constrói uma característica, com sua interface do usuário, e então você testar esse recurso utilizando a interface do usuário. Embora este método é importante, não é suficiente em aplicações complexas. Você pode testar a interface do usuário cada vez que você mudar algo em

o modelo de domínio? Tem certeza que você vai testá-lo corretamente e não perca um buraco em algum lugar?

Aí vem o teste de unidade. Como você pode imaginar, é sobre o teste de cada único unidade (Geralmente uma classe).

Note-se que os testes são automatizados. Isto significa que você pode executá-los literalmente apenas pressionando em um botão e beber o seu café enquanto o computador funciona para você. E usando um modelo de domínio faz esta abordagem muito mais simples (devido à separação de interesses).

O quadro. NET tem algumas bibliotecas testes popular, sendo a mais popular do NUnit ([Http://nunit.org/](http://nunit.org/)). Este capítulo irá brevemente apresentar-lhe o teste de unidade do modelo de domínio e a camada de negócios usando esta biblioteca.

Como DDD, Esta metodologia é uma prática de Desenvolvimento Ágil de Software ([Http://en.wikipedia.org/wiki/agile_software_development](http://en.wikipedia.org/wiki/agile_software_development)). Ele está fora do escopo deste livro para escavar em estes temas. Para mais detalhes sobre NUnit e TDD, leia Test-Driven Development no Microsoft .NET [Newkirk et al 2004]. Você também pode dar uma olhada em ferramentas como o ReSharper ([Http://www.jetbrains.com/resharper/](http://www.jetbrains.com/resharper/)), que ajuda muito quando se aplica TDD.

Você pode aprender mais sobre padrões nos Padrões livros de Enterprise Architecture Aplicação [Fowler 2003] e Design Patterns [Gamma et al 1995].

8.1.2 Processo de desenvolvimento de uma aplicação

A separação clara das preocupações que nos permite desenvolver as camadas de uma aplicação quase de forma independente.

No entanto, como a arquitetura em camadas aponta, ainda há uma ordem na qual a camada deve ser implementada (com base no dependências). E é por isso que geralmente começam com o modelo de domínio; portanto, a aplicação de DDD.

Nas seções seguintes, vamos através do desenvolvimento de cada camada do modelo de domínio (Que não tem nenhuma dependência) para a camada de apresentação (que é a camada superior de acordo com todos os outros).

Há duas partes neste processo: a implementação e os testes. Ao falar sobre implementação de uma camada, vamos ver o que esta camada é feita e como seus elementos podem ser escritos. Depois disso, nós damos-lhe uma idéia de como esses elementos podem ser testados. Um bom entendimento do implementação é necessária a fim de testá-lo; é por isso que o teste vem depois. Depois de entender este processo, você está livre para aplicar TDD (escrever o primeiro teste, então a implementação).

Figura 8.2 ilustra o processo de desenvolvimento que estamos a seguir.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

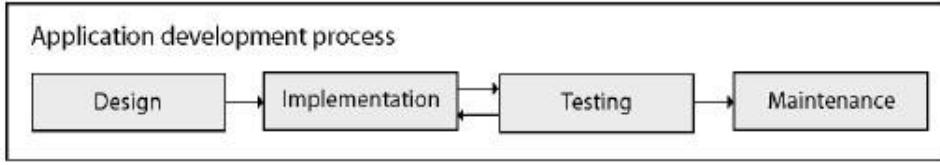


Figura 8.2 representação simplista de um processo de desenvolvimento de aplicações

Esta figura mostra que, primeiro, criar um aplicativo e, depois, implementamos e testá-lo. Estes passos são feito em um loop (como quando aplicar TDD). Uma vez que a aplicação é feita, a manutenção começa. Nota que, ao contrário nesta figura, o processo de desenvolvimento de uma aplicação do mundo real geralmente tem um laço grande

entre o projeto ea implementação / teste. A razão é que as exigências da aplicação evoluir como ele é desenvolvido.

O projeto de um aplicativo NHibernate tem sido discutido desde o capítulo 1 e algumas adições são feitas na seção 9.3.1. Vamos falar sobre a sua implementação (e testes) desta seção e na seguintes capítulos. E falamos sobre a manutenção no ponto 8.3.2.

Agora, podemos começar a falar sobre a implementação e testes do coração da aplicação: O modelo de domínio.

8.1.3 O modelo de domínio

O modelo de domínio é comumente percebido como parte da camada de negócio. No entanto, vamos discutir -los separadamente. O modelo de domínio é o coração do negócio. Ele diz que a empresa está fazendo (o domínio), e representa as entidades manipuladas no domínio, o negócio (o modelo).

É conceitualmente independente de qualquer outra camada, até mesmo da camada de negócios, uma vez que não utiliza nenhum

classe que não é em si parte do modelo de domínio. Você já deve estar familiarizado com ele como temos sido implementação de entidades desde o capítulo 2.

A aplicação CaveatEmptor tem um modelo de domínio mais complexas, como ilustrado no capítulo 4, figura 4.2. A sua aplicação requer nove classes para as entidades e ainda mais para os outros componentes.

Implementação

No contexto do desenvolvimento de aplicativos empresariais, implementando um modelo de domínio pode não ser tão

simples como parece. Esta secção apenas vai enumerar os passos deste processo, juntamente com alguns dos problemas que podem ocorrer, vamos cobri-lo profundamente no capítulo 10.

Primeiro, você pode escrevê-lo a partir do zero, ou você pode gerá-lo (a partir do mapeamento ou a partir do banco de dados). Neste caso, você pode ter que lidar com um banco de dados legado. Apesar de um banco de dados legado com

um desenho estranho pode fazer esta etapa complexa, é geralmente fácil, você só tem que escrever um conjunto de classes com construtores e propriedades.

Então, implementar o comportamento e as regras das entidades (a lógica de negócios) pode ser confuso se você não fazê-lo corretamente. Você deve ter cuidado ao escolher onde e como você implementá-los em Para evitar constrangimentos desnecessários.

Como se não bastasse, as outras camadas ainda pode ter uma influência sobre o modelo de domínio: Pode não ser completamente ignorante ou persistência pode implementar alguns mecanismos, por exemplo, para facilitar a de ligação de dados sobre a camada de apresentação.

Finalmente, você pode ter que se comunicar com outros componentes / serviços / aplicações que não podem manipular o seu modelo de domínio como está;. NET em aplicações, é muito comum ter um conjunto específico de classes (dados Objetos de transferência ou DataSet) como meio de comunicação. Neste caso, você deve converter suas entidades para que esta comunicação, e esta conversão pode ser bastante problemático. Você daria o seu modelo de domínio para evitar este problema?

Todos estes problemas exigem uma boa quantidade de pensar e de experimentação. Felizmente, o capítulo 10 oferece uma ampla gama de opções para resolvê-los.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Ensaio

Se os seus aplicativos anteriores não têm qualquer teste para o seu modelo de domínio, você realmente deve reconsiderar sua posição. Lembre-se, seu modelo de domínio é o coração do seu software e, ao contrário da camada de apresentação, quebrá-lo pode não ser facilmente visível.

Note que, quando a unidade testar seu modelo de domínio, as unidades são as entidades.

Há globalmente dois tipos de testes: testes de integridade de dados e testes de lógica. Vamos dar uma simples exemplo para cada um deles e ver como podemos fazer isso usando o NUnit. Para tornar ainda mais divertido, vamos aplicar TDD.

Testes de integridade de dados são testes simples para se certificar que o conteúdo das entidades são válidos (de acordo a lógica de negócio) e permanecer válido, não importa o que aconteça.

Por exemplo, se você tiver um Usuário entidade em seu modelo de domínio, seu nome será obrigatória e, porque ele é armazenado no banco de dados, este nome não deve exceder um número de caracteres (digamos 64).

Escrevemos três testes para codificar esta especificação na listagem 8.1.

Listagem 8.1 Unidade de teste de uma entidade

```
using NUnit.Framework;

[TestFixture]
UserFixture public class {

    [Test]
    WorkingName public void () {
        . sequência aleatória = new Random () Next () ToString ();
        Usuário u = new User ();
        u.Name = random;

        Assert.AreEqual (aleatória, u.Name);
    }

    [Test, ExpectedException (typeof (BusinessException))]
    NotNullName public void () {
        Usuário u = new User ();
        u.Name = null;
    }

    [Test, ExpectedException (typeof (BusinessException))]
    TooLongName public void () {
        Usuário u = new User ();
        . u.Name = "" PadLeft (65, 'x');
    }
}
```

1 Há um nome de propriedade mantendo seu valor

2 O nome não pode ser nulo

3 O nome não deve exceder 64 caracteres

Os testes são agrupados em classes de público chamado acessórios. NUnit usa atributos para identificá-los, é por isso

esta classe é decorada com o atributo [TestFixture]. Os testes são marcados usando métodos [Teste]. Note que estes métodos devem ser públicas e devem ter nenhum parâmetro.

O primeiro teste garante que existe uma propriedade chamada Nome na classe Usuário que mantém o valor nós atribuímos a ele. Nós usamos um valor aleatório para evitar fraudes. O teste real é feita usando Assert.AreEqual (...); A classe Afirma pertence ao NUnit e fornece uma ampla gama de métodos para testar diferentes.

Os dois outros testes usar uma abordagem diferente. Eles tentam definir valores inválidos para a propriedade Nome e

esperar que ele jogue um BusinessException. Se isso não acontecer, o teste falha. Como explicado no capítulo 10, seção 10.4.2, você pode permitir que valores inválidos e validar entidades alterado antes de salvá-los.

Lembre-se que esta classe deve ficar em um biblioteca de testes separados.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Agora, podemos passar para a implementação da parte da classe Usuário que fará com que esses três testes passem.

Primeiro, temos que criar a classe Usuário e sua propriedade Nome. Então usamos um campo `_name` para manter a valor atribuído a essa propriedade. Neste ponto, o primeiro teste passa.

Depois disso, nós adicionamos uma validação para fazer os dois outros testes passar. Aqui está o resultado final:

```
Usuário public class {

    _name private string;

    Nome {public string
        get {_name return;}
        conjunto {

            if (string.IsNullOrEmpty (valor) || value.Length > 64)
                throw new BusinessException ("nome inválido");

            _name = valor;
        }
    }
}
```

Este código é bastante fácil de entender: no setter da propriedade Nome, jogamos em exceção se o nome que está prestes a ser definido é inválido (nulo, vazio ou muito longo), então nós mantemos esse valor no campo `_name`.

O segundo tipo de testes para uma entidade é o teste de lógica que os testes qualquer comportamento na entidade. Para exemplo, ao alterar sua senha, o usuário deve fornecer o velho e duas vezes o novo.

Aqui estão alguns testes para este método: (vamos supor que um usuário recém-criado tem uma senha em branco

e que não há criptografia)

```
public void WorkingPassword () {
    . seqüência aleatória = new Random () Next () ToString () .;
    Usuário u = new User ();
    u.ChangePassword ("", aleatório, aleatório);

    Assert.AreEqual (aleatória, u.Password);
}

[Test, ExpectedException (typeof (BusinessException))]
public void BadOldPassword () {
    Usuário u = new User ();
    u.ChangePassword ("?", "", "");
}

[Test, ExpectedException (typeof (BusinessException))]
public void DifferentNewPasswords () {
    Usuário u = new User ();
    u.ChangePassword ("", "x", "y");
}
```

Há pouco a explicar. O primeiro teste ter certeza de que podemos mudar a senha com sucesso e os outros dois se certificar de que não podemos mudar a senha com uma senha inválida velho ou um novo senha diferente da senha de confirmação.

Aqui está uma implementação do método `ChangePassword()` que faz com que estes testes passam:

```
public void ChangePassword (string OLDPWD, string newpwd, string confirmPwd) {

    if ((_password = OLDPWD) != !(newpwd == confirmPwd))
        throw BusinessException nova (...);

    _password = newpwd;
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note que você provavelmente deve separar as duas validações para fornecer mensagens significativas na jogada exceção. E, no mundo real, você deve mover essas validações para a camada de negócio como que é feito na próxima seção.

O que você deve lembrar é que um modelo de domínio pode (deve) ser testado de forma eficiente. A sua autonomia torna muito fácil não só para implementar, mas também para testar.

Vamos continuar com as outras classes que formam o aspecto do negócio da aplicação.

8.1.4 A camada de negócios

A camada de negócios funciona como um porta de entrada que as camadas superiores (como a camada de apresentação) deve usar para manipular as entidades.

É comum ver .NET (principalmente aqueles que utilizam DataSet) Acessando diretamente o banco de dados. Você já sabe que é errado e por quê (se você se esqueceu, volte leitura do capítulo 1, seção 1.2, "Implementando persistência. NET").

A camada de negócios desempenha vários papéis: É uma camada em cima da camada de persistência. Ele executa alta nível lógico de negócio (que não pode ser realizada pelas próprias entidades). Pode também integrar serviços como segurança e log de auditoria.

O Controller (do padrão MVC) também pode ser percebido como sendo parte desta camada. Ele pilota o fluxo de informações entre o usuário final (através da View) e do Modelo. No entanto, é uma boa prática de manter os controladores como uma fina camada em cima do núcleo da camada de negócios.

Implementação

Dependendo do acoplamento entre as entidades, você pode escrever uma classe empresarial para gerir cada entidade ou de um para muitas entidades. Você também pode ter classes de negócio para alguns casos de uso / cenários.

Ao implementar as operações CRUD-like, não tente imitar a camada de persistência. A este nível, guardar e atualizar não são palavras de negócios. É realmente fácil de encontrar as palavras certas ao ver a problema de uma perspectiva de negócios. Por exemplo, quando colocamos um lance em um item, embora estejamos salvar essa oferta em banco de dados (teoricamente falando), não devemos chamar o método de executar este operação SaveBid (), Neste caso, PlaceBid () é mais expressiva. Obviamente, PlaceBid () (No camada de negócios) vai enviar a proposta para a persistência usando um método da camada de persistência que pode ser chamado Save () (Ou MakePersistent () como explicado no capítulo 11, seção 11.1).

Vamos mudar o nosso exemplo anterior para ilustrar um pedaço da camada de negócios. E se nós queremos permitir aos administradores alterar senhas dos usuários (sem conhecer os seus atuais)?

Neste caso, a classe Usuário Não é possível executar essa lógica, porque não sabe qual o usuário é atualmente conectado; a menos que você disponibilizar essas informações usando, por exemplo, o padrão Singleton
(No entanto, isso pode não ser uma boa idéia).

Aqui está como o método ChangePassword () podem ser implementadas: (agora, este método pertence a uma classe na camada de negócios)

```
public void ChangePassword (User u,
                           cadeia OLDPWD, string newpwd, string confirmPwd) {

    if (u! = null)
        throw new ArgumentNullException ("u");

    if (LoggedUser == null)
        throw new BusinessException ("Deve ser registrada");

    if ((! LoggedUser.IsAdministrator & &
         | | (! Newpwd confirmPwd =))
        throw BusinessException nova (...);

    u.Password = newpwd;

    UserPersister.Save (u); camada / Persistência
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Há muitas validações neste método, no entanto, note que nós não validar a nova senha; isso é até a propriedade User.password. By the way, você pode fazer o setter da propriedade interna (Fornecendo o modelo de domínio ea camada de negócios estão na mesma biblioteca) para ter certeza de que o camada de apresentação não pode alterar a senha diretamente.

Note-se que, em vez de receber uma instância da classe Usuário, Este método pode receber o usuário do identificador e carregá-lo internamente. As vantagens são que a camada de negócio é certo que nada mais tem alterado no usuário e que pode ser mais fácil para a camada de apresentação para fornecer um identificador (para exemplo, porque não manter a instância de usuário). No entanto, é menos orientada a objetos.

Sobre os métodos de consulta, a menos que você está escrevendo uma interface de usuário personalizável fortemente motor de busca, você deve fornecer métodos para todos os tipos de consultas que o usuário final pode ser executado. Não deixe que a apresentação camada de criar consultas arbitrárias. Faz a camada de apresentação cliente da camada de persistência, pode tornar-se um problema de segurança e torna a aplicação menos testáveis. Por outro lado, você deve usar uma abordagem extensível para evitar ter que criar métodos demais.

Você também pode incluir algum código de log de auditoria para acompanhar o que aconteceu e quem fez isso (Inestimável ao depurar um aplicativo em produção). No entanto, você vai aprender na seção 9.4 outra maneira de lidar com este tipo de serviços.

Falamos sobre a implementação da lógica de negócio no capítulo 10, seção 10.4. Não podemos falar sobre a aplicação geral, pois depende muito da aplicação. Basta fazer certeza de que você limpa separá-la das outras camadas. Você pode, por exemplo, colocar as classes de negócio em um Negócio namespace e os controladores em um Controladores namespace.

Também não podemos realmente dar muitos detalhes sobre a implementação dos controladores porque eles tendem a ser muito dependente da plataforma. Você vai descobrir, no apêndice C, uma biblioteca que permite escrever limpa controladores.

Ensaio

Como seria de imaginar, testar a camada de negócios é crucial. Na prática, é muito semelhante ao teste do modelo de domínio. Se você entendeu o teste de unidade, como ilustrado na listagem 9.1, você deve ser capaz de facilmente testar o método anterior ChangePassword () .

No entanto, porque esta camada utiliza a camada de persistência, pode se tornar um incômodo em alguns lugares.

Alguma lógica de negócios complexos pode exigir uma estratégia de persistência específico. Estes cenários borderline requerem um compromisso entre a separação de interesses e da facilidade de implementação e testes.

Os testes para a camada de negócio só deve ser relacionado para a camada de negócio em si. Você pode tomar um olhar para os testes da camada de persistência na próxima seção para ver o que não deve ser testado aqui.

8.1.4 A camada de persistência

Mesmo os testes para as entidades do modelo de domínio devem ser separados.

A camada de persistência fornece métodos CRUD para as entidades. Graças ao NHibernate, ele pode ser implementado como um serviço (isto é, não intrusiva) para o modelo de domínio.

No entanto, algumas bibliotecas como ActiveRecord (Cf. apêndice C) mesclá-lo no modelo de domínio. Nós geralmente vejo isso como uma má prática, no entanto, a razão por trás dessa escolha de design é a simplicidade: Quando

escrevendo um novo aplicativo que não é muito complexo (em termos de camadas de acoplamento e questões de integração),

ter um modelo de domínio persistência ignorante não é necessariamente um requisito.

De qualquer forma, recomendamos que você construir a camada de persistência separadamente e que escondê-la atrás

a camada de negócios. Mas nós concordamos que isso depende do seu estilo de programação.

Implementação A prática geral, ao implementar a camada de persistência, é escrever uma classe de persistência por entidade, comumente chamado EntityNameDAO. DAO significa Data Access Object. É um bem conhecido

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

padrão para implementar as camadas de persistência. Na listagem de código anterior, `UserPersister` poderia ter sido chamado `UserDAO`.

Classes de persistência têm métodos para operações CRUD simples e pode permitir que a camada de negócios para executar consultas personalizadas. Explicamos essa abordagem no capítulo 11, seção 11.1.2.

A camada de apresentação (e outras camadas superiores) não deve ser capaz de aceder a esta camada, a menos que o aplicação é simples, caso em que a camada de negócios não é necessário (no entanto, cuidado com os "simples" aplicações evoluindo e se tornando muito complexo). Se esta camada é a biblioteca mesmo que o negócio camada, suas classes podem ser feitas interna. Outra coisa, simplesmente evitar a inclusão de uma referência na biblioteca de apresentação à biblioteca de persistência.

Observe também que ele deve semântica do banco de dados abstrato (a não ser por razões de desempenho). A razão é que a camada de persistência é entre a camada de negócios e banco de dados, portanto, ele deve esconder a banco de dados para a camada de negócios.

Como o modelo de domínio, existem duas maneiras de testar a camada de persistência. Você pode testar a exatidão de o mapeamento entre o modelo de domínio eo banco de dados, e você também pode testar o lógica de persistência.

Testando o mapeamento significa certificar-se que o mapeamento NHibernate está correctamente escrito e os entidades estão corretamente carregados e persistiu. Na prática, isso implica salvar uma entidade com um random conteúdo, carregá-lo e certificar-se que o conteúdo não mudou.

Ao testar a lógica de persistência, a lógica é representada por qualquer código personalizando a maneira NHibernate funciona. Exemplo: As consultas (cláusula where, a ordenação, etc.) A idéia é fazer com que certeza de obter os dados como pretendido.

No entanto, evitar testes NHibernate em si. Existem testes NHibernate para isso. Vamos dar este teste como exemplo:

```
session1.Save (entidade);
Assert.IsNotNull (session2.Get <Entity> (id));
```

Este teste é inútil porque se `Save()` conseguiu, então a entidade foi salvo. Você terá um tempo duro se você não confia em NHibernate e todas as bibliotecas que você usa para fazer o seu trabalho. Por outro lado, você pode certifique-se que a carga adequada preguiçoso e opções em cascata estão habilitados.

By the way, no caso de você não notou, usamos duas sessões diferentes (`session1` e `session2`) porque `session1.Get <Entity> (id)` não atingiu o banco de dados, que fará uso do seu cache (primeiro nível) em seu lugar. Se a criação de duas sessões em um teste é muito caro para você, você pode usar `session1.Clear()` ou fornecer sua conexão com a própria base de dados chamando `sessionFactory.openSession(yourDbConnection)`.

Para mais detalhes sobre os caches, leia o capítulo 6, seção 6.3.

Note-se que testes de persistência são mais lentos do que outros testes por causa do custo de usar um banco de dados; embora você possa acelerá-las usando in-memory RDBMSs capazes como SQLite. Também é possível zombar do banco de dados, no entanto, os testes podem tornar-se menos significativa. Mocking é uma técnica que permite faking a component to avoid its dependencies. For more details, leia http://en.wikipedia.org/wiki/Mock_object.

Como a camada de persistência está escondido atrás da camada de negócios, você pode realmente testar a persistência camada através da camada de negócio. Embora os testes clutter camada de negócios com a "não relacionados" testes.

8.1.6 A camada de apresentação

pode ser aceitável para casos simples.

A camada de apresentação é basicamente a interface do usuário (e seu código-behind). Ele serve como uma ponte entre o usuário final ea camada de negócios.

Seu trabalho principal é o de formatar e exibir informações (entidades), que também recebe comandos e informações do usuário final para enviá-los para a camada de negócio (ou o controlador).

Implementação

Implementação da camada de apresentação é em grande parte fora do âmbito deste livro. No entanto, usando NHibernate pode ter algumas consequências sobre ele. Leia a próxima seção para mais detalhes sobre problemas de implantação de Aplicações .NET utilizando NHibernate.

Do ponto de vista do modelo de domínio, o maior problema é exibir e recuperar entidades. .NET fornece alguns dados poderoso mecanismos de ligação que pode ser difícil de alavancagem com um modelo de domínio.

Nós mostramos no capítulo 10, seção 10.5, que existem muitas alternativas para ligar dados de entidades.

, A fim de mostrar como um método de manipulação de comando típico pode parecer, vamos implementar o método que pode ser chamado quando o usuário final, clique no botão para alterar uma senha.

```
private void btnChangePassword_Click (object sender, EventArgs e) {  
    try {  
  
        Business.ChangePassword (editedUser,  
                               editOldPwd.Text, editNewPwd.Text, editConfirmPwd.Text);  
  
        MessageBox.Show ("Sucesso!");  
        // Ou ir para a página de sucesso  
    }  
    catch (Exception ex) {  
  
        MessageBox.Show ("Falha:" + ex.Message);  
        // Ou ir para a página não  
    }  
}
```

Este método simplesmente envia as informações para a camada de negócios e exibe uma mensagem depois. Note-se que

este método faz o papel do Controller. Estritamente falando, é necessário apenas chamar o controlador que Em seguida, fazer exatamente o que é neste método. Este é um exemplo de situações em que o code-behind é usados para implementar a lógica do controlador, o que significa que o controlador não faz parte da camada de negócios, mas é mesclado na camada de apresentação.

Nós usamos o nome genérico Negócio para a classe empresarial porque seu nome pode variar amplamente dependendo da maneira como você organiza a sua camada de negócios.

By the way, você deve provavelmente registrar erros, que pode poupar muito trabalho ao tentar descobrir o que aconteceu em um aplicativo em produção.

Ensaio

A camada de apresentação é o mais difícil de testar automaticamente. A razão é que é algo inherentemente visuais, daí a forma mais comum de testá-lo é através da execução do aplicativo e ver se funciona.

Por outro lado, se você escrever a sua aplicação como descrito neste capítulo, a camada de persistência deve consistir no código de design (HTML para aplicações Web) e uma fina code-behind para a formatação e de ligação de dados. Este código pode ser facilmente testado visualmente: não é complexa e é mais difícil de quebrar.

Note-se que existem algumas técnicas e bibliotecas para testar a camada de apresentação. No entanto, iremos notcoverthem.Formoredetails, startbyreading http://en.wikipedia.org/wiki/List_of_GUI_testing_tools.

Ao implementar essas camadas, você pode encontrar alguns problemas ao tentar fazer o seu NHibernate aplicativo funcione com algumas características NET;. Vamos ver como você pode resolvê-los.

8.2 Resolver questões relacionadas com a.

Aplicativos usando o .NET tendem a usar alguns recursos .NET que pode ser problemático por causa da algumas restrições na forma como o trabalho ou devido a restrições de segurança.

Nesta seção, nós fornecemos falar de duas características específicas: trabalhar em um ambiente web e usando .NET Remoting.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

8.2.1 aplicação Web

Aplicações web são muito mais restritas do que as aplicações Windows. Eles têm uma estrutura específica e eles devem seguir as regras de muitos. Esta secção tem como objectivo proporcionar orientações sobre como você deve desenvolver uma aplicação com uma interface web.

Primeiro, vamos começar com o desenvolvimento de aplicações Web, então vamos falar sobre um determinado título emissão.

Início Rápido

Você não deve ter grandes problemas de iniciar um aplicativo usando páginas da Web como camada de apresentação. Em

Para atingir uma boa separação de preocupações, as outras camadas não deve ser na aplicação Web; em termos práticos, isso significa que o code-behind de páginas e App_Code seção deve conter apenas a lógica da camada de apresentação (ou seja, a formatação, exibindo, recuperar informações e chamando o outro camadas). E as outras camadas podem estar em uma biblioteca (ou muitos, se necessário). Note que este é também bastante verdadeiro para aplicações Windows.

Existem dois problemas comuns ao implementar a camada de apresentação de um NHibernate aplicação: de vinculação de dados das entidades (e suas coleções) e gerenciamento de sessão de forma eficiente persistirem essas entidades. Falamos de ligação de dados no capítulo 10, seção 10.5 e gerenciamento de sessão no capítulo 11.

Código de segurança de acesso

O quadro. NET tem uma política de segurança poderoso. Ele permite, por exemplo, os administradores da Web para limitar a quantidade de permissões dadas para montagens com base em sua origem. Isto é necessário porque as aplicações Web são geralmente acessíveis por um monte de pessoas incontroláveis;, portanto, eles devem ser cuidadosamente configurado para evitar problemas de segurança.

Servidores web são geralmente configurados para confiança média. Então, se você pretende implantar seu aplicativo em um público de serviços de hospedagem, você pode ter alguns problemas de fazê-la funcionar.

A política de confiança média, impõe algumas restrições que afetam os aplicativos NHibernate: Você não pode usar a reflexão para acessar membros privados de uma classe e você não pode usar proxies, porque eles não podem ser gerado devido à segurança apertada.

Portanto, você deve mapear o banco de dados para campos públicos / propriedades, você deve desativar o carregamento lento definindo Lazy = false no mapeamento de cada classe e você deve desligar o otimizador de reflexão.

Leia o capítulo 3, seções 3.5.3 e 3.5.4 para mais detalhes.

No caso em que suas propriedades estão em configuração NHibernate Web.config, Você deve adicionar um extra atributo para a declaração de sua seção: "... requirePermission = "false" />

Definindo o atributo requirePermission para falso NHibernate permite ler esta seção, quando carregar a configuração.

8.2.2 Remoting. NET

Muitas aplicações NHibernate uso. NET Remoting para tornar o negócio (ou persistência) da camada acessível através de uma rede.

Isto é principalmente o caso de aplicações Windows. Neste cenário, a maioria das camadas pode ser executado em computador do cliente, exceto a camada de persistência que é executado no servidor. Dessa forma, o acesso banco de dados pode ser restrito para que somente o servidor sabe como acessá-lo eo cliente comunica com o servidor usando, por exemplo, uma camada de marshal-by-reference objetos.

O modelo de domínio devem ser serializáveis. Como proxies NHibernate não são serializáveis, também é obrigados a devolver as entidades com as associações totalmente inicializado. Em alguns casos extremos, você pode considerar o uso de objetos de transferência de dados. Para mais detalhes, leia o capítulo 11, seção 11.3.1.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Uma vez que o pedido é feito, você pode se perguntar se você realmente atingido seus objetivos iniciais. E após o desenvolvimento é feito, a manutenção começa (corrigindo bugs, melhorando o desempenho, etc.) Vamos ver como podemos ajudar com estas tarefas chatas e estressante.

Alcançar as metas 8.3 e resolução de problemas

Agora que já olhou para o processo de desenvolvimento de uma aplicação NHibernate, vamos dar um passo para trás e pensar novamente sobre a concepção desta aplicação. Mesmo com uma boa implementação e um conjunto de testes, um aplicativo mal projetado seria inútil. É por isso que você deve ter alguns objetivos de design e formas de medir o quanto eles são alcançados.

Uma boa compreensão do processo de desenvolvimento é necessário para tirar o máximo partido deste seção. No entanto, não cometa o erro de adiar esta tarefa quando na verdade desenvolvimento de um aplicação. Você deve perceber que os custos para alterar o design de uma aplicação como aumentar rapidamente o aplicativo é implementado. E não hesite, no meio do desenvolvimento de fazer uma pausa e olhar para trás em seus objetivos iniciais e quão longe você está com eles.

Avaliação de um status da aplicação nem sempre é fácil. Os desenvolvedores também podem ter um tempo duro entender o que está errado com a sua aplicação. Ter algumas habilidades na resolução de problemas é definitivamente mais um quando se trata de estruturas complexas como NHibernate.

Finalmente, você deve lembrar que uma única ferramenta não pode fazer todos os trabalhos. Aprenda a escolher sabiamente

NHibernate quando é a melhor opção e retorno para outras alternativas para outras situações.

Esperemos que, até o final desta seção, você terá uma melhor compreensão de como lidar com essas tarefas. Vamos começar com o primeiro no processo de desenvolvimento: Atingir os objetivos do projeto.

8.3.1 Metas de design aplicado a um aplicativo NHibernate

A aplicação NHibernate é simplesmente uma aplicação .NET utilizando NHibernate. No entanto, por causa da papel central desempenhado pelo NHibernate, existem algumas implicações que devem ser tomados em consideração ao projetar um aplicativo NHibernate.

Como explicado na seção seguinte, nenhuma ferramenta se adapta a cada trabalho. NHibernate é certamente um bom quadro para resolver a incompatibilidade ORM, mas também é um problema complexo. Isso requer algumas habilidades a serem correctamente utilizados. Aconselhamo-lo a testar cuidadosamente NHibernate e sua competência, antes de começar a utilizar

lo em um ambiente de missão crítica.

Há seis objetivos do projeto definido pelo MSDN e vamos olhar para cada um deles com

NHibernate em mente. Este é a capacidade de um aplicativo para estar presente e pronto para uso. Basicamente, o aplicativo deve objetivo é ser livre de bugs.

NHibernate tem um impacto sobre a forma como a sua aplicação é testada. Porque não é intrusiva, o lógica de negócios em seu modelo de domínio e camada de negócios pode ser totalmente testado fora do NHibernate escopo. Note que você ainda deve testar a maneira como você usa NHibernate, ver o desempenho e escalabilidade seção.

Extensos testes é a primeira recomendação para atingir esse objetivo; testar a internas do seu aplicação e suas interações com o mundo exterior. Se o seu aplicativo interage com externa serviços, verifique se uma falha em um desses serviços não fará com que seu aplicativo falhar.

Finalmente, se sua aplicação fornece serviços para outros sistemas, verifique se eles não podem travar o seu aplicação através do envio de informações inválidas ou usar de seus serviços de uma maneira específica. O usuário final pode ser considerado como parte desses sistemas. O primeiro passo é validar as entradas que você recebe a partir desses sistemas.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Gerenciamento e Securability

Esta é a capacidade de um aplicativo a ser administrado e para proteger os seus recursos e seus usuários. O finalidade de gerenciamento é para facilitar a (re) configuração e manutenção da aplicação.

Outro benefício do NHibernate é que ela incentiva a separação de interesses na sua aplicação (Arquitetura em camadas), portanto, aumentar a sua capacidade de gerenciamento. NHibernate também é configurável usando XML

arquivos, permitindo assim a produção em tempo de mudanças.

A segurança não era uma grande preocupação de alguns anos atrás, mas ele está ganhando mais atenção. O primeiro conselho

relacionadas com a segurança é manter a sequência de conexão em um lugar seguro (não de texto simples e não em um montagem), por exemplo, você pode mantê-lo codificado em Web.config.

O conselho geral é a de implementar a sua aplicação com segurança em mente e para incentivar o uso da quantidade mínima de privilégios.

Desempenho e escalabilidade

Desempenho é a medida de operação de um aplicativo sob carga. Os desenvolvedores tendem a considerar esta objetivo como o mais importante. Também é o objetivo mais incompreendido (por exemplo, é comumente confundido com escalabilidade).

NHibernate é uma camada em cima de ADO.NET; fazer alguns testes para se certificar que seu aplicativo executa

assim, identificar gargalos e certifique-se que NHibernate é eficientemente utilizado (a busca preguiçosa / ansiosos,

caching, etc.) Como último recurso, NHibernate permite fallback para o clássico ADO.NET (subjacente conexão é acessível através da propriedade: ISession.Connection); Não somos contra armazenados procedimentos para o processamento batch. Note-se que NHibernate 1.2.0 é agora mais rápido do que a mão-coded acesso a dados

para operações clássico sobre o SQL Server porque ele está usando um recurso de lotes oposto da NET. quadro.

A próxima seção dá algumas dicas para melhorar o desempenho de sua aplicação. Lembre-se que otimização de desempenho não é algo para fazer no final da execução; pensar nisso a partir o início (mas não muito).

Escalabilidade é a capacidade de um aplicativo para atender a demanda crescente com o aumento dos recursos.

Há muitas técnicas para atingir esse objetivo. Você pode usar programação assíncrona (embalagem chamadas caras usando delegados assíncrono). Você também deve usar o pool de conexão ADO.NET (E talvez aumentar o seu tamanho), neste caso, evite usar uma sequência de conexão por usuário.

Outra boa prática é abrir uma sessão NHibernate o mais tarde possível e fechá-lo tão logo possível. Você também pode considerar o uso de um cache distribuído (leia o capítulo 6, seção 6.3, "Caching teoria e prática").

Nota final

Você pode certamente propor a cumprir todos estes objetivos do projeto, no entanto, uma escolha de design que tem um

efeito positivo sobre uma meta pode, ao mesmo tempo ter um efeito negativo sobre a outra.

Lembre-se que é certamente importante ter regras rígidas para se certificar de que essas metas são mantidos em

mente através do processo de desenvolvimento inteiro, mas também é importante para o bem equilibrar a quantidade de

esforço colocado em uma otimização com o resultado deste trabalho.

Não trabalhe em uma característica mais do que vale a pena e não otimizar cegamente, isto é, sem qualquer forma

8.3.2 Identificando os problemas e resolvendo

saber se essa otimização é realmente necessário e para medir a melhoria de otimização.

Não há nada mais frustrante do que recebendo um erro e não entender de onde vem.

No entanto, um processo de depuração rigorosa ajuda facilmente fixar a maioria dos erros. Erros em aplicativos. NET são geralmente jogado exceções.

Os usuários finais podem também queixar-se quando se utiliza uma aplicação livre de bugs. Os principais reclamar sendo que é

muito lento. Este problema pode ser muito frustrante para os usuários finais e você pode ter certeza que eles vão bug por isso.

Processo de resolução de bugs

Antes de pensar sobre a fixação de um bug, certifique-se que há uma infra-estrutura para a captura e registrá-la e certifique-se que o usuário final recebe uma mensagem útil. Isto é muito importante na produção ambiente.

O primeiro passo para corrigir um bug revelado através de uma exceção é para ler o conteúdo dessa exceção.
Não

apenas a mensagem, mas também o rastreamento de pilha, exceções internas, etc (tudo retornado por exception.ToString ()).

Então, você deve tentar entender o significado desta exceção (consulte a documentação) e começar a investigar a sua origem. Uma boa técnica, nesta fase, é isolar o problema até a sua origem (E solução) torna-se óbvia. Em termos práticos, isso significa remover processos até que o culpado um está localizado. Se você tem dificuldade de entender uma exceção NHibernate, leia no apêndice C do seção sobre pedir ajuda.

Depois de ter plenamente identificado o problema, TDD recomenda que você escrever alguns testes reproduzir este problema, estes testes irão ajudar a evitar que este problema vem mais tarde despercebido.

Você pode levar algum tempo repensar sobre o design do seu aplicativo para ver se este problema não é o sintoma de uma maior.

Finalmente, você pode corrigir o bug, fazendo os testes que você escreveu passar.

Melhorar o desempenho

Se você ler esta seção, você pode ter apenas receber um destes desempenho reclama falamos aproximadamente. Aqui estão alguns erros comuns que você pode ter feito e dicas para ajudar a melhorar drasticamente desempenho do seu aplicativo (e fazer os usuários finais felizes).

By the way, você deve considerar escrever testes de desempenho em um estágio inicial para evitar a espera para o usuário final para lhe dizer que sua aplicação é muito lenta. E não se esqueça de também otimizar seu banco de dados
(Somando os índices corretos, etc.)

Um erro que alguns novos desenvolvedores NHibernate cometem é que eles criam a fábrica de sessão mais do que o necessário. Este é um processo muito caro. Na maioria das vezes, é feito uma vez no início da a aplicação. Evite manter a fábrica de sessão em um lugar que vidas mais curtas do que a sua aplicação (Como mantê-lo em uma solicitação de página web).

Outro erro comum, relacionado ao fato de que NHibernate torna tão fácil para carregar as entidades, é que você pode carregar mais informações do que você precisa (mesmo sem conhecê-lo). Por exemplo, associações e coleções são totalmente inicializado quando o carregamento lento não está habilitado. Assim, mesmo quando o carregamento de uma única entidade, você pode acabar buscando um gráfico objeto inteiro. O conselho geral aqui é sempre permitir carregamento lento e cuidadosamente escrever suas consultas.

Outro problema relacionado, que surgem quando permitindo o carregamento lento, é a n +1 problema de seleção. Para mais mais detalhes, leia o capítulo 8, seção 8.6.1, "Resolução de n +1 seleciona problema". By the way, uma forma agradável ponto esta questão é cedo para medir o número de consultas executadas por página, você pode facilmente atingir que, escrevendo uma ferramenta para assistir os logs de NHibernate.SQL em DEBUG nível. Se for maior que um certo limite, você tem um problema a resolver e fazê-lo imediatamente, antes que esqueça o que está acontecendo em esta página. Você pode também outra medida de desempenho assassino de operações (como o número de chamadas remotas por página) e informações sobre o desempenho global (como o tempo que leva para processar cada página).

Você também deve tentar carregar as informações que você precisa usando o número mínimo de consultas (No entanto, evitar consultas caros, como os que envolvem produto cartesiano). Note que é geralmente mais importante para minimizar o número de entidades carregado (contagem de linha) do que o número de campos

Por favor, postei comentários para cada entidade (particularmente polimórficas) Autor

<http://www.mindsharp.com.br/descrivendo-muitas-caracteristicas> que podem ajudá-lo a escrever consultas otimizadas.

Agora, vamos falar sobre um tema menos conhecido. Este está relacionado com as obras NHibernate caminho. Quando você carregar entidades, a sessão NHibernate mantém uma série de informações sobre eles (por transparente

persistência, checagem suja, etc.) E quando cometer / rubor, a sessão usa essa informação para realizar as operações necessárias.

Há uma situação específica em que este processo pode ser um gargalo de desempenho: Quando você carrega um

Muitas entidades para atualizar apenas alguns deles, este processo será mais lento do que deveria ser. A razão é que a sessão irá verificar todas essas entidades para encontrar aqueles que devem ser atualizados. Você deve ajudá-la

evitar essa perda por expulsão de entidades inalterada ou usar uma outra sessão para salvar entidades mudou.

Como último recurso, considere o uso do cache de nível 2 (e do cache de consulta) para acertar o banco de dados menos

muitas vezes e reutilizar os resultados anteriores. Leia o capítulo 6, seção 6.3, "teoria e prática Caching", para mais

Exemplos usando

Detalhes sobre os prós e contras deste recurso.
É comum em outros ambientes (como C++) para escrever um código como:

```
if (algo der errado) return -1;
```

No entanto, Diretrizes NET recomendamos o uso de exceções. Eles são muito mais poderosos e mais difíceis de miss. É muito importante entender este conceito, porque NHibernate depende deles para fornecer relatórios de erros e sua aplicação deve fazer o mesmo.

Embora você já deve estar familiarizado com este conceito, faremos uma breve revê-lo e explicar como lidar com exceções NHibernate.

Seu primeiro passo deve ser o de criar suas próprias exceções para fornecer melhor informação (daí ser capaz de lidar melhor com eles). Uso. NET build-in exceções apenas quando significativo (por exemplo ArgumentNullException ao relatar um argumento null).

Nestes capítulos, utilizamos BusinessException quando uma regra de negócio está quebrado. Você pode adicionar mais exceções específicas se você precisar.

Outra boa prática é não deixar que as exceções de bibliotecas externas atingem a camada de apresentação (Ou qualquer fachada como WebService) intocados. Você deve envolvê-los em suas próprias exceções.

Aqui está um exemplo mostrando a aplicação de uma simples Save() método em uma classe da camada de persistência (esta classe poderia ser chamado UserDAO).

```
public void Salvar (Usuário) {
    try {
        session.SaveOrUpdate (usuário);
    }
    catch (HibernateException ex) {

        log.error ("Erro ao salvar um usuário.", ex);
        throw new PersistenceException ("Erro ao salvar um usuário.", ex);
    }
}
```

HibernateException é a exceção (base) lançada pelo NHibernate. Obviamente, a camada superior deve fechar a sessão se uma exceção é lançada. Note que você pode mover este tratamento de exceção para o negócios layer (para fornecer uma mensagem business-friendly para o usuário final).

Se você se deparar com um problema (performance) ao utilizar NHibernate e você achar que é difícil de resolver,

você deve pisar para trás e perguntar-se se era realmente a ferramenta certa para este processo.

8.3.3 Use a ferramenta certa para o trabalho certo

Como explicado no capítulo 1, mapeamento objeto / relacional (ORM) e NHibernate são certamente poderosa ferramentas. No entanto, nós tomamos muito cuidado para não fazer NHibernate parecer ser uma bala de prata. Não é uma

solução que vai fazer todos os problemas de seu banco de dados desaparecerem magicamente.

A criação de aplicativos de banco de dados é uma das tarefas mais desafiadoras em desenvolvimento de software.

NHibernate trabalho é reduzir a quantidade de código que você tem que escrever para os mais comuns de 90 por cento de casos de uso (CRUD comum e relatórios).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Os 5 por cento ao lado são mais difíceis; consultas tornam-se complexos, semântica da transação não são claras em primeiro, e os gargalos de performance estão ocultos. Você pode resolver estes problemas com NHibernate elegante e manter seu aplicativo portátil, mas você também vai precisar de alguma experiência para obtê-lo direito.

Curva de aprendizagem do NHibernate é alto no início. Em nossa experiência, um desenvolvedor precisa de pelo menos dois a quatro semanas para aprender o básico. Não saltar sobre NHibernate uma semana antes de seu prazo de projeto-it não irá salvar você. Esteja preparado para investir mais tempo do que você precisa para uma outra aplicação web quadro ou utilitário simples.

Finalmente, use SQL e ADO.NET para os 5 por cento de casos de uso não se pode implementar com NHibernate, como massa de manipulação de dados ou consultas de relatórios complexos com fornecedor específico SQL funções. Você deve perceber que há muitas tarefas que não são inherentemente orientada a objetos. ORM forçando pode facilmente prejudicar o desempenho de sua aplicação.

Mais uma vez: Use a ferramenta certa para o trabalho certo.

8.4 serviços de integração: Caso de log de auditoria
Há outros lugares onde você deve pensar cuidadosamente sobre a abordagem correta. Por exemplo, serviços adicionando pode ser feito de várias maneiras, cada um com seus pros e contras. Vamos estudar o ~~Aprendendo a usar o NHibernate~~ falado sobre o desenvolvimento de uma aplicação NHibernate sem levar em conta ~~exemplos de código~~ características que podem ser difíceis de encaixar na nossa arquitetura em camadas. Nesta seção, vamos falar sobre a integração desses serviços.

No contexto deste livro, um serviço é um subsistema claramente separados (conjunto de classes) que é integrado ao aplicativo principal para adicionar alguma funcionalidade. Note-se que falar de independente serviços (por exemplo, usando COM) é em grande parte fora do escopo deste livro.

Os serviços mais comuns são o log de auditoria e segurança. Também é comum ter algum negócio componentes implementados como serviços. Por exemplo, em uma plataforma de mensagens, um serviço pode analisar mensagens para detectar usuários mal-comportados ou linguagem forte filtro.

Nós os chamamos de serviços porque eles devem ser fracamente acoplados com a lógica de negócios (embora não é tão importante para aplicações simples). Uma propriedade importante dos serviços é que, devido à sua acoplamento fraco, que pode evoluir e ser configurado de forma independente, mas também é fácil de desativá-las sem afetar profundamente a aplicação principal. É por isso que eles são muitas vezes parte do não-funcionais requisitos.

O log de auditoria é o processo de gravação de alterações feitas nos dados (que ocorrem os eventos, em geral). Um log de auditoria é uma tabela do banco de dados que contém informações sobre as alterações feitas a outros dados, especificamente sobre o evento que resulta na mudança. Por exemplo, podemos registrar informações sobre a criação e eventos de atualização para o leilão *ItemS*. A informação que é gravado geralmente inclui o usuário, o data e hora do evento, que tipo de evento ocorreu, eo item que foi alterado.

NHibernate tem instalações especiais para implementar registros de auditoria (e outros aspectos similares que requerem um mecanismo de persistência de eventos). Nesta seção, vamos utilizar o *IInterceptor* interface para implementar registro de auditoria. Mas primeiro, vamos falar brevemente sobre a maneira dura (fazê-lo manualmente), para que você possa

compreender o nível de dificuldade deste problema. Então, descobrimos como *IInterceptor* torna muito mais fácil.

8.4.1 Fazer o dia mais difícil,

O difícil caminho (Ou o manual de vias) descreve uma abordagem que requer uma quantidade de esforço contínuo através do processo de desenvolvimento. No caso de log de auditoria, que significa chamar o log de auditoria serviço cada vez que é necessário.

Que não posso usar triggers de banco de dados?

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Logs de auditoria são muitas vezes manipulados usando triggers de banco de dados, e pensamos esta é uma excelente abordagem. No entanto, às vezes é melhor para o aplicação de assumir a responsabilidade, especialmente se um processamento complexo

é usado ou se a portabilidade entre diferentes bancos de dados é necessária.

Praticamente falando, você pode ter uma implementação semelhante ao seguinte:

```
public void Salvar (Usuário) {
    try {
        session.SaveOrUpdate (usuário);
        AuditLog.LogEvent (LogType.Update, user);
    }
    catch {...}
}
```

A chamada para o método LogEvent () da classe Auditlog irá gerar e salvar um log sobre este mudar. LogType é uma enumeração, é melhor do que usar uma corda. Note que você pode usar o Padrão Observer para remover a dependência sobre este serviço, leia o capítulo 10, seção 10.3.2 para mais detalhes sobre esse padrão.

A principal vantagem é que a melhor mensagem podem ser gerados para cada operação (devido ao fato de que sabemos exatamente o que estamos fazendo cada vez que ligar para este serviço).

As desvantagens desta abordagem é que ela é verbose (que tumultua o código), e mais importante: Ele pode ser esquecida ou ignorada, isto é inaceitável quando a construção de um fiel serviço de log de auditoria. Esta abordagem pode funcionar muito melhor para outros serviços, por isso não descartá-lo completamente.

8.4.2 Fazendo-o NHibernate-way

Vamos ver como NHibernate nos permite automatizar registro de auditoria. As vantagens desta abordagem são as desvantagens da maneira mais difícil, e vice-versa.

Você precisa executar vários passos para implementar essa abordagem:

- 1 Mark as classes persistentes para o qual deseja habilitar o log.
- 2 Definir as informações que devem ser registrados: usuário, data, hora, tipo de modificação, e assim por diante.
- 3 Unir tudo isso com um NHibernate IInterceptor que cria automaticamente a trilha de auditoria para você.

Criando o atributo marcador

Primeiro criamos um atributo de marcador, AuditableAttribute. Nós usamos esse atributo para marcar todos os persistentes

classes que devem ser automaticamente auditado:

```
[AttributeUsage (AttributeTargets.Class, AllowMultiple = false)]
[Serializable]
AuditableAttribute public class: Atributo {
```

Este atributo pode ser aplicado uma vez em classes, você pode adicionar algumas propriedades para personalizar a exploração madeireira por classe (por exemplo, um nome localizado para usar ao invés do nome da classe). Habilitando o log de auditoria para uma classe persistente particular é agora trivial; nós só adicioná-lo à declaração da classe. Aqui está um exemplo, para

```
[Auditable]
Item<class> Item {pública
    ...
}
```

Note que, usando um atributo implica confiar em entity.ToString () para obter registro detalhes porque não haverá outros meios para o log de auditoria de serviços para extraí-los, a menos que você use um grande interruptor

declaração para converter o objeto (viável se este serviço está ciente do modelo de domínio).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Em vez de um atributo, podemos criar um IAuditable interface. Dessa forma, as entidades podem ativamente participar no processo de log. De qualquer forma, você pode fazer as duas coisas e escolher o melhor um para cada entidade.

Criar e mapear o registro de log

Agora vamos criar uma nova classe persistente, AuditLogRecord. Esta classe representa as informações que deseja registrar na tabela do banco de dados auditoria:

```
public class {AuditLogRecord

    público Id prazo;
    público Mensagem string;
    público longo EntityID;
    público Tipo EntityType;
    público UserId prazo;
    público Criado DateTime;

    interna AuditLogRecord () {}

    pública AuditLogRecord (string mensagem,
                           EntityID prazo,
                           EntityType tipo,
                           UserId tempo) {
        this.Message = mensagem;
        this.EntityId = EntityID;
        this.EntityType = EntityType;
        this.UserId = userId;
        this.Created = DateTime.Now;
    }

}
```

Você não deve considerar esta parte da classe do seu modelo de domínio. Portanto, você não precisa ser tão cauteloso

sobre expor campos públicos. O AuditLogRecord é parte de sua camada de persistência e, possivelmente, partes do mesmo conjunto com outros persistência classes relacionadas, como o seu mapeamento de tipos personalizados.

Em seguida, mapeamos essa classe para a AUDIT_LOG tabela do banco:

```
<Nome da classe = "NHibernate.Auction.Persistence.Audit.AuditLogRecord,
  NHibernate.Auction.Persistence "
  table = "AUDIT_LOG"
  mutable = "false">

<id name="Id" column="AUDIT_LOG_ID">
  <generator class="nativé"/>
</ Id>

  Propriedade nome = "Mensagem" coluna = "MENSAGEM" />
  Propriedade nome = coluna "EntityID" = "ENTITY_ID" />
  Propriedade nome = "EntityType" coluna = "ENTITY CLASS" />
  Propriedade nome = "UserId" coluna = "USER ID" />
  Propriedade nome = "Criado" coluna = "criado" />
</ Class>
</ Hibernate-mapping>
```

Marcamos a classe mutable = "false", Uma vez AuditLogRecords são imutáveis, NHibernate agora não atualizar o registro, mesmo se você tentar.

A preocupação log de auditoria é um pouco ortogonal à lógica comercial que faz com que o log-able evento. É possível misturar lógica para o log de auditoria com a lógica de negócios, mas em muitas aplicações é log de auditoria preferível que ser tratado em uma peça central de código, de forma transparente para a lógica de negócios.

Nós não iríamos criar manualmente um novo AuditLogRecord e guardá-lo sempre que um Item é modificado. Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

NHibernate oferece um ponto de extensão, de modo que você pode plugar em uma rotina de log de auditoria (ou qualquer outro semelhante ouvinte de evento). Esta extensão é conhecida como uma NHibernate IInterceptor.

Escrever um interceptor

Nós preferimos que um LogEvent () método ser chamado automaticamente quando chamamos Save (). A melhor maneira de fazer isso com NHibernate é implementar o IInterceptor interface. Aqui está um exemplo:

Listagem implementação IInterceptor 8,2 para log de auditoria

```
AuditLogInterceptor public class: (NHibernate.Cfg.EmptyInterceptor

    sessão ISession privado;
    userId longo privado;

    inserir ISET private = HashedSet new ();
    atualizações privadas ISET HashSet = new ();

    Sessão pública ISession {
        get {this.session return;}
        set {this.session = valor;}
    }
    UserId longa pública {
        get {return this.userId;}
        set {this.userId = valor;}
    }

    public virtual bool OnSave (entidade objeto,
                               identificação do objeto,
                               objeto [estado],
                               string [] propertyNames,
                               ITipo tipos []) {

        if (entity.GetType (). GetCustomAttributes (
            typeof (AuditableAttribute), false) Comprimento.> 0)
            inserts.Add (entidade);

        retorno base.OnSave (entidade, id, estado, propertyNames, os tipos);
    }

    public virtual bool OnFlushDirty (entidade objeto,
                                    identificação do objeto,
                                    object [] currentState,
                                    object [] previousState,
                                    string [] propertyNames,
                                    ITipo tipos []) {

        if (entity.GetType (). GetCustomAttributes (
            typeof (AuditableAttribute), false) Comprimento.> 0)
            updates.Add (entidade);

        base.OnFlushDirty retorno (id, entidade,
                                  currentState, previousState, propertyNames, os tipos);
    }

    PostFlush public void virtual (System.Collections.ICollection c) {
        try {
            foreach (entidade objeto em inserções) {
                AuditLog.LogEvent (LogType.Create,
                                   entidade,
                                   userId,
                                   session.Connection);
            }
            foreach (entidade objeto em atualizações) {
                AuditLog.LogEvent (LogType.Update,
                                   entidade,
                                   userId,
                                   session.Connection);
            }
        }
    }
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        }
    } catch (HibernateException ex) {
        throw new CallbackException (ex);
    } finally {
        inserts.Clear ();
        updates.Clear ();
    }

}

}

# 1 Collections para manter as entidades novas e
# modificadas
# 2 Uma sessão aberta, devem ser fornecidos
# 3 Necessário para AuditLogRecord
# 4 Colete novas entidades
# 5 Colete entidades modificadas
# 6 processo de Log

```

Em vez de diretamente implementação do `IInterceptor` interface, que herdamos de `EmptyInterceptor` o que nos permite ignorar os métodos desta interface que não precisamos.

Este interceptor particular tem dois aspectos interessantes. Primeiro, o `sessão` e `userId` são campos deste interceptor precisa fazer o seu trabalho, assim que um cliente usando este interceptor terá que definir as propriedades quando permitindo que o interceptor. O outro aspecto interessante é a rotina de log de auditoria em `OnSave()` e `OnFlushDirty()`, Onde adicionamos novas entidades e atualizado para coleções. O `OnSave()` interceptor método é chamado sempre NHibernate salva uma entidade, a `OnFlushDirty()` método é chamado NHibernate sempre detecta um objeto sujo. O log de auditoria é feito no `PostFlush()` método, que NHibernate chamadas após a execução do SQL sincronização.

Note-se que `entity.GetType().GetCustomAttributes()` irá executar mal (em relação ao uso `IAuditable`), Embora, você pode otimizar esse código, cache de todos os tipos de decoração.

Nós usamos a chamada estática `AuditLog.LogEvent()` (Uma classe e método que discutiremos em seguida) para registrar o evento.

Note que não podemos registrar eventos no `OnSave()`, Porque o valor do identificador de uma nova entidade pode não ser conhecida neste momento. NHibernate tem a garantia de ter definido todos os identificadores entidade após a lavagem, de modo

`PostFlush()` é um bom lugar para executar o log de auditoria.

Observe também como usamos a sessão: Passamos a conexão ADO.NET de uma sessão dada ao chamada estática para `AuditLog.LogEvent()`. Há uma boa razão para fazer isso, como vamos discutir em mais detalhe. Vamos primeiro amarrá-lo todos juntos e ver como você ativar o interceptor novo.

Permitindo que o interceptor

Você precisa atribuir o `IInterceptor` a um NHibernate `ISession` quando você abre a sessão:

```

AuditLogInterceptor interceptor = new ();
usando (sessão ISession =
    sessionFactory.openSession (interceptor)) {
    interceptor.Session sessão =;
    interceptor.UserId = currentUser.Id;

    usando (session.BeginTransaction ()) {
        Session.save (newItem) // Triggers OnSave () do interceptor
        session.Transaction.Commit (); // Triggers PostFlush ()
    }
}

```

Você deve mover-se a sessão de abertura a um método auxiliar para evitar de fazer esse trabalho cada vez.

Vamos voltar a esse código de sessão de manipulação de interessante dentro do interceptor e descobrir por que nós passou a Conexão da corrente `ISession` para `AuditLog.LogEvent()`.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Usando uma sessão temporária

Deve ficar claro por que precisamos de uma `ISession` exemplo, dentro do `AuditLogInterceptor`. O interceptor tem que criar e persistir `AuditLogRecord` objetos, portanto, uma primeira tentativa para a `OnSave()` método poderia ter sido a seguinte rotina:

```
if (entity.GetType (). GetCustomAttributes (typeof (AuditableAttribute), false) Comprimento.> 0) {
    try {

        objeto EntityID = session.GetIdentifier (entidade);
        AuditLogRecord LogRecord = new AuditLogRecord (...);
        / / ... definir as informações de log

        Session.save (LogRecord);
    } Catch (HibernateException ex) {
        throw new CallbackException (ex);
    }
}
```

Veja como usamos `session.GetIdentifier (entidade)` facilmente obter o identificador. Esta implementação parece simples: criar um novo `AuditLogRecord` instância e salvá-lo, usando a corrente sessão.

No entanto, ele não funciona.

É ilegal para invocar o NHibernate originais `ISession` a partir de um `IInterceptor` callback. O sessão está em um estado frágil durante as chamadas de interceptor. Um truque legal que evita esse problema é abrir um novo `ISession` com o único propósito de salvar um único `AuditLogRecord` objeto. Para manter este tão rápido quanto possível, reutilizar a conexão ADO.NET a partir do original `ISession`.

Este sessão temporária manipulação é encapsulado no `Auditlog` classe auxiliar:

Listagem 8.3 padrão de sessão temporária

```
public class {auditlog

    LogEvent public static void (
        LogType LogType,
        entidade objeto,
        userId prazo,
        Conexão IDbConnection) {

        usando (tempSession ISession sessionFactory.openSession = (conexão)) { | 1

            AuditLogRecord record =
                nova AuditLogRecord (logType.ToString (), | 2
                    tempSession.GetIdentifier (entidade), | 2
                    entity.GetType (), | 2
                    userId); | 2

            tempSession.Save (registro); | 2
            tempSession.Flush (); | 2
        }
    }
}
```

1 Criar uma nova sessão reutilizar uma conexão aberta

2 Criar e guardar o registro de log

Note que este método nunca confirma ou começa quaisquer transações de banco de dados, tudo que ele faz é executar

adicional INSERIR declarações sobre uma conexão ADO.NET existente e dentro do banco de dados atual transação. Usando uma temporária `ISession` para algumas operações na conexão ADO.NET e mesmo transação é uma boa dica você também pode achar útil em outros cenários.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O NHibernate-way é poderoso, simples e mais fácil de integrar. No entanto, existem outros tipos de operações que não podem trabalhar usá-lo. No caso de o log de auditoria, o NHibernate-way só registra operações por entidade. Você não pode registrar uma operação que alterou a muitas entidades ou não relacionados à persistência.

A linha inferior é que você provavelmente usará as duas abordagens.

Nós encorajamos você a experimentar e tentar padrões interceptor diferentes. O site NHibernate também tem exemplos usando interceptores aninhadas ou registrar uma história completa (incluindo a propriedade atualizada e de coleta de informações) para uma entidade.

8.4.3 Outras formas de integração de serviços

As abordagens que abrangeu comuns e muito simples de implementar. No entanto, mais complexo aplicações podem exigir uma abordagem mais flexível. Nesta seção, descobrimos padrões chamados Inversão de Controle e Injeção de dependência. Nós também lhe dar uma dica de como você pode usar um registro biblioteca para fundir seus registros com os registros NHibernate.

Inversão de Controle e Injeção de Dependência

Está fora do escopo deste livro para cobrir esses padrões, no entanto, se você não conhece, uma breve introdução certamente seria útil.

A motivação desses padrões é evitar a alta de acoplamento entre os serviços de abordar diferentes preocupações.

Vamos dar um exemplo: No nosso aplicativo de leilão, que termina um leilão pode exigir que nós atualizamos o banco de dados, enviar uma notificação ao vencedor, recolher o pagamento e envio do item. Estes passos exigem que se comunicar com diferentes serviços. A alta de acoplamento com eles pode prejudicar o gerenciamento e flexibilidade da nossa aplicação. Configurar e alterar esses serviços podem tornar-se muito difícil.

A solução trazida pela inversão de controle é a exteriorizar a ligação entre a aplicação e os serviços. Em termos práticos, isso significa que você definir interfaces (contratos) para se comunicar com os serviços e você usar um arquivo de configuração (geralmente escrito em XML) para especificar o serviço para uso para cada interface. Dessa forma, mudando o serviço pode ser feito pela simples edição da configuração arquivo.

No caso de log de auditoria, você pode criar uma interface chamada IAuditLog e especificar no arquivo de configuração que a classe (de serviços) a utilizar é o Auditlog classe definida na Listagem 9.3.

Existem muitas bibliotecas que fornecem esses recursos, cada um com seus pros e contras. Os dois da mais populares são o Castelo de Windsor (<http://www.castleproject.org/container/>) e Spring.NET ([Http://www.springframework.net/](http://www.springframework.net/)).

Para mais detalhes, leia http://en.wikipedia.org/wiki/Dependency_injection.

Integrar NHibernate registro

Você pode decidir usar uma biblioteca de registro em vez de (ou além) a gravação de logs usando NHibernate. Este tipo de registro, geralmente não é para a auditoria, mas para a manutenção.

É fácil de fundir seus registros com os registros NHibernate usando log4net. Esta biblioteca fornece vários destinos para os logs, é possível até mesmo enviá-los através de e-mails ou para salvar a em um banco de dados.

No entanto, cuidado com os custos de desempenho.

Se você não pode usá-lo, você ainda tem outra opção: biblioteca log4net Wrap (para redirecionar chamadas de método) para ser capaz de mudar de log4net a outras soluções, como o Enterprise Library ou o System.Diagnostics API.

Embora o registro é um requisito não-funcional (que os cuidados finais sobre algumas), eles são inestimável para depurar aplicativos em produção. Então, pense duas vezes antes de decidir que você não precisa dele.

8.5 Resumo

Este capítulo focado em desenvolvimento de aplicações e as questões de integração que podem ocorrer quando escrever aplicações NHibernate. Primeiro falou a aplicação prática de uma aplicação em camadas. Nós discutimos como o modelo de domínio ea camada de negócio deve ser implementado e testado. Em seguida, falou sobre a camada de persistência e terminou com a camada de apresentação.

O próximo objetivo deste capítulo foi para ajudar a integração de aplicações NHibernate em produção ambientes. Nós falamos especificamente sobre a questão confiança média, você pode encontrar ao desenvolvimento de aplicações web.

Depois disso, resumimos como NHibernate pode ajudar a alcançar os objetivos do projeto padrão de um NET. aplicação. NHibernate tem um impacto sobre a maneira de projetar uma aplicação e uma utilização cuidadosa da sua recursos pode melhorar significativamente a qualidade de sua aplicação. Nós também deu algumas dicas ajudando identificar e resolução de bugs e problemas de desempenho.

Na última seção do capítulo, falamos sobre a integração dos serviços em uma aplicação NHibernate. Discutimos os prós e contras da maneira mais difícil eo NHibernate-way. Falamos também de mais alguns alternativas.

Implementamos o log de auditoria de entidades persistentes com uma implementação do NHibernate IInterceptor interface. Nossa interceptor personalizado usa uma temporária ISession truque para rastrear eventos de modificação em uma tabela de histórico de auditoria.

Agora você está pronto para cavar os detalhes da implementação das duas camadas diretamente relacionado com NHibernate: A camada de modelo de domínio ea camada de persistência. Estes são os tópicos abordados no próximos dois capítulos. Ir para o próximo capítulo para começar com a camada de modelo de domínio.

9

Escrita Modelos de Domínio Real World

Este capítulo aborda

- „ Processos de domínio modelo de desenvolvimento e legado
- „ mapeamento de esquemas
- „ Ignorância compreensão persistência e negócios
- „ lógica
- „ Implementação de ligação de dados GUI
- „ Abordagens para obter um DataSet de entidades
- „

Este capítulo se concentra em alguns aspectos avançados do desenvolvimento de um modelo de domínio.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Desde os primeiros capítulos, você tem sido familiarizados com o que uma entidade se parece. No entanto, nós foram manualmente implementação de entidades simples que contêm dados e aprender como mapear estes dados para que NHibernate pode carregar e salvá-lo.

Na primeira seção deste capítulo, você aprenderá muitas maneiras automatizado de implementação do modelo de domínio, criando o banco de dados, e escrever o mapeamento. No caso que você está trabalhando contra um banco de dados existente, você pode ter para mapear algumas tabelas exóticos. A próxima seção apresenta vários recursos do NHibernate que ajudam a apoiar essas mapeamento.

No mundo real, um modelo de domínio contém muito mais do que dados. Ele contém comportamento. A fim de evitar dependências desnecessárias, este capítulo também irá ensinar como implementar as entidades desconhecem a camada de persistência. Depois disso, vamos explicar o que a lógica de negócios de um modelo de domínio é e como implementá-lo.

Neste ponto, você terá um modelo de domínio totalmente funcional. Será tempo para interagir com o outras camadas. As duas últimas seções explicam como vincular dados de uma entidade para a camada de apresentação e como para preencher uma DataSet com o conteúdo de uma entidade, a fim de comunicar-se com componentes que requerem

DataSets.

Os processos de desenvolvimento 9.1 Descobrir formas eficientes de implementar o modelo de domínio.

ferramentas

Nos primeiros capítulos, que sempre começou por definir o modelo de domínio antes de criar o banco de dados e escrever o mapeamento. No entanto, no mundo real, nem sempre é assim. Nesta seção, vamos apresentar os processos que podem ser usados para desenvolver o modelo de domínio, o banco de dados e o mapeamento em qualquer ordem. Você vai aprender como ferramentas podem ajudá-lo a ir de para os outros.

Depois de ter uma dessas representações (o modelo de domínio, o banco de dados ou o mapeamento), NHibernate fornece ferramentas que podem ser usados para (parcialmente) gerar a outras representações. Geralmente, você terá que completar e personalizar o código gerado.

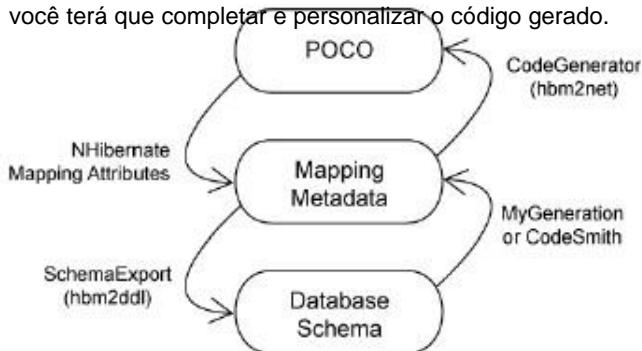


Figura 9.1 Os processos de desenvolvimento

Vamos rever esses vários processos nesta seção.

9.1.1 Gerando o mapeamento e banco de dados de entidades

Temos vindo a utilizar essa abordagem neste livro. É comumente chamado top-down de desenvolvimento. Na ausência de um modelo de dados existente, este é o desenvolvimento mais confortável estilo para a maioria dos desenvolvedores .NET.

Ao usar esta abordagem, começamos com um modelo de domínio existentes .NET (o ideal é implementado com POCOs) e completa liberdade com relação ao esquema de banco de dados.

Então, usamos uma biblioteca como NHibernate.Mapping.Attributes para gerar o mapeamento (ou podemos manualmente escrevê-lo usando um editor de XML). O Apêndice B contém algumas dicas úteis quando se utiliza NHibernate.Mapping.Attributes. Você também vai descobrir outra biblioteca atributos no Apêndice C.

Finalmente, deixamos NHibernate hbm2ddl ferramenta de gerar o esquema do banco de dados usando o mapeamento metadados. Esta ferramenta é parte da biblioteca NHibernate. Não é uma ferramenta gráfica. Sua interface é a classe

NHibernate.Tool.hbm2ddl.SchemaExport, Pelo que é chamado também às vezes SchemaExport.

Ao planejar a usar esta ferramenta, alguns elementos especiais e atributos podem ser usados no mapeamento arquivos. A maioria deles são relevantes apenas para um esquema personalizado. NHibernate tenta usar sensata padrão se você não especificar seus próprios nomes e estratégias (por exemplo, os nomes das colunas); no entanto, ser avisado de que um DBA profissional não pode aceitar este esquema padrão sem manual alterações. No entanto, os padrões podem ser satisfatórias para um ambiente de desenvolvimento ou protótipo.

Note-se que você também pode usar uma estratégia de nomeação (como explicado no capítulo 3, seção 3.5.7) para alterar

entidades os nomes maneira são convertidos em nomes de tabelas.

Preparar os metadados de mapeamento

Neste exemplo, temos marcado o mapeamento para a Item classe com hbm2ddlAtributos específicos e elementos. Estas definições opcionais integrar perfeitamente com os elementos de mapeamento, como você pode ver na listagem 9.1.

Listing 9.1 elementos adicionais na Item mapeamento para SchemaExport

```
<class name="Item" table="ITEM">

<id name="Id" type="string">
    <column name="ITEM_ID" sql-type="char(32)" />
    <generator class="uuid.hex"/>
</ Id>                                | 1

<property name="nome" type="string">
    <Nome da coluna = "NOME"
        not-null = "true"
        comprimento = "255"
        index = "IDX_ITEMNAME" />
</ Property>                            | 2
                                         | 2
                                         | 2

<Nome da propriedade = "Description"
    type = "String"
    coluna = "DESCRIÇÃO"
    comprimento = "4000" />                | 3

<Nome da propriedade = "InitialPrice"
    type = "MonetaryAmount">
    <column name="INITIAL_PRICE" check="INITIAL_PRICE > 0" />
    <column name="INITIAL_PRICE_CURRENCY"/>
</ Property>                            | 4

<set name="Categories" table="CATEGORY_ITEM" cascade="none">
    <key>
        <column name="ITEM_ID" sql-type="char(32)" />
    </ Key>
    <many-to-many class="Category">
        <column name="CATEGORY_ID" sql-type="char(32)" />
    </ Many-to-many>
</ Set>                                  | 5

...
</ Class>
```

hbm2ddl gera automaticamente uma NVARCHAR coluna digitado se uma propriedade (mesmo de propriedade do identificador)

é do tipo de mapeamento Corda. Sabemos que o gerador de identificador uuid.hex sempre gera strings que são 32 caracteres, por isso, usamos um CHAR SQL tipo e definir seu tamanho fixado em 32 caracteres # 1. O

Por favor, postar comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

anhadas `<column>` elemento é necessário para esta declaração, porque não há nenhum atributo para especificar o SQL datatype na `<id>` elemento.

O `coluna`, `não-nulo` e comprimento atributos também estão disponíveis no `<property>` elemento, mas nós desejamos criar um índice adicional no banco de dados, daí voltamos a usar uma nested `<column>` elemento # 2. Este índice vai acelerar nossas buscas por itens pelo nome. Se reutilizar o mesmo nome de índice em outros o mapeamento da propriedade, podemos criar um índice que inclui colunas de banco de dados múltiplos. O valor deste atributo também é usado para nomear o índice no catálogo do banco de dados.

Para o campo de descrição, escolhemos a abordagem preguiçosa, usando os atributos na `<property>` elemento em vez de um `<column>` elemento. O DESCRIÇÃO coluna será gerado como VARCHAR (4000) # 3.

O tipo personalizado definido pelo usuário MonetaryAmount requer duas colunas do banco de dados para trabalhar. Temos

usar o `<column>` elemento. O verificar atributo # 4 desencadeia a criação de um verificação da restrição; o valor dessa coluna deve coincidir com a dada expressão SQL arbitrários. Note que há também uma verificar atributo para o `<class>` elemento, que é útil para várias colunas restrições de verificação.

A `<column>` elemento também pode ser usado para declarar os campos de chave estrangeira em um mapeamento da associação.

Caso contrário, as colunas de nossa tabela de associação CATEGORY_ITEM seria NVARCHAR (32) em vez de o mais apropriado CHAR (32) Tipo de # 5.

Nós agrupados todos os atributos relevantes para a geração de esquema na tabela 19,1; alguns deles não foram incluído no anterior Item mapeamento exemplo.

Tabela 9.1 XML atributos de mapeamento para hbm2ddl

Atributo	Valor	Descrição
Coluna	corda	Pode ser usado em elementos mais mapeamento; declara o nome do SQL coluna. <code>hbm2ddl</code> (E núcleo do NHibernate) padrão para o nome do .NET propriedade) se o <code>coluna</code> atributo é omitido e nenhum aninhadas <code><column></code> elemento está presente. Este comportamento pode ser alterado por implementação de um costume <code>INamingStrategy</code> ; veja a seção 3.5.7, "Naming convenções" no capítulo 3.
não-nulo	true / false	Força a geração de um NOT NULL restrição de coluna. Disponível em um atributo em elementos mais mapeamento e também no dedicada <code><column></code> elemento.
Único	true / false	Força a geração de uma única coluna- UNIQUE restrição. Disponíveis para vários elementos de mapeamento.
Comprimento	número inteiro	Pode ser usado para definir um "tamanho" de um tipo de dados. Por exemplo, <code>comprimento = "4000"</code> para um <code>corda</code> propriedade mapeada gera um NVARCHAR (4000) coluna. Este atributo também é usado para definir o precisão dos tipos decimal.
índice	corda	Define o nome de um índice de banco de dados que podem ser compartilhados por múltiplos elementos. Um índice em uma única coluna também é possível. Apenas disponível com o elemento <code><column></code> .
unique-chave	corda	Permite restrições únicas envolvendo colunas de banco de dados múltiplos. Todos elementos usando este atributo deve compartilhar o mesmo nome da restrição a ser parte de uma definição única restrição. Este é um <code><column></code> elemento-somente o atributo.

sql-type	corda	Substitui hbm2ddl de detecção automática do tipo de dados SQL; útil para os tipos de banco de dados específico de dados. Estar ciente de que isso efetivamente impede independência de banco de dados: hbm2ddl irá gerar automaticamente um VARCHAR ou VARCHAR2 (Para Oracle), mas vai sempre usar uma declarada SQL-tipo, em vez, se houver. Este atributo só pode ser usado com o dedicado <column> elemento.
de chave estrangeira	corda	Nomes de uma restrição de chave estrangeira, disponível para <many-to-one>, <one-to-one>, <key>, e <many-to-many> cartografia elementos. Note-se que inverse = "true" lados de uma associação mapeamento não será considerado para a nomeação de chave estrangeira, somente os não-lado inverso. Se nenhum nome for fornecido, NHibernate gera única nomes aleatórios.

Depois de ter revisto (provavelmente em conjunto com um DBA) seus arquivos de mapeamento e acrescentou esquema relacionados atributos, você pode criar o esquema.

Criando o esquema

O hbm2ddl ferramenta é instrumentado usando uma instância da classe SchemaExport. Por exemplo:

```
Cfg configuração Configuração = new ();
cfg.Configure ();
SchemaExport SchemaExport = new SchemaExport (cfg);
schemaExport.Create (false, true);
```

Este exemplo cria e inicializa uma configuração NHibernate. Então, ele cria uma instância de SchemaExport que usa o mapeamento e propriedades de conexão do banco de dados da configuração para gerar e executar os comandos SQL criar as tabelas do banco de dados.

Aqui é a interface pública da classe com uma breve descrição de cada método:

```
SchemaExport public class
{
    pública SchemaExport (cfg Configuração);

    // Especifique as propriedades de conexão do banco de dados separadamente
    pública SchemaExport (cfg Configuração, ConnectionProperties IDictionary);

    // O script gerado será gravado este arquivo
    pública SchemaExport SetOutputFile (string filename);

    // Definir o fim da declaração de SQL delimitador (normalmente ponto e vírgula)
    pública SchemaExport SetDelimiter (string delimiter);

    // Execute o script de esquema de criação
    public void Criar (script bool, exportação bool);

    // Execute o script de esquema queda
    Gota public void (script bool, exportação bool);

    // Para executar tanto a queda e criar scripts DDL.
    public void Execute (script bool, exportação bool, justDrop bool, formato bool);
    public void Execute (script bool, exportação bool, justDrop bool, formato bool,
                        Conexão IDbConnection, exportOutput TextWriter);
}
```

Tabela 9.2 explica o significado dos parâmetros do Execute () métodos.

Hbm2ddl.SchemaExport.Execute Tabela 9.2 () descrição parâmetros

Opção	Descrição

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Script	A saída do script gerado para o console.
Exportação	Execute o script gerado no banco de dados.
justDrop	Apenas uma gota as mesas e limpar o banco de dados.
Formato	Formatar o script gerado muito bem em vez de usar uma linha para cada declaração.
Conexão	Abriu conexão com o banco de usar quando a exportação é verdade.
exportOutput	A saída do script gerado para este escritor.

Esta ferramenta é indispensável ao aplicar TDD (explicado no capítulo 8, seção 8.1.1) porque liberta-o manualmente modificar o banco de dados sempre que as alterações de mapeamento. Tudo que você tem a fazer é chamá-lo antes de executar seus testes, e você obterá uma nova e up-to-banco de dados de data para trabalhar. Note-se que também está disponível como uma tarefa NAnt: NHibernate.Tasks.Hbm2DdlTask. Para mais detalhes, leia a documentação da API.

No entanto, o fato de que esta ferramenta totalmente regenera o banco de dados cada vez que significa que você não pode usá-lo se você quiser migrar dados de uma versão antiga do banco de dados. Felizmente, a seção 9.1.6 fornece algumas abordagens para lidar com esta questão.

Criação de objetos de banco de dados mais

Se você precisar executar declarações SQL arbitrárias ao gerar seu banco de dados, você pode adicioná-los ao seu documento de mapeamento. Isto é especialmente útil para criar triggers e armazenados procedimentos utilizados no mapeamento.

Estas declarações são escritas em `<database-object>` elementos. Se eles estão no `<create>` sub-elemento, são executadas ao criar o banco de dados. Outra coisa, eles estão no `<drop>` sub-elemento, e eles são executados quando deixar cair o banco de dados.

Aqui está um exemplo:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  <database-object>
    <create>
      CRIAR FindItems_SP procedimento como
      SELECIONE item_Id, NAME, INITIAL_PRICE, INITIAL_PRICE_CURRENCY, ...
      ITEM DE
    </ Criar>
    <drop>
      FindItems_SP DROP PROCEDURE
    </drop>
    <Gota />
  </ Banco de dados objeto->
  <dialect-scope name="NHibernate.Dialect.MsSql2005Dialect"/>
  <dialect-scope name="NHibernate.Dialect.MsSql2000Dialect"/>
</ Hibernate-mapping>
```

Este exemplo fornece o código necessário criar e soltar o procedimento armazenado usado na final da seção 7.6.2 no capítulo 7.

Como essas instruções SQL podem ser dialeto-dependente, também é possível usar `<dialect-scope>` para especificar para qual dialeto que eles devem ser executados. No exemplo anterior, o código só será executados em bases de dados SQL Server.

9.1.2 Geração de entidades a partir do mapeamento

O documento de mapeamento fornece informação suficiente para deduzir a completamente DDL esquema e gerar POCOs de trabalho. Além disso, o documento de mapeamento não é muito

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

verbose. Então, você pode preferir middle-out, desenvolvimento onde você começa com um manuscrito NHibernate documento de mapeamento e gerar o DDL usando hbm2ddl eo modelo de domínio usar geração de código. Esta abordagem é atraente, principalmente durante a migração dos actuais Hibernate de mapeamento (usada por um aplicativo Java).

A ferramenta usada para gerar entidades do mapeamento é chamado hbm2net. É bastante semelhante ao hbm2ddl, No entanto, está disponível como uma biblioteca separada, juntamente com um aplicativo de console (NHibernate.Tool.hbm2net.Console) E uma tarefa NAnt (NHibernate.Tasks.Hbm2NetTask).

Antes de usar esta ferramenta, você deve se certificar de que o mapeamento fornece todas as informações necessárias como tipos de propriedades ". Depois, você pode executá-la usando a sua CodeGenerator classe:

```
string [] args = new string [] {
    "- Config = hbm2net.config", "- output = DomainModel", "* . hbm.xml"};
NHibernate.Tool.hbm2net.CodeGenerator.Main (args);
```

Aqui é o equivalente usando seu aplicativo de console:

```
NHibernate.Tool.hbm2net.Console.exe
- Config = hbm2net.config - output = DomainModel *. hbm.xml
```

Este código irá gerar uma classe C # para cada documento de mapeamento no diretório atual e salve-o no DomainModel diretório. O conteúdo do arquivo hbm2net.config Parece que o seguinte anúncio:

```
<? Xml version = "1.0"?>
<codegen>

    attribute="implements"> <meta
        NHibernateInAction.CaveatEmptor.Persistence.Audit.IAuditable
    </ Meta>

    <Gerar
        renderer = "NHibernate.Tool.hbm2net.BasicRenderer" />

    <Gerar
        renderer = "NHibernate.Tool.hbm2net.FinderRenderer"
        sufixo = "Finder" />

</ Codegen>
```

Gerada C # classes irão herdar o especificado IAuditable interface. Processadores são usado para gerar partes específicas da classe C #, não há sequer um VelocityRenderer, Com base em a biblioteca NVelocity, que permite a utilização de um modelo. Consulte a sua documentação da API para mais detalhes.

Note que esta ferramenta não é tão completa quanto hbm2java Hibernate; consulte a documentação do último para mais detalhes.

Mais. NET desenvolvedores se sentir mais confortável usando o top-down de desenvolvimento com um atributo biblioteca como NHibernate.Mapping.Attributes que dá um controle máximo, ou utilizando o bottom-up desenvolvimento quando há um modelo de dados existente.

9.1.3 Gerando o mapeamento e as entidades do banco de dados

O bottom-up de desenvolvimento começa com um esquema de banco de dados existente e modelo de dados. Neste caso, o maneira mais fácil de proceder é usar uma ferramenta de geração de código como MyGeneration ActiveWriter ou CodeSmith para gerar documentos de mapeamento NHibernate e classes POCO esquelético persistente (dados de contêineres com campos e implementação simples de propriedades, mas nenhuma lógica). Você geralmente tem que melhorar e modificar o mapeamento NHibernate gerada pela mão, porque nem todos os detalhes associação de classe e

Por favor, nos comentários ou correções para o fórum online em <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Este tipo de geração de código é geralmente baseado em modelo: Você escreve um modelo descrever o mapeamento e POCO com espaços reservados (para os nomes, tipos, etc.) O código gerador executa este modelo para cada tabela no banco de dados. Este processo é bastante intuitiva. É mesmo possível para preservar escrito à mão regiões de código que não é o excesso de escrito quando regenerar as classes.

MyGeneration é livre gerador de código NET, você pode aprender mais sobre ele em seu site.: <http://www.mygenerationsoftware.com/>. Ele vem com um modelo NHibernate simples que você pode personalizar à vontade.

CodeSmith é uma ferramenta semelhante, que está disponível em edições gratuitas e comerciais. Para mais detalhes, acesse <http://www.codesmithtools.com/>.

Há também um trabalho em andamento Visual Studio add-in chamado ActiveWriter (<Http://www.altinoren.com/activewriter/>). Tem a vantagem de ser visualmente atraente e fornece entidades com ActiveRecord atributos (introduzido no Apêndice C).

É uma coisa ruim para escrever modelos de domínio anêmico?

Como explicado na seção 10.4, um modelo de domínio é feita de dados e comportamento.

Ao usar um gerador de código simples, você pode ser tentado a escrever o seu modelo de domínio como um contêiner de dados e mover todo o comportamento em outras camadas. Em

Neste caso, seu modelo de domínio é dito ser anêmica. Para mais detalhes, leia http://en.wikipedia.org/wiki/Anemic_Domain_Model.

Isso não é necessariamente uma coisa ruim. Esta abordagem pode funcionar bem para simples aplicações. No entanto, você deve perceber que vai contra a idéia básica de projeto orientado a objetos. O comportamento não pode ser corretamente representado

em outros camadas, levando à duplicação de código e outras questões. Pior, pode acabar no

9.1.4 Encontre-se em meio a camadas errado (como a camada de apresentação).

O cenário mais difícil combina existente. NET e um esquema relacional existente. É muito difícil para mapear modelos de domínio arbitrário para um determinado esquema.

Este cenário geralmente requer, pelo menos, refactoring algumas das classes. NET, esquema de banco de dados, ou ambos. O documento de mapeamento deve quase de certeza ser escrito à mão (embora seja possível usar NHibernate.Mapping.Attributes). Este é um cenário incrivelmente dolorosa que é, felizmente, extremamente raros.

Ao tentar usar esse cenário, não hesite em tirar o máximo proveito da extensão de numerosas interfaces de NHibernate. Eles foram introduzidos no capítulo 3, seção 3.1.4.

9.1.5 manutenção do esquema automático de banco de dados

Uma vez que você implantou uma aplicação, torna-se difícil alterar o esquema de banco de dados. Isso pode até ser o caso em desenvolvimento, se o cenário requer dados de teste que tem que ser reimplantado após cada mudança de esquema. Com hbm2ddl, Sua única opção é abandonar a estrutura existente e criá-lo novamente, possivelmente seguido por um time-consuming importar dados de teste.

Hibernate vem com uma ferramenta para a evolução do esquema, SchemaUpdate, Que é usado para atualizar um esquema de banco de dados existente SQL; cai obsoletos tabelas, colunas e restrições. No entanto, no momento da escrita, esta ferramenta não está disponível para NHibernate. Portanto, precisamos de uma outra maneira de resolver este problema.

A solução mais básica é a de salvar manualmente cada comando ALTER usado para atualizar progressivamente banco de dados. No entanto, pode ser difícil manter o controle de todas essas mudanças quando mantendo muitas versões do banco de dados ao mesmo tempo.

Você pode encontrar também software comercial que pode analisar um esquema de banco de dados antes e depois de uma mudança e, em seguida, gerar comandos SQL para alterar e / ou migrar os dados da versão antiga do banco de dados para a nova versão.

Há uma maneira muito simples de gerir a evolução de um banco de dados. Ela exige de versão que banco de dados e escrever um serviço que acompanhar as mudanças feitas em cada versão.

A idéia básica é a de adicionar uma tabela separada com uma coluna de "Version". Esta é a única tabela que deve não mudar no ciclo de vida do banco de dados. Deve conter também uma linha exclusiva especificando o atual versão do banco de dados. Esse valor é definido ao criar o banco de dados e atualizada cada vez que o banco de dados evolui.

As alterações feitas no banco de dados deve ser escrito como instruções SQL. Estas declarações vão geralmente criar novas tabelas, alterar os antigos, migrar dados de tabelas antigas para as novas e soltar essas tabelas antigas.

Agora, tudo que você precisa é escrever um sistema que mantém o controle das mudanças para cada versão. Aqui está uma exemplo simples:

```
DatabaseSystem.AddUpdate (1.0, 1.1, new string [] {declarações "SQL ..."});
```

Cada vez que uma nova versão do banco de dados está definido, as mudanças são adicionados a este sistema. Quando um banco de dados deve ser atualizado, este sistema irá ler sua versão actual, execute todas as mudanças feitas desde esta versão e defina a versão para seu novo valor. A migração é feito.

Note que você pode facilmente suportar desclassificação, se necessário.

Esta abordagem é semelhante ao utilizado pela Migrator open source do projeto:
<http://code.macournoyer.com/migrator/>.

Até agora, temos assumido que o banco de dados pode ser facilmente mapeados para o modelo de domínio. No entanto, alguns bancos de dados antigos podem ser mais difíceis de mapa.

9.2 esquemas Legacy e chaves compostas

Alguns dados requer um tratamento especial, além dos princípios gerais que discutimos no resto do livro. Nesta seção, iremos descrever tipos importantes de dados que introduzir uma complexidade extra em seu código NHibernate.

Quando seu aplicativo herda um esquema banco de dados existente legado, você quer fazer como poucos alterações no esquema existente quanto possível. Toda mudança que você faz poderia quebrar existentes outras aplicativos que acessam o banco de dados e exige a migração dos dados existentes caro. Em geral, não é possível construir um novo aplicativo e não fazer alterações aos dados existentes modelo de uma nova aplicação normalmente significa requisitos de negócios adicionais que naturalmente requerem evolução do esquema de banco de dados.

Vamos considerar, portanto, dois tipos de problemas: problemas que se relacionam com negócios em constante mudança requisitos (que geralmente não podem ser resolvidos sem alterações de esquema) e os problemas que se relacionam apenas a forma como você deseja representar o problema de negócio mesmo em seu novo aplicativo (que normalmente pode- mas não sempre ser resolvidos sem alterações de esquema do banco de dados). Geralmente você pode manchar o primeiro tipo de problema, olhando para a lógico modelo de dados. O segundo tipo mais frequentemente refere-se à implementação do modelo lógico de dados como um esquema de banco de dados físico.

Se você aceitar esta observação, você verá que os tipos de problemas que exigem alterações de esquema são aqueles que se chamam para adição de novas entidades, refatoração de entidades existentes, a adição de novos atributos existentes, e modificação das associações entre entidades. Os problemas que podem ser resolvido sem alterações de esquema geralmente envolvem definições de coluna inconveniente para uma entidade particular.

Vamos agrupar estes tipos de problema. Estes inconvenientes definições de coluna mais geralmente vêm em duas categorias:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

▪ Inconveniente tipos de coluna

Mencionamos que pensamos naturais chaves primárias são uma má idéia. Chaves naturais, muitas vezes fazê-lo difícil refatorar o modelo de dados quando a mudança exigências do negócio. Eles podem até, em extrema casos, o desempenho de impacto. Infelizmente, muitos usam esquemas legado (natural) chaves compostas fortemente, e, pela mesma razão que nós desencorajar o uso de chaves compostas, pode ser difícil mudar o esquema legado de usar chaves substitutas. Portanto, NHibernate suporta o uso de chaves naturais. Se o chave natural é uma chave composta, o apoio é através da `<composite-id>` mapeamento.

A segunda categoria de problemas pode geralmente ser resolvido usando um tipo de mapeamento personalizado NHibernate

(Implementar as interfaces `IUserType` ou `ICompositeUserType`), Conforme descrito no capítulo 7.

Vejamos alguns exemplos que ilustram as soluções para os problemas. Vamos começar com naturais mapeamentos de teclas.

9.2.1 Mapeamento de uma tabela com uma chave

natural

Nosso USUÁRIO tabela tem uma chave primária sintética, `USER_ID`, E uma restrição de chave única na NOME DE USUÁRIO.

Aqui está uma parte do nosso mapeamento NHibernate:

```
<class name="User" table="USER">
    <id name="Id" column="USER_ID">
        <generator class="native"/>
    </ Id>

    <Nome da versão = "Version"
        coluna = "VERSION" />

    <Nome da propriedade = "Username"
        coluna = "username"
        unique = "true"
        not-null = "true" />
    ...
</ Class>
```

Note que um mapeamento identificador sintética pode especificar um `unsaved` valor, Permitindo ao NHibernate determinar se uma instância é uma instância separada ou uma nova instância transitória. Assim, o seguinte trecho de código pode ser usado para criar um novo usuário persistente:

```
Usuário Usuário Usuário = new ();
usuário.username = "João";
usuário.Firstname = "João";
usuário.Lastname = "da Silva";
session.SaveOrUpdate (usuário) / / Gera valor id por efeito colateral
System.Console.WriteLine (session.GetIdentifier (usuário)) / / Imprime
Session.flush ();
```

1

Se você encontrou um USUÁRIO tabela em um esquema de legado, NOME DE USUÁRIO provavelmente seria a chave primária. Em

Neste caso, teríamos nenhum identificador sintético, em vez disso, nós usaria o atribuído gerador de identificador estratégia para indicar ao NHibernate que o identificador é uma chave natural atribuído pelo aplicativo antes de o objeto é salvo:

```
<class name="User" table="USER">
    <id name="username" column="USERNAME">
        <generator class="assigned"/>
    </ Id>

    <Nome da versão = "Version"
        coluna = "VERSION"
        unsaved-value = "-1" />
    ...
</ Class>
```

Nós já não pode tirar vantagem do `unsaved` valor atributo na `<id>` mapeamento. Um atribuído identificador não pode ser usado para determinar se uma instância é separada ou transitória, uma vez que é atribuído

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

pela aplicação. Em vez disso, especificar uma unsaved valor mapeamento para o <versão> propriedade. Fazendo assim consegue o mesmo efeito, essencialmente o mesmo mecanismo. O código para salvar uma nova Usuário não é alterado:

```
Usuário Usuário = new ();
user.username = "john" / Atribuir um valor de chave primária
user.Firstname = "João";
user.Lastname = "da Silva";
session.SaveOrUpdate (user) / Vontade, desde a versão é -1
System.Console.WriteLine (session.GetIdentifier (usuário)) / Imprime "john"
Session.flush ();
```

No entanto, teremos que alterar a declaração da propriedade de versão no Usuário classe para atribuir o valor -1 (versão private int = -1).

Se uma classe com uma chave natural não declarar uma propriedade timestamp ou versão, é mais difícil obter SaveOrUpdate () e cascatas para funcionar corretamente. Você pode usar um personalizado NHibernate IInterceptor como discutido mais adiante neste capítulo. (Por outro lado, se você está feliz em uso explícito Save () e explícito Update () em vez de SaveOrUpdate () e cascatas, NHibernate não precisa ser capaz de distinguir entre instâncias transitória e destacada, assim, você pode ignorar este conselho).

Composite chaves naturais estender as mesmas idéias.

9.2.2 Mapeamento de uma tabela com uma chave composta

Na medida em que NHibernate está em causa, uma chave composta pode ser tratado como um identificador atribuído de valor

tipo (o tipo NHibernate é um componente). Suponha que a chave primária da nossa tabela de usuário consistiu em um

NOME DE USUÁRIO e um Organization_id. Poderíamos acrescentar uma propriedade chamada Organizationid ao Usuário classe:

```
[Classe (Tabela = "USER")]
Usuário public class {

    [CompositeId]
    [KeyProperty (1, Nome = "Username" Coluna = "username")]
    [KeyProperty (2, Nome = "organizationid", Coluna = "organization id")]

    Nome de usuário {public string ... }
    public int organizationid {... }

    [Version (coluna = "VERSION", UnsavedValue = "0")]

    Versão public int {... }

    ...
}
```

Aqui está o mapeamento XML correspondente:

```
<class name="User" table="USER">

    <composite-id>
        <Nome da chave de propriedade = "Username"
            coluna = "username" />

        <Nome da chave de propriedade = "organizationid"
            coluna = "organization_id" />
    </ Composite-id>

    <Nome da versão = "Version"
        coluna = "VERSION"
        unsave-value = "0" />
    ...
</ Class>
```

O código para salvar uma nova Usuário ficaria assim:

```
Usuário Usuário = new ();
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    // Atribuir um valor de chave primária
    user.username = "João";
    user.OrganizationId = 37;

    // Definir valores de propriedade
    user.Firstname = "João";
    user.Lastname = "da Silva";
    session.SaveOrUpdate (user) // vai economizar, desde a versão é de 0
    Session.flush ();

```

Mas o objeto poderíamos usar como identificador quando chamado Load () ou Get ()? É possível usar um instância do Usuário, Por exemplo:

```

    Usuário Usuário = new ();

    // Atribuir um valor de chave primária
    user.username = "João";
    user.OrganizationId = 37;

    // Carrega o estado persistente em usuários
    Session.load (usuário, usuário);

```

Neste trecho de código, Usuário age como seu próprio identificador de classe. Note que agora temos de implementar

Equals ()/GetHashCode () para esta classe (e torná-lo Serializable). Podemos mudar isso usando um separados classe como identificador.

Usando uma classe de identificador composto

É muito mais elegante para definir um separado classe identificador composto que declara apenas a chave propriedades. Vamos chamar essa classe UserId:

```

[Serializable] public class {UserId
    string username privado;
    private string organizationid;

    UserId pública (string nome, string organizationid) {
        this.username username =;
        this.organizationId = organizationid;
    }

    // Propriedades aqui ...

    public override bool Equals (Object o) {
        if (o == null) return false;
        if (Object.ReferenceEquals (este, o)) return true;
        UserId userId = o como UserId;
        if (userId == null) return false;
        if (organizationid! = userId.OrganizationId)
            return false;
        if (nome! userId.Username =)
            return false;
        return true;
    }

    public override int GetHashCode () {
        retorno username.GetHashCode () + 27 * organizationId.GetHashCode ();
    }
}

```

É fundamental que nós implementamos Equals () e GetHashCode () corretamente, já que NHibernate usa esses métodos para fazer pesquisas cache. Além disso, o código de hash devem ser consistentes ao longo do tempo. Isso significa que

que, se a coluna NOME DE USUÁRIO é case insensitive, deve ser normalizada (para cima / cordas minúsculas) antes de usá-los. Composite classes principais são também deverá ser Serializable.

Agora, nós remover o UserName e Organizationid propriedades de Usuário e adicionar um UserId propriedade. Usaríamos o seguinte mapeamento:

```
<class name="User" table="USER">
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<composite-id name="UserId" class="UserId">
    <Nome da chave de propriedade = "UserName"
        coluna = "username" />

    <Nome da chave de propriedade = "organizationid"
        coluna = "organization_id" />
</ Composite-id>

<Nome da versão = "Version"
    coluna = "VERSION"
    unsaved-value = "0" />
.
.
.
</ Class>

```

Poderíamos economizar uma nova instância usando este código:

```

Usuário Usuário = new ();

// Atribuir um valor de chave primária
user.UserId UserId = new ("john", 42);

// Definir valores de propriedade
user.Firstname = "João";
user.Lastname = "da Silva";

session.SaveOrUpdate (user) // vai economizar, desde a versão é de 0
Session.flush ();

```

O código a seguir mostra como carregar uma instância:

```

Id UserId UserId = new ("john", 42);

Usuário user = (User) Session.load (typeof (User), id);

```

Agora, suponha que o Organization_id era uma chave estrangeira para a ORGANIZAÇÃO mesa, e que desejava para representar essa associação em nosso modelo C #. Nossa maneira recomendada para fazer isso é usar uma <Many-to-um> associação mapeada com insert = update "false" = "false", Como segue:

```

<class name="User" table="USER">

<composite-id name="UserId" class="UserId">
    <Nome da chave de propriedade = "UserName"
        coluna = "username" />

    <Nome da chave de propriedade = "organizationid"
        coluna = "organization_id" />
</ Composite-id>

<Nome da versão = "Version"
    coluna = "VERSION"
    unsaved-value = "0" />

<Many-to-one name = "Organização"
    class = "Organização"
    coluna = "organization_id"
    insert = update "false" = "false" />
.
.
.
</ Class>

```

Este uso de insert = update "false" = "false" NHibernate diz ignorar que a propriedade ao atualizar ou inserir um Usuário, Mas podemos, naturalmente, lê-lo com john.Organization.

Uma abordagem alternativa é usar um <key-many-to-one>:

```

<class name="User" table="USER">

<composite-id name="UserId" class="UserId">
    <Nome da chave de propriedade = "UserName"

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        coluna = "username" />

    <Key-many-to-one name = "Organização"
                      class = "Organização"
                      coluna = "organization_id" />
</ Composite-id>

    <Nome da versão = "Version"
                  coluna = "VERSION"
                  unsaved-value = "0" />
    ...
</ Class>

```

No entanto, normalmente é inconveniente ter uma associação em uma classe identificador composto, de modo que este abordagem não é recomendada, exceto em circunstâncias especiais.

Fazendo referência a uma entidade com uma chave composta

Desde USUÁRIO tem uma chave primária composta, qualquer tecla referência externa é também composta. Por exemplo,

a associação de Item para Usuário (O vendedor) agora é mapeado para uma chave composta estrangeiros. Para nosso alívio,

NHibernate pode esconder esse detalhe a partir do código C #. Podemos usar o mapeamento da associação a seguir para

```

Item: <many-to-one name="Seller" class="User">
      <column name="USERNAME"/>
      <column name="ORGANIZATION_ID"/>
</ Many-to-one>

```

Qualquer coleção de propriedade do Usuário classe também terá uma política externa de chave composta-por exemplo, o

inversa associação, Itens, Vendido por esse usuário:

```

<set name="Items" lazy="true" inverse="true">
    <key>
        <column name="USERNAME"/>
        <column name="ORGANIZATION_ID"/>
    </ Key>
    <one-to-many class="Item"/>
</ Set>

```

Note que a ordem em que as colunas são listadas é significativo e deve coincidir com a ordem em que eles aparecem dentro da <composite-id> elemento.

Vamos voltar ao nosso segundo problema legado esquema, colunas inconveniente.

9.2.3 Usando um tipo personalizado para mapear colunas de legado

A frase tipo de coluna inconveniente abrange uma ampla gama de problemas: por exemplo, o uso do CHAR (Em vez de VARCHAR) Tipo de coluna, o uso de uma VARCHAR coluna para representar dados numéricos, eo uso de uma valor especial em vez de um SQL NULL. É simples de implementar um IUserType implementação para lidar com o legado CHAR valores (aparando o corda devolvido pelos dados ADO.NET leitor), para executar as conversões de tipo entre tipos de dados numéricos e string, ou para converter valores especiais para um C #

nulo. Nós não vamos mostrar o código para qualquer um desses problemas comuns; vamos deixar isso para você, eles são todos

fácil se você estudar o capítulo 7, seção 7.1.3, "Criando tipos de mapeamento custom" com cuidado.

Nós vamos olhar para um problema um pouco mais interessante. Até agora, a nossa Usuário classe tem duas propriedades para representam nomes de um usuário: Firstname e Lastname. Assim que adicionar um Inicial, O nosso Usuário classe irá tornar-se confuso. Graças ao apoio do NHibernate componente, podemos facilmente melhorar o nosso modelo

com um único Nome propriedade de uma nova Nome Tipo C # (que encapsula os detalhes).

Também supor que há uma única NOME coluna no banco de dados. Precisamos mapear a concatenação de três propriedades diferentes de Nome a uma coluna. A seguinte implementação de IUserType demonstra como isso pode ser feito (nós fazemos a hipótese simplificadora de que o Inicial é

Por favor nos comentários ou correções para o fórum on-line em Autor

<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public class NameUserType: IUserType
{
    private static readonly NHibernate.SqlTypes.SqlType[] = SQL_TYPES
    {NHibernate.NHibernateUtil.AnsiString.SqlType};
    public NHibernate.SqlTypes.SqlType[] SqlTypes {get {SQL_TYPES return;}}
}

ReturnType Tipo pública {get {return typeof (Nome);}}
{public bool IsMutable
    get {return true;}
}

deepcopy objeto público (value object) {
    Nome = nome (Name) de valor;
    retorno novo nome (name.Firstname,
                       name.Initial,
                       name.Lastname);
}

new public bool Equals (objeto x, y objeto) {
    // Usar equals () a implementação da classe Nome
    return x == null? y == null: x.Equals (y);
}

pública objeto NullSafeGet (IDataReader dr, string [] nomes, proprietário do objeto) {
    cadeia dbname =
        (String) NHibernateUtil.AnsiString.NullSafeGet (dr, nomes);
    if (dbName == null) return null;
    string [] = dbName.Split tokens ();

    Realname name =
        novo nome (tokens [0],
                   tokens [1],
                   tokens [2]);
    retorno realname;
}

NullSafeSet public void ( IDbCommand cmd, objeto obj, int index) {
    Nome = nome (Name) obj;
    Cadeia namestring = (nome == null)?
        null:
        name.Firstname
        + '' + Name.Initial
        + '' + Name.Lastname;

    NHibernateUtil.AnsiString.NullSafeSet (cmd, namestring, index);
}
}

```

Observe que esta implementação delegados para um dos NHibernate built-in para alguns tipos funcionalidade. Este é um padrão comum, mas não é uma exigência.

Esperamos que agora você pode ver quantos tipos diferentes de problemas que têm a ver com o inconveniente definições de coluna podem ser resolvidos pelo usuário inteligente de NHibernate tipos personalizados. Lembre-se que cada

tempo NHibernate lê dados de um ADO.NET IDataReader ou grava dados em um ADO.NET IDbCommand. Ele passa por uma ITipo. Em quase todos os casos, que ITipo poderia ser um tipo personalizado. (Este inclui associações-a NHibernate ManyToOneType, Por exemplo, os delegados para o tipo de identificador a classe de associado, que pode ser um tipo personalizado.)

Um outro problema, muitas vezes surge no contexto de trabalho com dados legados: banco de dados integrando gatilhos.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

9.2.4 Trabalhando com triggers

Existem algumas motivações razoáveis para a utilização de triggers, mesmo em um novo banco de dados, para que os dados legado não é o único contexto em que surgem problemas. Gatilhos e ORM são muitas vezes uma problemática combinação. É difícil sincronizar o efeito de um disparo com a representação em memória de os dados.

Suponha que o ITEM tabela tem um CRIADO coluna mapeada para uma Criado propriedade do tipo DateTime, que é inicializado por um disparador de inserção. O seguinte mapeamento é apropriado:

```
<Nome da propriedade = "Criado"
  type = "Timestamp"
  coluna = "Criado"
  insert = "false"
  update = "false" />
```

Observe que esta propriedade mapa insert = "false" e update = "false" para indicar que ele não deve ser incluído no SQL INSERIR ou ATUALIZAÇÕES.

Depois de salvar um novo Item, NHibernate não vai estar ciente do valor atribuído a esta coluna pelo gatilho, uma vez que ocorre após o INSERIR ou linha do item. Se precisamos usar o valor em nossa aplicação, temos de dizer explicitamente NHibernate para recarregar o objeto com um novo SQL SELECIONE. Por exemplo:

```
Item Item item = new ();
...
NHibernateHelper.BeginTransaction ();
Sessão ISession NHibernateHelper.Session =;

Session.save (item);
Session.flush (); / Force / o INSERT a ocorrer
session.Refresh (item); / / Atualizar o objeto com um SELECT

System.Console.WriteLine (item.Created);

NHibernateHelper.CommitTransaction ();
NHibernateHelper.CloseSession ();
```

A maioria dos problemas envolvendo triggers podem ser resolvidos desta forma, usando uma explícita Flush () para forçar execução imediata do gatilho, talvez seguido de uma chamada para Refresh () para recuperar o resultado de o gatilho.

Você deve estar ciente de um problema especial quando você está usando objetos destacados com um banco de dados com gatilhos. Uma vez que não instantâneo está disponível quando um objeto destacado é re-associado a uma sessão utilização Update () ou SaveOrUpdate (), NHibernate pode executar SQL desnecessários ATUALIZAÇÃO declarações para assegurar que o estado do banco de dados é completamente sincronizado com o estado da sessão. Isto pode causar uma ATUALIZAÇÃO gatilho para disparar inconvenientemente. Você pode evitar esse comportamento, permitindo que select-before-table = "ITEM" select-before-update = "true"> atualizar no mapeamento para a classe que é mantido para a mesa com o gatilho. Se o ITEM tabela tem um gatilho de atualização, podemos usar o seguinte mapeamento:

Essa configuração força NHibernate para recuperar um instantâneo do estado atual banco de dados usando um SQL

SELECIONE, Possibilitando a posterior ATUALIZAÇÃO ser evitado se o estado da na memória Item é o mesmo.

Vamos resumir nossa discussão de modelos de dados legados: NHibernate oferece várias estratégias para lidar

com (natural) chaves compostas e colunas inconveniente. No entanto, nossa recomendação é que você examinar cuidadosamente se a uma alteração de esquema é possível. Em nossa experiência, muitos desenvolvedores

imediatamente ignorar as alterações do banco de dados de esquema como muito complexo e demorado, e eles olham para

uma solução NHibernate. Às vezes, essa opinião não se justifica, e nós pedimos que você considere esquema

Por favor postar comentários na seção para o tópico online em Autor a evolução de esquemas na perspectiva para o desenvolvimento dos seus dados. Se fazer alterações na tabela e exportação / [Importação](http://www.martinfrans.sandbox.com/forum.jspa?forumID=295)

dados resolve o problema, um dia de trabalho pode salvar muitos dias, no longo prazo, quando muitos soluções e casos especiais se tornam um fardo.

Agora que estamos a fazer em desenvolvimento e mapeamento dos dados secundários do modelo de domínio, é hora de cavar em seu comportamento e, especialmente, o quanto ele é suposto saber sobre a persistência.

9.3 ignorância persistência Compreensão

Na descrição das camadas de um aplicativo de NHibernate no capítulo 9, seção 9.1.3, destacamos que o modelo de domínio não deveria depender de qualquer outra camada ou serviço (embora não seja uma regra estrita). Este

é importante porque influencia a sua portabilidade. A menos dependente ele é, melhor ele pode ser modificado, testado e reutilizados.

Esta recomendação leva à noção de ignorância persistência (PI). A persistência ignorantes entidade é uma entidade que não tem conhecimento da forma como ele é mantido (que nem sequer sabe que ele pode ser persistentes).

Em termos práticos, isso significa que a entidade não tem métodos como `Save()` ou estática (de fábrica) métodos como `Load()` e não tem qualquer referência à camada de persistência. Este já é o caso para as entidades que têm escrito.

Indo um pouco além, poderíamos dizer também que as entidades não devem ter um Identificador propriedade e um

Versão propriedade. O motivo seria que as chaves primárias e controle otimistas têm nada a ver com o domínio do negócio. No entanto, eles certamente são úteis (e até mesmo obrigados a escrever eficiente aplicações).

Note-se que a ignorância persistência não é um requisito para a maioria dos modelos de domínio, é útil quando conscientização persistência iria dificultar a portabilidade e a flexibilidade do modelo de domínio. No Apêndice C, apresenta-se uma biblioteca chamada ActiveRecord que os sacrifícios ignorância persistência em favor da simplicidade.

Agora, vamos ver como podemos implementar uma entidade que é tão livre de persistência relacionadas código como possível, enquanto ainda está sendo funcional.

9.3.1 persistência relacionadas Abstraindo código

Um compromisso comum, no nível de consciência de persistência, é separar a persistência código relacionado a partir do código de negócios na implementação de uma entidade. Você pode simplesmente executar um visual separação usando `# Region` a fim de melhorar a legibilidade do código, ou você pode ir tão longe a ponto de criar uma classe base abstrata para entidades a lidar com isso.

Vamos dar uma olhada em como podemos implementar a última solução. Nós usaremos `NHibernate.Mapping.Attributes` porque permite que a classe base para as informações de mapeamento abstrato junto com o código. Você verá que o resultado final pode ser aceitável desde que você não se importa herdar desta classe base (se você mente, basta copiar o conteúdo dessa classe em suas entidades).

Note que esta implementação apresenta muitas idéias independentes e padrões; sinta-se livre para extrair alguns deles para suas aplicações.

Nós vamos implementar uma classe abstrata que as entidades podem herdar de ganhar a persistência código relacionado que eles precisam. Esta classe irá fornecer um identificador e uma propriedade junto com a versão adequada sobrecarga de `System.Object` métodos. Vamos chamar esta classe `VersionedEntity` e você pode ver a sua aplicação na listagem 9.2.

Classe base Liaring 9.2 abstraindo persistência código relacionado

```
[Serializable]
VersionedEntity public abstract class {

    privada Guid id = Guid.NewGuid ();

}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

private int version = 0;

[Id (Name = "Id" Access, = "nosetter.camelcase-sublinhado")]
[Generator (1, Classe = "atribuído")]
public virtual Guid Id {
    get {return;}
}

[Version (Access = "nosetter.camelcase-sublinhado")]
Versão int {public virtual
    get {versão return;}
}

public override string ToString () {
    retorno GetType () FullName + "#" + Id.;
}
public override bool Equals (object obj) {
    Entidade VersionedEntity obj = como VersionedEntity;
    if (entidade == null) return false;
    retorno Id == entity.Id;
}
public override int GetHashCode () {
    retorno Id.GetHashCode ();
}

```

}

Usando um atribuído Guid como identificador # 1 oferece muitas vantagens. Por exemplo, simplifica-se o implementação de Equals () e GetHashCode () .

A versão # 2 é usado para controle de simultaneidade otimista, explicado no capítulo 6, seção 5.2.1, "Usando controle de versão gerenciado".

Estes implementação # 3 de System.Object métodos são simples, mas eficaz.

Note que você pode substituir a inicialização do identificador por:

```
privada Guid id
    = (Guid) NHibernate.Id.GuidCombGenerator new () Gera (null, null);
```

Esta inicialização usa o guid.comb gerador de identificador. Você pode ler as suas vantagens na tabela 4.1 do capítulo 4, seção 4.3.3.

No entanto, como você não deveria fazer referência a NHibernate aqui, você deve criar um private static methodintoVersionedEntityusingthesamealgorithmosthemethod
GuidCombGenerator.GenerateComb (). Lembre-se que NHibernate está licenciado sob a LGPL; portanto, itssourcecodeispublishedavailable. Formoredetails, leia http://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License.

Implementação de persistência de entidades abstratas

Ao usar esta classe, suas entidades só terão a mapear suas propriedades de negócio. Ele vai herdar o que a camada de persistência (e NHibernate) precisa: O identificador, a versão ea sobrecarga dos System.Object métodos.

Você pode se perguntar se o uso de NHibernate.Mapping.Attributes diminui a persistência ignorância de suas entidades (como é sobre o mapeamento, que é uma preocupação a persistência).

Isso é verdade, de certa forma. No entanto, os atributos não têm comportamento e não adicionar qualquer restrição para o seu modelo. Eles são apenas como documentação, uma documentação útil, pois a impedância incompatibilidade não é magicamente resolvido. Você deve sempre lembrar que o modelo é mantido em um banco de dados de relação e que tem algumas implicações. Os prós e contras de atributos são enumeradas no capítulo 3, seção 3.4.

Vamos dar um exemplo simples para ilustrar o aspecto documentação destes atributos:

[Classe]

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

Usuário public class : {VersionedEntity
    private string name;

    [Propriedade (Comprimento = 64 Access, = "field.camelcase-sublinhado")]
    string Nome {...}
}

```

Sem as informações Comprimento = 64, Um desenvolvedor desatento poderia pensar que os nomes podem ser ilimitadas em comprimento e do usuário final obterá uma aplicação que trunca o seu nome por uma razão desconhecida.

By the way, como você pode ver, usando VersionedEntity torna esta implementação completamente livre de código não relacionada com o domínio do negócio enquanto está a ser totalmente funcional.

O fato de que o modelo de domínio não tem conhecimento de outras camadas (como a camada de apresentação) significa que não pode informá-los diretamente, sempre que acontece alguma coisa nele (por exemplo, para notificar quando um mudança é feito para que o GUI é atualizada). Felizmente, existe um padrão para resolver este tipo de problema.

9.3.2 Aplicando o padrão Observer para uma entidade

O padrão Observer permite que um assunto para notificar os observadores sem um link codificado entre eles. Este padrão é frequentemente usado em uma arquitetura MVC, como explicado no capítulo 8, seção 8.1.1.

Esse padrão pode ser usado, por exemplo, para que a interface do usuário registre para observar um evento que podem ser levantadas pela entidade, sempre que seu estado interno muda.

Para implementar este padrão, você precisa adicionar o evento para a entidade. Então, o observador deve registrar-se para o evento para começar a ouvir. Na maioria das vezes, o registro é feito apenas após a criação ou carregamento da entidade.

Vamos dar um exemplo para ilustrar como implementar este padrão. No exemplo anterior, a classe Usuário tem uma propriedade Nome. Se queremos informar a camada de apresentação quando esta propriedade mudanças, aqui é o caminho (e ruim) direto:

```

Usuário public class {

    Nome {public string
        get {return nome;}
        conjunto {
            if (nome == valor) return;
            nome = valor;
            PresentationLayer.User_NameChanged (this);
        }
    }

}

```

Aqui, assumimos que a entidade tenha acesso à camada de apresentação que fornece um método para chamar quando ele muda. O problema, nesta implementação, é que a entidade é ligada à camada de apresentação e isso é muito ruim, porque, você não pode usá-lo em qualquer outro contexto (por exemplo, quando o teste).

Aqui está a solução usando o padrão Observer:

```

NameChangedEventArgs public void delegate (
    object sender, EventArgs e);

Usuário public class {

    private string name;

    Nome {public string
        get {return nome;}
        conjunto {
            if (nome == valor) return;
            nome = valor;
            OnNameChanged ();
        }
    }

}

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        }
        NameChangedEventHandler evento público NameChanged;
        protected void virtuais OnNameChanged () {
            if (NameChanged! = null)
                NameChanged (this, EventArgs.Empty);
        }

    }
}

```

Nós definimos um delegado para o evento chamado NameChanged. Em seguida, na implementação da propriedade (Aqui Nome), Nós levantamos o evento depois de mudar o valor. Usamos o método OnNameChanged () para que porque é uma diretriz recomendada. NET. Leia a documentação oficial do. NET para mais detalhes sobre a implementação eo uso de eventos.

O próximo passo é ouvir o evento:

```
User = usuário BusinessLayer.LoadUser (userId);
user.NameChanged += User NameChanged;
```

Neste código, nós carregamos um usuário e registrar o NameChanged logo após o evento. O método User_NameChanged () será chamada sempre que esta alteração de propriedade.

Note-se que o quadro. NET fornece uma interface para esse cenário. Ele é chamado INotifyPropertyChanged. Aqui está uma implementação do Usuário classe herdando essa interface:

```
using System.ComponentModel;

Usuário public class: INotifyPropertyChanged {

    private string name;

    Nome {public string
        get {return nome;}
        conjunto {
            if (nome == valor) return;
            nome = valor;
            OnPropertyChanged ("Nome");
        }
    }
    PropertyChangedEventHandler evento público PropertyChanged;
    protected void virtuais OnPropertyChanged (propertyName string) {
        if (PropertyChanged! = null)
            PropertyChanged (esta, novo PropertyChangedEventArgs (propertyName));
    }
}
```

}

Esta solução é similar ao anterior. No entanto, ele irá trabalhar para todas as propriedades e pode ser útil em outras situações (como a ligação de dados).

O padrão Observer também pode ser usado em muitas outras situações. Por exemplo, para fins de segurança:

```
SecurityService public class {

    public void User IsAdministratorChanging (object sender, EventArgs e) {
        if (! loggedUser.IsAdministrator)
            throw new SecurityException ("Não permitido");
    }
}
```

Aqui, o serviço de segurança escuta para o evento User.IsAdministratorChanging. É, portanto, capaz de cancelar uma modificação lançando uma exceção se o usuário conectado não é um administrador.

Esta seção explica como evitar sobrecarregar o modelo de domínio com preocupações não relacionadas. Agora, vamos falar sobre a sua principal preocupação: A lógica do negócio.

9.4 Implementação da Lógica de Negócios

Neste livro, a que chamamos lógica de negócios qualquer código que determina como as entidades devem se comportar. Ele define

que pode ser feito com as entidades e regras de negócio impõe sobre os dados que eles contêm.

Note que não são, estritamente falando sobre o modelo de domínio único, mas sobre a camada de negócios em geral.

A camada de negócios pode conter muitos tipos de lógica de negócios. Vamos usar um pouco de estudo de caso para cobrir todos eles. Digamos que queremos implementar um sistema de sub-da nossa aplicação que CaveatEmptor permite aos usuários colocar um lance em um item e, quando isso for feito, todos os outros candidatos são notificados desta nova lance via e-mail.

Não é possível fazer isso dentro do Oferta ou o Item entidades, pois o envio de e-mails não é a sua responsabilidade. A camada de negócios deve fazê-lo. Aqui está um exemplo de como:

```
usando (ItemDAO persister = new ItemDAO ()) {  
  
    Item item = persister.LoadWithBids (itemId);  
    item.PlaceBid (bid);  
  
    foreach (bid Bid em item.Bids)  
        Notifique (bid.Author, o lance, ...);  
    }  
}
```

Antes de prosseguir, é preciso explicar alguns detalhes: Este método usa o identificador do item e um número para fins de conveniência. Na sua execução, ele carrega o item usando um DAO chamada ItemDAO.

Para mais detalhes sobre o padrão DAO, leia o capítulo 11, seção 11.1. Esta classe pode carregar itens e automaticamente persistem as alterações feitas quando ele é fechado por `usando ()` porque mantém o ISession exemplo que ele usa aberta.

Depois de carregar o item, nós colocamos o lance e notificar os autores de todos os outros lances. Note-se que, porque a sessão NHibernate ainda está aberta, o Lances coleção pode ser preguiçosamente carregado. No entanto, como sabemos que vai precisar dele, usamos um `Load ()` método que carrega-lo ansiosamente, que resultam em uma rápida aplicação.

Agora, vamos dissecar esse método para ver como vários tipos de lógica de negócios são executados quando é chamado.

9.4.1 A lógica do negócio na camada de negócios

O método `BusinessLayer.PlaceBid ()` contém a lógica que pertence à camada de negócios. De lado código executivo como este, ele também pode conter regras de negócio, como validação de segurança:

```
BusinessLayer.PlaceBid (int itemId lance Bid) {  
  
    If (loggedUser.IsBanned)  
        throw new SecurityException ("Não permitido");  
    / / ...  
}
```

Aqui, antes de colocar o lance, nós temos certeza que o usuário conectado não é proibido.

Também é comum o uso do IInterceptor API quando uma regra de negócio está ligado à persistência de entidades. Leia a seção 9.4 do capítulo 9 para ver como ele é feito. Para regras de negócio complexas, você pode considerar o uso de um mecanismo de regras.

A lógica de negócios restante é dentro do modelo de domínio.

9.4.2 A lógica de negócios no modelo de domínio

A lógica de negócios em uma entidade é uma codificação do que esta entidade é suposto fazer. Aqui é um simples

implementação do método Item.PlaceBid ():

```
PlaceBid public void (bid Bid) {
    if (lance == null)
        throw new ArgumentNullException ("bid");
    if (bid.Amount < CurrentMaxBid.Amount)
        throw new BusinessException ("Lance muito baixo.");
    if (this.EndDate < DateTime.Now)
        throw new BusinessException ("Leilão já terminou.");
    Bids.Add (bid);
    CurrentMaxBid bid =;
}
```

Esta implementação tem a finalidade de ilustrar o tipo diferente de lógica de negócios. Capítulo 11 contém uma discussão detalhada sobre a implementação real deste método.

Nesta implementação, começamos com verificação de vários e então executar a ação. Um específica tipo de verificação é a verificação de integridade de dados. Ele pode ser facilmente identificado quando se impede que o usuário de atribuir um valor que entra em conflito com sua natureza. É o caso quando, por exemplo, não permitir que um string nula como nome do usuário.

Às vezes, acontece que alguma lógica devem ser executados em um tempo específico (por exemplo, antes salvar a entidade). Este é o caso quando a muitas propriedades deve ser preenchido antes da mão. Neste caso, você pode adicionar um método que será chamado na época.

Por exemplo, se alguma lógica de validação deve ser realizada antes de salvar uma entidade, um método deve ser adicionado para que nesta entidade, que poderia ser chamado Validate (). Este método pode ser chamado a cada tempo que a camada de persistência é chamado, ou usando o IInterceptor API. Como para a sua implementação, validar as propriedades individuais pode ser feito através da reutilização da verificação feita nessas propriedades (em ordem para evitar duplicação de código). Por exemplo, com a implementação desta propriedade:

```
Nome {public string
    get {return nome;}
    conjunto {
        if (string.IsNullOrEmpty (valor))
            throw new BusinessException ("Nome obrigatório.");
        if (nome == valor) return;
        nome = valor;
    }
}
```

Podemos testar se o nome está corretamente inicializado sem duplicar a verificação. Nós apenas tem que escrever:

```
public void validar () {
    Nome =;
```

Este truque legal só funciona se a verificar: if (string.IsNullOrEmpty (valor)) é feito antes o código: if (nome == valor) return; uma vez que são feitos com a validação das propriedades de um por um, podemos fazer as validações entre as propriedades como, por exemplo, verificar que o valor mínimo é realmente menor do que o máximo.

Também é comum a demora na validação das propriedades até que a entidade deve ser salvo. Isso é útil quando as propriedades pode manter valores intermediários que não são válidas ainda.

Ao implementar a lógica de negócios do modelo de domínio, você deve ter cuidado para evitar dependências indesejáveis. Eles devem ser mover-se a uma camada superior (na maioria das vezes, a camada de negócios).

Por exemplo:

```
Senha {public string
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        if (CryptographicService.IsNotAStrongPassword (valor))
            throw new BusinessException ("Não é forte o suficiente.");
        password = valor;
        MailingService.NotifyPasswordChange (this);
    }

}

```

Esta é obviamente errada; o modelo de domínio não deve depender destes criptográfico e de discussão serviços. A camada de negócios deve conter essa lógica.

Existem também alguns regras que não é suposto ser implementado no modelo de domínio e não mesmo no próprio aplicativo.

9.4.3 As regras que não são regras de negócio

Alguns regras não deve ser implementado em qualquer camada da aplicação. Uma maneira fácil de encontrá-los é ver se elas realmente são regras de negócio.

Estes regras são geralmente testar o código, para se certificar de que tudo correu como esperado. Para exemplo:

```

PlaceBid public void (int itemId lance Bid) {

    ...
    item.PlaceBid (bid);

    if (! item.Bids.Contains (bid))
        throw new Exception ("PlaceBid () falhou.");
    ...
}

```

Isso é errado porque, se Item.PlaceBid () falhar, ele já deve lançar uma exceção, pois este regra deve ser um teste que garante que uma exceção é lançada quando realmente Item.PlaceBid () falha.

Vamos dar outro exemplo:

```

pública LoadMinAndMaxBids void () {
    ...
    min = BidDAO.LoadMinBid (item);
    max = BidDAO.LoadMaxBid (item);

    if (max <min)
        throw new Exception ("Algo está errado com as consultas");
    ...
}

```

Este é um teste para a camada de persistência. Se ele falhar, isso significa que existe um bug em sua implementação.

Para mais detalhes sobre o teste, leia o capítulo 9, seção 9.1.

Até agora, temos implementado a estrutura interna do modelo de domínio, cuidando de seus dados e seu comportamento. Agora, é hora de tratar de algumas questões relacionadas com o ambiente em que este domínio modelo é usado.

9.5 Dados entidades de ligação

A camada de apresentação é utilizado pelo usuário final para mostrar e modificar as entidades de nossa NHibernate

aplicação. Isto implica que os dados dentro de nossas entidades são exibidos usando. Controles NET GUI e que as entradas do usuário final são enviados de volta para mudar essas entidades.

Embora esta transferência de dados pode ser feito manualmente., NET fornece uma maneira de criar uma ligação entre uma objeto (chamado de fonte de dados) e um controle para que uma alteração em um deles é reverberou ao outras. Desta forma é chamado ligação de dados. No contexto do NHibernate, não vamos chamar esses objetos entidades, mas POCOs, a ênfase do fato de que eles não têm qualquer infra-estrutura especial para facilitar os dados

vinculativo (que é o caso de DataSets, por exemplo).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

As pessoas costumavam DataSets (e os magos da NET do Visual Studio.) Encontrar a ligação de dados bem diferentes quando começar a escrever POCOs. Felizmente, a maioria. Controles NET GUI suporte de dados básicos de ligação para POCOs e .NET fornece algumas interfaces para melhorar esse apoio.

Nesta seção, vamos discutir uma série de alternativas para a ligação de dados; essas alternativas se aplicam igualmente para aplicações Windows e Web. Vamos primeiro ver como podemos interagir. GUI NET controles sem o uso de ligação de dados. Então, vamos ligar dados de POCOs e enumerar uma série de extensões que melhorar essas capacidades. Também vamos ver como NHibernate pode ajudar a implementar ligação de dados. Finalmente, vamos descobrir uma biblioteca que ajuda muito quando POCOs ligação de dados.

Existem três tipos de dados em um POCO: Uma propriedade simples (cujo tipo é um tipo primitivo), uma referência a outro POCO (como componente ou muitos-para-um relacionamento), e uma coleção de Poços (ou primitivas).

Nesta discussão, vamos ignorar a referência a outro POCO porque é geralmente visualizados, exibindo uma de suas propriedades (o seu nome, por exemplo) e um mecanismo especial (um botão, para exemplo) é fornecido para visualizar ou editar esta referência.

A fim de cobrir como o tipo outros dois de dados (propriedades simples e coleções) podem ser dados obrigado, vamos pegar o exemplo da escrita uma forma de gerir um usuário e seus detalhes de faturamento (conforme definido para o nosso aplicativo de leilão no capítulo 4, seção 4.1.2). Este formulário irá receber o usuário e permitem-nos atualização dele e de sua lista de detalhes de faturamento.

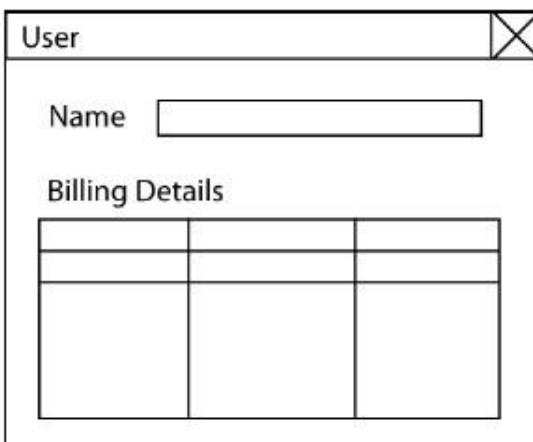


Figura modelo de Domínio 9,3 vinculado a um interface de utilizador

O aspecto interessante deste exemplo é que BillingDetails é uma classe abstrata, portanto, as entidades no coleta pode ser BankAccount ou CreditCard instâncias. Esta complicação nos dará a oportunidade de ver as limitações de algumas abordagens.

Note que não damos uma explicação completa do .NET APIs, vamos utilizar, se você precisa aprender mais sobre eles, consulte a documentação oficial .NET. Você também pode ler o livro de Dados Ligação com o Windows Forms 2.0 [Noyes 2006].

Vamos começar por ignorar todas as APIs e exibir essas / recuperação de dados manualmente.

9.5.1 Implementação de ligação de dados manual

A idéia por trás dessa abordagem é muito simples: Nós copiar os dados de Poços de / para o GUI. Quando nós necessidade de mostrar algo, nós levá-la a partir do POCO e enviá-lo para o GUI:

```
editName.Text = user.Name;
```

Quando precisamos processar esses POCOs, nós recuperamos quaisquer alterações na GUI e aplicá-los de volta em o POCOs:

```
user.Name = editName.Text;
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Para a recolha, devemos acrescentar seus itens um por um no controle.

Esta abordagem é simples de entender e implementar. Também é bom para a personalização. Para exemplo, ao exibir um identificador (do tipo inteiro), você pode decidir exibir Novo para um entidade transitória (em vez de 0).

É também simples de apoio polimorfismo:

```
if (billingDetails é BankAccount) {  
    ...  
    editBankName.Text = (billingDetails como BankAccount).BankName;  
}  
else {  
    ...  
    editExpYear.Text = (billingDetails como CreditCard).ExpYear.ToString();  
}
```

No entanto, pode ser entediante implementar para objetos complexos e é menos interessante no Windows aplicações porque eles podem ter formas muito falador.

9.5.2 Usando controles ligados a dados

Neste caso, contamos com o apoio da associação de dados de propriedades públicas e coleções em controles. Aqui

é um exemplo para o Nome:

```
editName.DataBindings.Add ("Text" do usuário, "Nome");
```

Neste exemplo, controlar o editName (A TextBox) É dados vinculados à propriedade Nome do Usuário exemplo. Para a recolha, a solução mais simples é usar o DataSource propriedade do controle:

```
DataGridView = user.BillingDetails;
```

O DataGridView controle usará o BillingDetails coleção como fonte de dados. No entanto, este solução é muito limitada. Por exemplo, ele não suporta polimorfismo, o que significa que você pode apenas editar as propriedades da classe BillingDetails. Você não pode editar as propriedades das subclasses BankAccount e CreditCard.

Classes auxiliares numerosas e as extensões podem ser usadas para melhorar esse apoio: ObjectDataSource, BindingSource;BindingList, IEditableObject, INotifyPropertyChanged, Etc Sugerimos que você olha para essas APIs para ver quais as suas necessidades.

Se você realmente valor a simplicidade em seu modelo de domínio e ainda quer fazer uma poderosa ligação, você pode implementar classes de embrulho (usando o Padrão de adaptador) que representaria um apresentação modelo:

```
DetailsWrapper BillingDetailsWrapper BillingDetailsWrapper = new (detalhes);  
editBillingDetails.DataSource detailsWrapper =;
```

Neste caso, você tem duas classes com finalidades específicas que lhe dão mais controle: A entidade mantém foco no seu valor de negócio eo wrapper fornece recursos de ligação de dados em cima da entidade.

No entanto, eles também lhe dará mais trabalho que você tenha duas classes para implementar, em vez de um.

Outro benefício é que você pode adicionar algumas propriedades para relatórios propósito. Um exemplo comum é a adicionar um FullName propriedade que retorna o primeiro nome eo sobrenome concatenados.

9.5.3 Dados NHibernate usando vinculativo

Se você pensar sobre as obras NHibernate forma, você vai perceber que ele faz algum tipo de ligação de dados: Quando você carrega uma entidade, enche-lo quando seus dados, e quando você salvá-lo, ele recupera os dados.

É possível acessar a parte do NHibernate responsável deste trabalho e usá-lo para os nossos benefícios. Capítulo 3, seção 3.5.10 explica como usar essa API. Ele mostra como extrair os nomes dos

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

propriedades em uma entidade, juntamente com os valores contidos em um exemplo. Aqui está um código com base nessa explicação:

```
User = usuário UserDAO.Load (userId);

NHibernate.Metadata.IClassMetadata meta =
    sessionFactory.GetClassMetadata (typeof (User));

string [] = meta.PropertyNames;
object [] = propertyValues meta.GetPropertyValues (usuário);

for (int i = 0; i < meta.PropertyNames.Length; i++) {
    Label label = new ();
    label.Text = meta.PropertyNames [i];
    Controls.Add (label);

    TextBox editar = new TextBox ();
    edit.Text = propertyValues [i].ToString ();
    Controls.Add (edit);
}
```

Esta implementação simplista recupera os dados de um usuário e gera rótulos e caixas de texto para exibi-los. Observe que a posição desses controles devem ser definidos.

A interface IClassMetadata também tem o método: SetPropertyValues (entidade objeto, objeto [] valores), Que pode ser usado para copiar os dados do GUI para a entidade. Note que você deve mantê-los na mesma ordem que quando você carregou-los.

Embora essa abordagem pareça poderosa, ela tem vários inconvenientes que não são aceitáveis em aplicativo de produção: Mesmo com um algoritmo bem projetado, o layout resultante da GUI será longe de ser perfeito. Você pode ter alguns problemas com a formatação dos valores (por exemplo, datas). Há controles que melhor TextBox para alguns tipos de dados (por exemplo, DateTimePicker). Finalmente, esta abordagem exige uma quantidade extra de trabalho para apoiar as referências a outros POCOs e coleções.

É certamente possível resolver estas questões com alguns esforços, e essa abordagem pode ajudar quando prototipagem pedido. Portanto, você deve adicionar à sua caixa de ferramentas.

9.5.4 Ligação de dados usando ObjectViews

ObjectViews é uma biblioteca de código aberto escrito especificamente para ajudar POCOs dados ligam. NET do Windows controles.

É em grande parte fora do âmbito deste livro para cobrir esta biblioteca. No entanto, vale a pena mencionar que suporta a ligação de dados de Poços individuais e de coleções.

Note-se que, no momento da redação deste texto, ObjectViews ainda é baseado em. NET 1.1 e não evoluir anymore.

Você pode baixar esta biblioteca (com uma aplicação de exemplo útil) em seu site:
<http://sourceforge.net/projects/objectviews/>.

9.6 Preencher um DataSet com dados de entidades

DataSets são amplamente utilizados, principalmente por data-centric aplicações alavancar os assistentes agradável de ferramentas como Visual Studio. NET para gerar o código. No entanto, eles são bastante diferentes para Poços. Se, por alguma razão, seu modelo de domínio deve de alguma forma comunicar com um componente usando DataSets, você terá que encontrar uma solução para este problema.

Antes de começar, lembre-se que você sempre pode executar código ADO.NET clássico, abrindo um conexão com o banco, deixando-se ou NHibernate fazê-lo e usar o ISession.Connection propriedade para recuperá-la. Neste caso, você terá que ter cuidado para não trabalhar com dados desatualizados ou mudar algo sem limpar o relacionadas cache de segundo nível NHibernate. Para mais detalhes, leia capítulo 6, seção 6.3.

Você pode também considerar reescrever o componente usando DataSets para uma melhor consistência.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Se ambas as opções não são aplicáveis, você terá que converter suas entidades de / para DataSets.

Nas seções seguintes, vamos considerar apenas indo de entidades para um DataSet preenchido com os seus de dados. Você não deve ter nenhum grande problema reverter esse processo.

9.6.1 conversão manual

ADataset é um container em memória de dados que imita a estrutura de um banco de dados relacional.

Enchimento

isso significa que a adição de linhas à sua tabelas. É relativamente fácil de descobrir o código necessário para encher um DataSet.

Aqui, assumimos que você está trabalhando com dataset tipado, já que são mais fáceis de trabalhar.

Listagem 9.3 contém um método que faz este trabalho para o Item entidade. É completo porque lida com as propriedades simples, a referência Vendedor ao Usuário entidade e os Lances coleção.

Listagem 9.3 Preenchendo um DataSet com o conteúdo de uma entidade

```
private static ArrayList adicionando = new ArrayList ();  
  
FillDataSet public static void (TypedDataset dataset item) {  
  
    if (adding.Contains (item)) | 1  
        retorno; | 1  
    adding.Add (item); | 1  
  
    Linha TypedDataset.ItemRow; | 2  
    if (dataset.Item.Rows.Contains (item.ItemID)) | 2  
        row = dataset.Item.FindByItemID (item.ItemID); | 2  
    outro | 2  
        linha dataset.Item.NewItemRow = (); | 2  
  
    row.ItemID = item.ItemID; | 3  
    row.Name = item.Name; | 3  
    ... | 3  
  
    if (item.Seller == null) | 4  
        row.SetSellerIDNull (); | 4  
    else { | 4  
        if (! dataset.User.Rows.Contains (item.Seller.UserID)) | 4  
            FillDataSet (dataset, item.Seller); | 4  
        row.SellerID = item.Seller.UserID; | 4  
    } | 4  
  
    if (NHibernateUtil.IsInitialized (item.Bids)) | 5  
        foreach (bid Bid em item.Bids) | 5  
            FillDataSet (dataset, bid); | 5  
  
    if (! dataset.Item.Rows.Contains (item.ItemID)) | 6  
        dataset.Item.AddItemRow (linha); | 6  
  
    adding.Remove (item); | 7  
}
```

A coleção é usado para manter a lista de entidades que estão sendo adicionadas # 1. Isso é necessário para evitar chamadas recursivas infinitas quando há uma referência circular.

Se a entidade já está no DataSet, ele deve ser atualizado, caso contrário sua linha deve ser criada # 2.

Preenchendo as propriedades simples é simples # 3.

Manipulação de referências a outras entidades é um pouco mais complexo: Devemos ou defini-lo como nulo ou adicioná-lo

se não for no DataSet ainda # 4.

Manipulação de coleções requer que primeiro se certificar de que já está carregado (a menos que, no seu caso, você

quer que ele seja carregado peregrino). Então só precisamos adicionar a um por um os lances # 5.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Agora, nós podemos adicionar a linha, se for um novo #

6.

Finalmente, remova a entidade a partir desta coleção # 7 porque são feitas acrescentando-lo.

Se você estiver usando um gerador de código como explicado na primeira seção deste capítulo, você pode ser capaz de

gerar este código para todas as suas entidades. Seria uma enorme salva de tempo.

Agora, vamos ver como NHibernate pode ajudar a atingir o mesmo resultado um pouco mais rápido.

9.6.2 conversão NHibernate assistida

Se você estiver trabalhando com um não-digitadas DataSet, É possível utilizar uma abordagem semelhante à explicado na seção 9.5.3: Você pode extrair os nomes e propriedades "as classes nomes e usá-los como "nomes e colunas 'tabelas nomes.

Na verdade, há um método em que o NHibernate é uma implementação próximos desta idéia. É o método ToString (entidade objeto) da classe NHibernate.Impl.Printer. Dê uma olhada em sua implementação antes de iniciar sua implementação.

Sucesso para implementar esta abordagem significa que ele é genérico o suficiente para trabalhar com qualquer entidade, pois você só será manipular metadados.

No entanto, isso significa que o modelo de domínio dita o esquema da DataSet. Note-se que, nesta edição pode ser resolvido usando o mapeamento entre o modelo de domínio eo banco de dados (como o DataSet esquema é geralmente baseado nele).

Com este suporte de comunicação com um componente usando DataSetS, estamos a fazer implementar nosso modelo de domínio do mundo real.

9.7 Resumo

Escrever modelos de domínio do mundo real pode ser particularmente difícil por causa da influência de sua ambiente. Esperemos que este capítulo tenha ajudado a compreender esse processo.

O primeiro passo é implementar o modelo de domínio, o banco de dados e escrever o mapeamento entre esses dois. Antes deste capítulo, sempre escreveu-los manualmente, um por um. Agora, nós sabemos como gerar -los. Nós até temos uma idéia de como podemos automatizar a migração do banco de dados como o domínio modelo evolui.

Ao escrever o mapeamento, este capítulo explicou como lidar com bancos de dados legados. NHibernate suporta o mapeamento de chaves naturais e compostas. Como último recurso, é possível implementar usuário tipos para lidar com situações personalizado. Também é possível trabalhar com um banco de dados usando triggers.

Após a implementação e mapeamento dos dados do modelo de domínio, nos mudamos para a sua lógica de negócios. Nós explicou o que significa a persistência da ignorância e como escrever um modelo de domínio limpa livre de indesejados dependências. Então, fizemos uma lista de diferentes tipos de lógica de negócio, explicando como eles devem ser implementadas. Nós também deu alguns conselhos de erros a evitar.

Quando tivermos concluído o modelo de domínio, precisamos exibi-la. Este é o lugar onde a ligação de dados vem para jogar. Como vimos, ele pode exigir uma boa quantidade de trabalho a ser feito corretamente.

Concluímos este capítulo analisando como podemos obter uma DataSet a partir do conteúdo de um entidade. Vimos que, embora possa exigir uma quantidade razoável de tempo no início, este processo pode ser automatizado.

Este capítulo foi apenas uma abertura para o mundo real de modelos de domínio. Você ainda pode precisar de fazer alguma pesquisa para encontrar a resposta perfeita para suas necessidades.

Agora, é hora de passar para outra camada: a camada de persistência. Até agora, temos sido escrita simples e operações de persistência curta. É hora de olhar para o que o mundo real parece daquele lado.

10

Técnicas Avançadas de Persistência

Este capítulo aborda

- „ Projetando a camada de persistência
- „ Implementação reutilizáveis Data Access Objects
- „ Conversas implementação
- „ Suportar transações Enterprise Services
- „

Você tenha atingido o último capítulo deste livro. Nos capítulos anteriores, você aprendeu como implementar persistência em uma aplicação .NET. Você já descobriu as características de NHibernate e você aprendeu a projetar um aplicativo em camadas.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Com esse conhecimento, você deve ser capaz de criar o modelo de domínio, mapeá-lo para o banco de dados e implementar a camada de negócios e a camada de apresentação. No entanto, não deram detalhes suficientes sobre a camada de persistência. No capítulo 2, a classe Persister representa a camada de persistência. Em um do mundo real aplicação, você vai precisar muito mais do que isso.

Este capítulo começa com a apresentação do Data Access Object (DAO) padrão. É um popular padrão que trata da organização da camada de persistência. Vamos ir tão longe como a construção de uma camada de persistência de objetos feitos genéricos com uma organização puro.

Você também vai aprender os conceitos básicos de gerenciamento de sessão. Voltaremos ao tema interessante de conversas (capítulo 5) e mostrar exemplos práticos de várias maneiras as conversas podem ser implementado com NHibernate.

Este capítulo termina com uma discussão sobre aplicações distribuídas. Ele irá explicar como fazer um Aplicação NHibernate participar para uma transação distribuída.

10.1 Projetando a camada de persistência

NHibernate se destina a ser usado em praticamente qualquer cenário arquitectónico imagináveis (desde que o aplicação é baseada em. NET). Pode ser executado em uma aplicação ASP.NET, ou em um Windows / Console aplicação. Pode até correr dentro de um Serviço de Web / Windows.

Estes ambientes são muito semelhantes, tanto quanto NHibernate está em causa, apenas poucas mudanças são necessários para uma aplicação NHibernate porta de um ambiente para outro, desde que a aplicação está corretamente em camadas.

Nós não esperamos que o design do aplicativo para coincidir exatamente com o cenário que show, e nós não espero que você a integrar NHibernate usando exatamente o código que usamos. Em vez disso, vamos demonstrar alguns padrões comuns e permitem adaptá-las a seus próprios gostos. Por esta razão, nossos exemplos são planícies C #, utilizando qualquer frameworks de terceiros.

Enfatizamos a importância da estratificação aplicação disciplinada no capítulo 1. Camadas ajuda a alcançar a separação de interesses, tornando o código mais legível por código de agrupamento que faz coisas semelhantes.

Por outro lado, camadas tem um preço: Cada camada extra aumenta a quantidade de código necessário para implementar um simples pedaço de código de funcionalidade e mais torna-se mais a funcionalidade difícil de mudar.

Não vamos tentar formar qualquer conclusão sobre o número correto de camadas de usar (e certamente não sobre o que essas camadas deve ser) uma vez que o "melhor" design varia de aplicação para aplicação e uma discussão completa de arquitetura de aplicação é bem fora do escopo deste livro. Nós apenas observar que, em nossa opinião, uma camada só deve existir se for necessário, uma vez que aumenta a complexidade

e os custos do desenvolvimento. No entanto, nós concordamos que uma camada de persistência dedicado é um sensível

escolha para a maioria das aplicações e que código de persistência relacionados não deve ser misturado com a lógica de negócios ou apresentação.

Nesta seção, vamos mostrar como separar NHibernate relacionados com o código de seu negócio e camadas de apresentação em um aplicativo de console. Como dissemos antes, será fácil voltar a usar esta camada de persistência em outra aplicação (Web ou Windows).

Precisamos de um simples caso de uso da aplicação CaveatEmptor para demonstrar essas idéias: Quando um usuário coloca um lance em um item, CaveatEmptor deve realizar as seguintes tarefas, tudo em um único pedido:

- 1 Verifique se o valor digitado pelo usuário for maior que o lance máximo existente para o item.
- 2 Verifique se o leilão ainda não terminou.
- 3 Criar um novo lance para o item.

Se qualquer uma das verificações falhar, o usuário deve ser informado do motivo da falha, se ambos os cheques são

bem sucedida, o usuário deve ser informado de que o novo lance foi feito. Estas verificações são os nossos negócios

regras. Se ocorrer uma falha durante o acesso ao banco de dados, o usuário deve ser informado de que o

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.oreilly.com/catalog/nhibernate/discuss/> (uma nova página de infra-estrutura).

Vamos ver como podemos implementar isso com uma simples aplicação.

10.1.1 Implementação de uma camada de persistência simples

No "Olá Mundo" aplicação do capítulo 2, a classe Persister representou sozinho a persistência camada: Ele internamente inicializada a fábrica de sessão e abriu as sessões, conforme necessário. Este projeto simplista

não pode trabalhar para a aplicação de grande porte. Você teria muitos métodos para executar operações CRUD em todas as entidades.

Nesta seção, vamos dividir a camada de persistência em um número de classes, cada uma responsável de um preocupação específica.

Primeiro, precisamos de um caminho para a nossa aplicação para obter novas ISession instâncias. Vamos escrever uma simples

ajudante (Ou utilidade) da classe para lidar com a configuração e ISessionFactory inicialização (ver capítulo 3) e facilitar o acesso a novos ISessions. O código completo para esta classe é mostrada na listagem 10.1.

Listagem 10.1 Uma classe auxiliar simples NHibernate

```
public class NHibernateHelper
{
    public static readonly ISessionFactory SessionFactory;
    static NHibernateHelper () {
        try {
            Cfg configuração Configuração = new ();
            . SessionFactory = cfg.Configure () BuildSessionFactory ();
        } Catch (Exception ex) {
            Console.Error.WriteLine (ex);
            throw new Exception ("inicialização NHibernate falhou",
                ex);
        }
    }

    openSession ISession public static () {
        sessionFactory.openSession return ();
    }
}
```

O ISessionFactory está vinculado a um estático (E readonly) Variável # 1. Todos os nossos tópicos pode compartilhar este uma constante, porque o ISessionFactory implementação é thread-safe. Esta fábrica da sessão é criado em um construtor estático # 2. Esse construtor é executado no primeiro acesso a esta classe auxiliar. O processo de construção do ISessionFactory a partir de um Configuração # 3 é o mesmo de sempre. Nós capturar e registrar a exceção # 4; Claro, você deve usar o seu mecanismo de registro próprio, em vez de Console.Error. Nossa classe utilitário tem apenas um método público, um método de fábrica para os novos ISessions # 5.

É um método conveniente para encurtar o código necessário para o uso mais comum desta classe: Abrindo novas sessões.

Esta implementação (muito trivial) armazena os ISessionFactory em uma variável estática.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note que este projeto é totalmente segura aglomerado. O ISessionFactory implementação é essencialmente apátrida (que não mantém estado relativo às transações em execução), exceto para o segundo nível

cache. É da responsabilidade do provedor de cache para manter a coerência de cache em um cluster. Assim, você pode seguramente ter tantos reais ISessionFactory instâncias como você gosta (na prática, como você quiser poucos possível, uma vez que o ISessionFactory consome recursos significativos e é caro inicializar).

Agora que já resolveu o problema de onde colocar o ISessionFactory exemplo (a FAQ), que continuar com a nossa implementação de caso de uso.

Realizar todas as operações dentro do mesmo método

Nesta seção, vamos escrever o código que implementa o "lugar bid" caso de uso em um PlaceBid () método (ver listagem 10.2). Estamos assumindo algum tipo de estrutura, e nós não mostram como ler entradas do usuário ou como encaminhar os resultados para a camada de apresentação. (Note que nós não consideramos este primeiro implementação para ser um bom um: nós fazer melhorias substanciais depois.)

Listagem 10.2 Implementação de um simples caso de uso em um método

```
PlaceBid public void (long itemId, userId longa, dupla bidAmount) {  
    try {  
        usando (sessão ISession NHibernateHelper.OpenSession = ()) | 1  
        usando (session.BeginTransaction ()) {  
  
            / / Carregar (e bloqueio) Item solicitado | 2  
            Item item = Session.load <Item> (itemId, LockMode.Upgrade);  
  
            / / Check leilão ainda válido | 3  
            if (item.EndDate <DateTime.Now) {  
                throw new BusinessException ("Leilão já terminou.");  
  
            / / Check quantidade de Licitação | 4  
            IQuery q =  
                session.CreateQuery (@ "select max (b.Amount)  
                    de Licitação, onde b = b.Item: item ");  
            q.SetEntity ("item", item);  
            duplo = maxBidAmount (double) q.UniqueResult ();  
            if (maxBidAmount > bidAmount) {  
                throw new BusinessException ("Lance muito baixo.");  
  
            / / Adicionar novo lance ao Item | 5  
            Licitante user = Session.load <User> (userId);  
            Licitação newBid Bid = new (bidAmount, item licitante);  
            item.AddBid (newBid);  
  
            session.Transaction.Commit (); | 6  
            / / Operação conseguiu  
  
        } } Catch (HibernateException ex) {  
            throw new InfrastructureException ( | 7  
                "Erro ao acessar o banco de dados", ex);  
        }  
    }  
}
```

Primeiro, temos uma nova ISession usando a nossa classe utilitário # 1 e depois iniciar uma transação de banco de dados. Estes sessão e transação será fechada pelo usando () declaração. Se não confirmar a transação (Ou, se este cometer falha), a transação será revertida.

Nós carregamos as Item do banco de dados # 2, usando o seu valor identificador, a obtenção de um bloqueio pessimista (isto

impede duas propostas simultâneas para o mesmo item).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Se a data final do leilão seja anterior à data atual # 3, que lançar uma exceção para que o camada de persistência exibe uma mensagem de erro. Normalmente você vai querer o tratamento de erros mais sofisticados para essa exceção, com uma mensagem de erro qualificado.

Usando uma consulta HQL, vamos verificar se há uma maior oferta para o item atual no banco de dados # 4. Se houver é, exibimos uma mensagem de erro. Se todas as verificações forem bem sucedidos, nós colocamos o novo lance, adicionando-o ao item # 5. Não temos para salvá-lo manualmente, ele será salvo com persistência transitiva (em cascata a partir do Item para Oferta). Cometendo o flushes transação o estado atual do ISession ao banco de dados # 6.

Este try-catch bloco # 7 é responsável pela exceções lançadas quando reverter a transação ou encerramento da sessão, que é envolto de abstrair detalhes NHibernate.

Esta implementação do método PlaceBid () não assume a existência de qualquer outra classe. Ele faz o trabalho da camada de persistência (usando NHibernate) e da camada de negócios.

Baseado no que você aprendeu no capítulo 9, é óbvio que algumas verificação feita neste método deve ser feito pelo modelo de domínio.

Criando um modelo de domínio "inteligentes"

Atualmente, nosso PlaceBid () método contém alguma lógica de negócios. Vamos movê-lo para seu devido lugar: o Item entidade.

Primeiro, vamos adicionar o novo método PlaceBid () ao Item classe:

```
PlaceBid licitação pública (Usuário licitante, duplo bidAmount, maxBidAmount double) {  
    if (this.EndDate < DateTime.Now)  
        throw new BusinessException ("Leilão já terminou.");  
  
    if (maxBidAmount > bidAmount) {  
        throw new BusinessException ("Lance muito baixo.");  
  
    / / Cria novo lance  
    Licitação newBid = novo lance (bidAmount, este licitante);  
  
    / Local / lance para este item  
    this.AddBid (newBid);  
    retorno newBid;  
}
```

Este código impõe regras de negócios que condicionam o estado de nossos objetos de negócios, mas não execute de acesso a dados de código. A motivação é encapsular lógica de negócios em classes do modelo de domínio sem qualquer dependência de acesso a dados persistentes.

Você pode ter descoberto que este método de Item exige que o valor do lance mais alto atual. É óbvio que o modelo de domínio não devem usar a camada de persistência para consultar este valor e seria ineficientes para iterar a sua coleção de lances procurando por ele. Então nós preferimos pedir a camada superior para fornecer este valor.

Agora, nós simplificar nossa PlaceBid () método para o seguinte:

```
PlaceBid public void (long itemId, userId longa, dupla bidAmount) {  
    try {  
        usando (sessão ISession NHibernateHelper.OpenSession = ())  
        usando (session.BeginTransaction ()) {  
  
            / / Carregar (e bloqueio) solicitou item  
            Item item = Session.load <Item> (itemId, LockMode.Upgrade);  
  
            / / Recuperar o valor maior lance  
            IQuery q =
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        session.CreateQuery (@ "select max (b.Amount)
                                de Licitação, onde b = b.Item: item ");
        q.SetEntity ("item", item);
        duplo = maxBidAmount (double) q.UniqueResult ();

        / / Recuperar o licitante e coloque o lance
        Licitante user = Session.load <User> (userId);
        item.PlaceBid (licitante, bidAmount, maxBidAmount);

        session.Transaction.Commit ();
        / / Operação conseguiu

    } Catch (HibernateException ex) {
        throw new InfrastructureException (
            "Erro ao acessar o banco de dados", ex);
    }

}

```

A lógica de negócios para a colocação de um lance agora é encapsulada no modelo de domínio. O restante do código pertence à camada de persistência ea camada de negócios.

É hora de separar claramente essas camadas. Como introduzido no chapter2, podemos criar um Persister classe. Esta classe deve conter um método para carregar itens e os usuários e outro para carregar o lance mais alto montante. Seria também contém métodos para carregar e salvar todas as outras entidades do nosso modelo de domínio.

O resultado seria uma classe muito grande com muitas redundâncias.

Existem muitos padrões para resolver este problema. Vamos descobrir um dos mais Populares.

Apresentando os dados padrão Access Object Código de acesso mistura dados (responsabilidade da camada de persistência), com lógica de controle (parte do camada de negócios) violam nossa ênfase na separação de interesses. Para todos, mas a mais simples aplicações, faz sentido esconder NHibernate API chamadas atrás de uma fachada com semântica de alto nível de negócios. Há mais do que uma maneira de projetar essa fachada, alguns pequenos aplicativos podem usar uma única classe para todas as operações de persistência; alguns poderiam uma classe para cada operação, mas nós preferimos o padrão DAO.

A DAO define uma interface para operações de persistência (CRUD e métodos finder) relativa a um entidade persistente particular. Ele aconselha a código de grupo que se relaciona com a persistência dessa entidade.

Outro nome comum para este padrão é Gateway (pensei que eles têm um significado ligeiramente diferente).

Vamos criar um ItemDAO classe, o que acabará por implementar todo o código relacionado com a persistência ItemS. Por enquanto, ele contém apenas os FindById () método, juntamente com GetMaxBidAmount () e um método para guardar itens. O código completo da implementação DAO é mostrado em 10,3 listagem.

A listagem de 10,3 DAO simples abstrair operações de itens relacionados com a persistência

```

public class {ItemDAO

    FindById item public static (long id) {
        usando (sessão ISession NHibernateHelper.OpenSession = ())
            retorno Session.load <Item> (id);
    }

    public static dupla GetMaxBidAmount (itemId tempo) {
        query string = @ "select max (b.Amount)
                            de Licitação, onde b = b.Item: item ";
        usando (sessão ISession NHibernateHelper.OpenSession = ()) {
            IQuery session.CreateQuery q = (query);
            q.SetInt64 ("itemId", itemId);
            retorno (double) q.UniqueResult ();
        }
    }
}

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    item public static MakePersistent (entidade Item) {
        usando (sessão ISession NHibernateHelper.OpenSession = ())
            session.SaveOrUpdate (entidade);
        retorno entidade;
    }
}

```

| 3

Essa classe fornece dois métodos estáticos para executar as operações necessárias para a nossa PlaceBid () método.

O FindById () cargas método # 1 itens (note que o bloqueio pessimista não está disponível). É possível recuperar a quantidade maior lance usando o GetMaxBidAmount () # método 2 e os MakePersistent () # método 3 pode ser usado para guardar itens.

Se GetMaxBidAmount () pertence a ItemDAO ou um BidDAO talvez seja uma questão de gosto, mas desde o argumento é uma Item identificador, parece pertencem naturalmente aqui.

Precisamos também de um UserDao com um FindUserById () método. Você deve ser capaz de descobrir como implementá-lo (basta substituir Item por Usuário na listagem anterior).

Nosso PlaceBid () método é mais limpo:

```

PlaceBid public void (long itemId, userId longa, dupla bidAmount) {

    try {
        Item item = ItemDAO.FindById (itemId);
        dupla maxBidAmount = ItemDAO.GetMaxBidAmount (itemId);
        Usuário licitante = UserDao.FindById (userId);
        item.PlaceBid (licitante, bidAmount, maxBidAmount);

        ItemDAO.MakePersistent (item);
    }
    } Catch (HibernateException ex) {
        throw new InfrastructureException (
            "Erro ao acessar o banco de dados", ex);
    }
}

```

Observe como muito mais auto-documentar esse código é do que a nossa primeira implementação. Alguém que não sabe nada sobre NHibernate ainda pode compreender imediatamente o que este método faz, sem a necessidade de comentários de código.

Também conseguiu uma separação clara de preocupação. Você pode estar satisfeito por esta implementação. No entanto, tem vários inconvenientes.

Primeiro, faz o impossível o uso de persistência transparente. É por isso que precisamos salvar explicitamente o item no final. Além disso, abre-se quatro sessões em que uma única seria suficiente. Por fim, o implementação do DAOs tem um alto nível de redundância para as operações básicas CRUD.

Estes problemas podem ser resolvidos por abstrair as operações de base comuns e por descobrir uma maneira de fazer essas partes DAOs mesma sessão.

Vamos pular para a solução certa.

10.1.2 Implementação de uma camada de persistência genéricos

Nós aprendemos na seção anterior que, embora seja fácil de implementar um DAO simples, há uma número de questões-chave que requerem uma solução mais inteligente. Deve permitir que todos os DAOs para compartilhar o mesmo sessão e deve minimizar a quantidade de redundância.

Vamos descobrir uma grande solução para a questão primeiro. É um novo recurso do NHibernate 1.2.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

`ISessionFactory.GetCurrentSession` usando ()

A idéia por trás deste recurso é que, para uma ação específica, que geralmente precisam de uma única sessão. Porque

Nesta sessão pode ser reutilizado para todas as operações (mesmo que eles não estão relacionadas), é lógico fazer isso de sessão disponíveis para toda a aplicação (isto é, a sua camada de persistência).

Seu primeiro impulso poderia ser a criação de uma sessão de estática como a fábrica de sessão definido na listagem 11.1.

No entanto, ele não vai funcionar para aplicações ASP.NET, porque cada sessão da Web deve ter o seu próprio NHibernate sessão (usando uma sessão estática resultaria em uma única sessão para a Web inteira aplicação).

Também é possível abrir uma sessão e enviá-lo para cada DAO. Neste caso, DAOs não terá mais métodos estáticos. Você iria instanciar esses DAOs e fornecer a sessão como um parâmetro em sua construtores. Esta solução pode funcionar. No entanto, é muito tedioso, você vai ter que passar o Sessão NHibernate em todos os lugares que podem ser necessários.

Em vez de resolver o problema sozinho, você pode alavancar um novo recurso do NHibernate 1.2. É expostos pelo método `ISessionFactory.GetCurrentSession` (). Este método retorna a sessão instância associada com corrente contexto de persistência, semelhante à noção ASP.NET de um pedido HTTP contexto. Qualquer componente chamado no mesmo contexto irão compartilhar a mesma sessão.

Ao usar esse recurso, o contexto específico da sua aplicação é abstrata. Portanto, sua persistência camada vai funcionar se o contexto é definido por uma Web de contexto do Windows.

O primeiro passo para ativar esse recurso é para definir o contexto. Isso é feito usando a configuração propriedade `current_session_context_class`. Por exemplo:

```
<property name="current_session_context_class">
    teia
</ Property>
```

Este exemplo define o contexto para `teia`, Que é o nome abreviado de uma implementação incluídos no NHibernate que usa `HttpContext` para acompanhar a sessão atual. Portanto, é apropriado para ASP.NET aplicações.

NHibernate 1.2.1 vem com uma série de built-in implementações contexto atual sessão:

Tabela 10.1 NHibernate built-in implementações contexto atual sessão

Nome curto	Descrição
Managed_web	Neste contexto foi o único disponível no NHibernate 1.2.0. No entanto, é agora obsoleta. Você deve usar <code>teia</code> em seu lugar.
Chamada	Neste contexto usa o <code>HttpContext</code> API para armazenar a sessão atual. Note-se que, embora ele funcione em qualquer tipo de aplicação, não é recomendado para ASP .NET 2,0 aplicações.
thread_static	Ao usar este contexto, as sessões são armazenados em um campo estático marcados com <code>[ThreadStaticAttribute]</code> . Portanto, cada segmento tem sua própria sessão.
Teia	Neste contexto usa o <code>HttpContext</code> API para armazenar a sessão atual. É recomendado para aplicações Web (e só funciona com eles).

É obviamente possível implementar o seu próprio contexto. Você apenas tem que escrever uma classe que implementa a interface de extensão: `NHibernate.Context.ICurrentSessionContext` e defini-lo, no mencionado propriedade. Para mais detalhes, veja a sua documentação e as implementações disponíveis no namespace `NHibernate.Context`.

Dependendo da implementação do contexto, você pode ter um trabalho adicional a fazer. Para exemplo, esses contextos não tomar cuidado de abrir e fechar a sessão. Você tem que fazê-lo sozinho e vinculá-lo ao contexto (usando a classe `CurrentSessionContext`).

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Vamos ver como usar este recurso. Primeiro, devemos adicionar o seguinte método para a classe NHibernateHelper:

```
getCurrentSession ISession public static () {
    retorno SessionFactory.getCurrentSession ();
}
```

Aqui está o que a classe ItemDAO parece agora:

```
public class {ItemDAO

    FindById item public static (long id) {
        retorno NHibernateHelper.GetCurrentSession () Load <Item> (id).;
    }

    public static dupla GetMaxBidAmount (itemId tempo) {

        query string = @ "select max (b.Amount)
                        de Licitação, onde b = b.Item: item ";
        IQuery q = NHibernateHelper.GetCurrentSession () CreateQuery (query).;
        q.SetInt64 ("itemId", itemId);
        retorno (double) q.UniqueResult ();

    }

    item public static MakePersistent (entidade Item) {
        . NHibernateHelper.GetCurrentSession () saveOrUpdate (entidade);
        retorno entidade;
    }

}
```

O DAO não é mais responsável da abertura da sessão NHibernate. Isto é feito em um nível superior. No nosso exemplo, isso pode ser feito no nosso PlaceBid () método.

Listagem 10.4 fornece a implementação deste método.

Listagem 10.4 gerenciamento de sessão simples usando a API atual sessão

```
usando NHibernate.Context;

PlaceBid public void (long itemId, userId longa, dupla bidAmount) {

    try {
        usando (sessão ISession NHibernateHelper.OpenSession = ())
        usando (session.BeginTransaction ()) {

            / / Anexar a sessão para o contexto | 1
            CurrentSessionContext.Bind (sessão);

            / / Execute a lógica como antes | 2
            Item item = ItemDAO.FindByIdAndLock (itemId);
            dupla maxBidAmount = ItemDAO.GetMaxBidAmount (itemId);
            Usuário licitante = UserDao.GetById (userId);
            item.PlaceBid (licitante, bidAmount, maxBidAmount);

            session.Transaction.Commit ();

        }
        } Catch (HibernateException ex) {
            throw new InfrastructureException (
                "Erro ao acessar o banco de dados", ex);
        }
        finally {
            CurrentSessionContext.Unbind (NHibernateHelper.SessionFactory); | 3
        }
    }
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Uma vez que a sessão é aberta, podemos anexá-lo ao contexto atual de modo que ele está disponível para qualquer objeto executados neste contexto # 1. Depois disso, podemos executar a lógica como antes # 2. Note-se que pessimistas bloqueio pode ser usado agora. No final da operação, é preciso separar a sessão (fechado) a partir do contexto atual # 3. Com essa implementação, DAOs pode acessar a mesma sessão de uma forma limpa e transparente. É tempo para resolver o outro problema na execução de uma camada de persistência: Minimizar a redundância.

Projetando DAOs usando generics

Sempre que você encontrar-se a repetição de um código semelhante várias vezes, é tempo para pensar herança e genéricos. Note que esta seção pressupõe que você tenha uma boa compreensão do .NET 2.0 genéricos. Em . o caso que você ainda está usando Framework 1.1, é possível adaptar a seguinte idéia, no entanto, o resultado não será tão limpo.

Como você viu na implementação do método `FindById()` para as classes `ItemDAO` e `UserDAO`, Apenas o nome da entidade para mudanças operações básicas CRUD. Portanto, é possível a utilização de genéricos para abstrata esta operação.

Aqui está como o novo DAOs pode se parece com:

```
public abstract class GenericNHibernateDAO <T, ID> {  
    T público FindById (ID id) {  
        try {  
            . NHibernateHelper.GetCurrentSession return () Load <T> (id);  
        }  
        catch (HibernateException ex) {  
            throw new Exceptions.InfrastructureException (ex);  
        }  
    }  
    ...  
}  
public class UserDAO: GenericNHibernateDAO <User, long> {  
}
```

A classe `GenericNHibernateDAO` só precisa dos tipos de entidade e seu identificador ID para implementar o método `FindById()`. Depois disso, a implementação da classe `UserDAO`, Significa simplesmente herdando `GenericNHibernateDAO` e fornecer estes tipos.

Muitos outros métodos podem ser implementados como essa. Antes de preencher a classe `GenericNHibernateDAO` com eles, vamos dar um passo atrás e pensar sobre o design final que queremos.

Quanto à camada de negócios está em causa, a camada de persistência deve oferecer um conjunto de interfaces para executar todas as operações que são necessárias. Isto significa que a implementação sublinhando não matéria e pode ser alterado, desde que as interfaces não mudam.

Em nosso projeto, teremos uma `GenericDAO` interface com operações comuns a todas as entidades e DAO interfaces, herança da `GenericDAO` interface, para cada entidade. Estas interfaces terão todos implementações usando NHibernate.

Figura 10.1 ilustra este projeto:

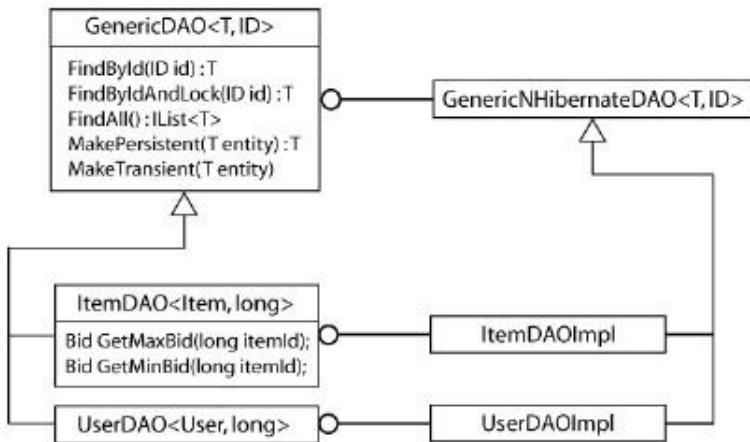


Figura 10.1 Interfaces genéricas DAO com separados implementação NHibernate

Algumas interfaces não podem ter qualquer método, mas ainda é importante para criá-los porque eles são rígidez interfaces e que pode ser estendido no futuro.

Finalmente, vamos usar o padrão de fábrica abstrata como uma fachada para fornecer as implementações do interfaces. Isto significa que as classes NHibernate será completamente escondido de outras camadas. Vai até ser possível mudar o mecanismo de persistência, mesmo em tempo de execução.

Teoria suficiente. Vamos dar uma olhada no GenericDAO interface:

```

public interface GenericDAO <T, ID> {
    T FindById (ID id);
    T FindByIdAndLock (ID id);

    IList <T> FindAll ();

    T MakePersistent (entidade T);
    vazio MakeTransient (entidade T);
}

```

Esta interface define métodos para carregar (o Encontrar ...) métodos), salvar (usando MakePersistent ()) E delete (usando MakeTransient ()) Entidades.

Por MakePersistent () e MakeTransient () em vez de Salvar () e Delete ()?

É mais simples para explicar as operações de persistência usando verbos como salvar ou excluir. No entanto, NHibernate é uma Estado-oriented quadro. Essa noção foi introduzida no capítulo 5, seção 5.1. Por exemplo, quando exclusão uma entidade, o que realmente acontece é que esta entidade torna-se passageira. Sua linha no banco de dados acabará por ser excluído, mas o entidade não deixará de existir (e pode até mesmo ser persistiu novamente). Como esses métodos são criados para a camada de negócios, seus nomes devem refletir o que acontece nesse nível.

As interfaces de herança da GenericDAO interface será parecido com este:

```

public interface ItemDAO: GenericDAO <Item, long> {

    Licitacao GetMaxBid (itemId longo
                           prazo);
    Licitacao GetMinBid (itemId longo
                           prazo);
}

```

Aqui, evitamos a definição de um método muito específicas, como GetMaxBidAmount (). Agora, vamos implementar essas

interfaces. Primeiro, o GenericNHibernateDAO:

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public abstract class GenericNHibernateDAO <T, ID>: GenericDAO <T, ID> {
    sessão ISession privado;

    Sessão pública ISession {
        get {
            if (sessão == null)
                sessão NHibernateHelper.GetCurrentSession = ();
            retorno da sessão;
        }
        conjunto {
            sessão = valor;
        }
    }
    T público FindById (ID id) {
        retorno Session.load <T> (id);
    }
    T público FindByIdAndLock (ID id) {
        retorno Session.load <T> (id, LockMode.Upgrade);
    }
    público IList <T> FindAll () {
        retorno session.createCriteria (typeof (T)) List <T> ();
    }
    T público MakePersistent (entidade T) {
        Session.SaveOrUpdate (entidade);
        retorno entidade;
    }
    MakeTransient public void (T entidade) {
        Session.Delete (entidade);
    }
}

```

Nesta implementação, nós adicionamos a propriedade `Sessão` de modo que a DAOs pode trabalhar sem precisar de um sessão ligado ao contexto atual. No entanto, neste caso, você deve fornecer manualmente esta sessão. Você pode escrever outra classe na camada de persistência para cuidar disso.

A implementação de outras classes é simples. Aqui está a implementação do ItemDAO interface:

```

public class ItemDAOImpl: GenericNHibernateDAO <Model.Item, long>, ItemDAO {
    public virtual Model.Bid GetMinBid (itemId tempo) {
        IQuery q = Session.GetNamedQuery ("MinBid");
        q.SetInt64 ("itemId", itemId);
        retorno q.UniqueResult <Model.Bid> ();
    }
    public virtual Model.Bid GetMaxBid (itemId tempo) {
        IQuery q = Session.GetNamedQuery ("MaxBid");
        q.SetInt64 ("itemId", itemId);
        retorno q.UniqueResult <Model.Bid> ();
    }
}

```

Embora tenha nada a ver com este projeto, decidimos seguir outro boas práticas que é a utilização de consultas nomeadas. Note que você deve envolver todos estes métodos em um `try / catch` declaração no caso de um exceção é lançada:

```

try {
    ...
} catch (HibernateException ex) {
    throw new Exceptions.InfrastructureException (ex);
}

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

O uso deste camada de persistência novo não é muito diferente do que é feito na listagem 11.4. Os principais diferença é que esses DAOs deve ser instanciado. Nós vamos adicionar outra interface para cuidar de que preocupação sem a introdução de uma dependência para a implementação NHibernate:

```
public abstract class DAOFactory
{
    public abstract UserDao GetUserDAO ();
    public abstract ItemDao GetItemDAO ();
}
```

Sua implementação é simples:

```
public class NHibernateDAOFactory : DAOFactory
{
    public override UserDao GetUserDAO ()
    {
        return new UserDaoImpl ();
    }
    public override ItemDao GetItemDAO ()
    {
        return new ItemDAOImpl ();
    }
}
```

O último passo é instanciar esta classe na inicialização da aplicação:

```
DAOFactory daoFactory = new NHibernateDAOFactory ();
```

Tudo está pronto para o novo PlaceBid () método. Aqui é a parte interessante:

```
ItemDAO itemDAO = daoFactory.GetItemDAO ();
Item item = itemDAO.FindByIdAndLock (itemId);
dupla maxBidAmount = itemDAO.GetMaxBid (itemId) Valor.Valor;
Licitante user = daoFactory.GetUserDAO () FindById (userId);
item.PlaceBid (licitante, bidAmount, maxBidAmount);
```

O restante deste método permanece como na listagem 11.4. Curiosamente, este código representa exatamente o que nós

gostaria que o nosso PlaceBid () método para se parecer com (o resto é apenas encanamento).

Mais importante, se estamos de acordo que a gestão da sessão NHibernate é uma preocupação camada de persistência, ele

não deve aparecer neste método. Nós vamos fazer um refactoring última para cuidar desta questão.
Gerenciamento de sessão para aplicativos da Web

Ao processar um pedido complexo, muitos métodos (e classes) da camada de negócios pode se envolver.
Se esses métodos segue a abordagem de nossa implementação anterior do PlaceBid () método, ele irá resultar em abertura de muitas sessões separadas.

De lado o problema de desempenho óbvio, isso significa que a lógica do negócio terá uma duração de banco de dados de muitos transações, o que implica que, se uma segunda operação falhar, os anteriores não serão revertidas e banco de dados será deixado em um estado inconsistente.

Outra questão é que, depois de executar a lógica de negócios, ou mesmo entre duas chamadas para a camada de negócios,
as entidades manipuladas são destacados. Isto significa que o carregamento lento é desativado. Portanto, uma outra camada,
como a camada de apresentação, não pode carregar transparente preguiçoso as coleções dessas entidades.

Por que não pode NHibernate abrir uma nova conexão (ou sessão) se tiver que lazy-load associações?

Primeiro, pensamos que é uma solução melhor para totalmente inicializar todos os necessários objetos para um caso de uso específico utilizando a busca ansiosa (esta abordagem é menos vulneráveis ao problema n seleciona +1). Além disso, a abertura novas conexões de banco de dados (e ad hoc transações de banco de dados!) implicitamente e de forma transparente para o desenvolvedor expõe a aplicação.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=293>

terminar a operação ad hoc, depois de cada associação preguiçoso é carregado?

Nós preferimos fortemente as transações sejam clara e explicitamente demarcada pelo desenvolvedor do aplicativo. Se você quiser habilitar preguiçoso

fetching para uma instância individual, você pode usar `Lock()` para anexá-lo a um

Felizmente, este problema é facilmente resolvido usando uma sessão que fica aberto para a solicitação de toda tempo de vida. Nesta seção, vamos dar o exemplo de uma solicitação da Web.

Processamento do minuciosa do pedido Web, o processamento da página web, a computação da camada de persistência e qualquer outro componente, a mesma sessão será usado. Esta sessão será fechada no final deste processo. Isto significa que qualquer associação não inicializada ou cobrança será inicializado com êxito quando acessado em qualquer ponto do processo.

No entanto, não ficar com preguiça e deixe carregar dados NHibernate on-demand, que acabará por matar o desempenho de sua aplicação. Você deve sempre ansiosamente carregar os dados que você sabe que você vai necessidade. Para mais detalhes, leia a seção 7.7.1 do capítulo 7.

A solução para nosso problema é para abrir e anexar uma sessão NHibernate no início da Web pedido. ASP.NET permite-nos implementar a interface `IHttpModule`, a fim de executar algum código em início e no final de um pedido web. Vamos aproveitar este recurso.

Listagem módulo Web 10,5 gerenciamento de sessões NHibernate

```
using System;
using System.Web;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Criterion;
using NHibernateHelper;
using NHibernateCurrentSessionWebModule;

public class NHibernateCurrentSessionWebModule : IHttpModule {
    public void Init (HttpApplication context) {
        context.BeginRequest += new EventHandler (Application_BeginRequest);
        context.EndRequest += new EventHandler (Application_EndRequest);
    }
    public void Dispose () {
    }
    void Application_BeginRequest (object sender, EventArgs e) {
        ISession sessão = NHibernateHelper.OpenSession ();
        sessão.BeginTransaction ();
        CurrentSessionContext.Bind (sessão);
    }
    void Application_EndRequest (object sender, EventArgs e) {
        ISession sessão = CurrentSessionContext.Unbind (
            NHibernateHelper.SessionFactory);
        if (sessão != null)
            try {
                sessão.Transaction.Commit ();
            }
            catch (Exception ex) {
                sessão.Transaction.Rollback ();
                Server.Transfer ("...", true); // Erro de página
            }
            finally {
                sessão.Close ();
            }
    }
}
```

No início de uma solicitação de # 1, abrimos e anexar uma sessão e no final # 2, destacamos e fechar a sessão. Nós nos comprometemos também as mudanças que acontecem vida completa da solicitação.

Por favor, postar comentários ou correções para o fórum on-line em Autor <http://www.manning-sandbox.com/forum.ispa?forumID=295>

O seguinte código deve ser adicionado ao Web.config file:

```
<configuration>
  <system.web>
    <httpModules>
      <add name="NHibernateCurrentSessionWebModule"
           type="NHibernate.NHibernateCurrentSessionWebModule" />
    </httpModules>
  </system.web>
</configuration>
```

Este código é necessário para registrar nosso módulo web para que o ASP.NET usa-lo.

Agora, o nosso PlaceBid () método é exatamente como queríamos que fosse:

```
PlaceBid public void (long itemId, userId longa, dupla bidAmount) {
  try {
    ItemDAO itemDAO daoFactory.GetItemDAO = ();
    Item item = itemDAO.FindByIdAndLock (itemId);
    dupla maxBidAmount = itemDAO.GetMaxBid (itemId) Valor.Valor;
    . Licitante user = daoFactory.GetUserDAO () FindById (userId);
    item.PlaceBid (licitante, bidAmount, maxBidAmount);

  } Catch (Exception ex) {
    throw new BusinessException ("Efetuar o lance falhou.", ex);
  }
}
```

Nós temos uma sessão que vive desde que a solicitação da Web. Pode ser suficiente ou não pode: se o para executar a operação requer solicitações muitos Web para ser concluída?

10.2 conversas Implementação

Agora que temos implementado nossa camada de persistência, precisamos discutir mais sobre seu uso. Quando usando uma estrutura de comando-oriented (por exemplo ADO.NET), cada chamada API destina-se a recuperar ou os dados de alteração: Adicionar / atualizar / excluir linhas em um banco de dados. No entanto, NHibernate é o estado-oriented: cada chamada API se destina a alterar o estado de uma entidade (como ilustrado pela figura 5.1 do capítulo 5).

Alterar o estado de uma entidade pode levar à execução de um comando SQL imediatamente, quando rubor da sessão ou nunca. A diferença entre comando e estado é muito importante quando execução de operações que se estendem por muitos pedidos de usuários. Essas operações são chamadas conversas.

Nós discutimos a noção de conversa no capítulo 5, secção 5.2, "Trabalhando com as conversas." Nós Também discutimos como NHibernate ajuda a detectar conflitos entre conversas simultâneas usando gerido versionamento. Não discutimos como as conversas são utilizados em aplicações NHibernate, então nós agora voltar a este assunto essencial.

Há três maneiras de implementar as conversas em um aplicativo que usa NHibernate: usando um longa sessão, utilização objetos individual, e fazendo-o maneira mais difícil. Vamos começar com a maneira dura, que irá ajudar a compreender os benefícios das duas outras maneiras. Primeiro, precisamos de um caso de uso para ilustrar essas.

10.2.1 Aprovação de um novo leilão

Nossos leilão tem um ciclo de aprovação. Um novo item é criado no Rascunho Estado. O usuário que criou o leilão pode colocar o item em Pendente Estado quando o usuário está satisfeito com os detalhes do item. Sistema Os administradores podem então aprovar o leilão, colocando o item na Ativo Estado e início do leilão. A qualquer momento antes do leilão for aprovado, o usuário ou qualquer administrador pode editar o item detalhes. Uma vez que o leilão for aprovado, nenhum usuário ou administrador pode editar o item. É essencial que a aprovação de administrador vê a revisão mais recente do detalhes do item antes de aprovar o leilão e que um leilão não pode ser aprovada duas vezes. Figura 10.2 mostra o ciclo de aprovação item.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

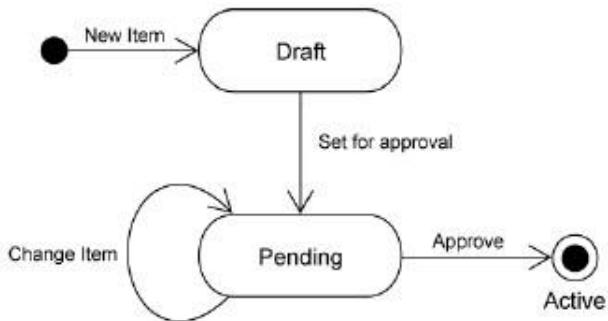


Figura 10.2 gráfico Estado do ciclo de aprovação item em CaveatEmptor

A conversa é a aprovação do leilão, que abrange duas solicitações do usuário. Primeiro, o administrador seleciona um pendente item para ver os seus detalhes, em segundo lugar, o administrador aprova o leilão, movendo-se o item o estado ativo. O segundo pedido deve executar uma verificação de versão para verificar se o item não foi atualizado ou aprovados desde que foi recuperado para mostrar.

Como de costume, a lógica de negócios para a aprovação de um leilão deve ser implementada pelo modelo de domínio.

Neste caso, nós adicionamos um `Aprovar()` método para o Item classe:

```

public void Aprovar (Usuário byUser) {
    if (! byUser.isAdmin)
        throw new PermissionException ("Não é um administrador.");

    if (state.equals( ItemState.Pending ))
        throw new BusinessException ("Item não pendentes.");

    estado = ItemState.Active;
    approvedBy = byUser;
    approvalDatetime = DateTime.Now;
}

```

No entanto, é o código que chama esse método que está interessado polegadas

São conversas realmente transações?

A maioria dos livros definir transação em termos das propriedades ACID: atomicidade, consistência, isolamento e durabilidade. É uma conversa realmente uma transação por essa definição? Consistência e durabilidade não parecem ser um problema, mas que sobre a atomicidade e isolamento? Nossa exemplo é tanto atômica e isolada, uma vez que todas as operações de atualização ocorrem no ciclo de solicitação / resposta anterior (isto é, o último banco de dados transação). No entanto, a nossa definição de uma conversa permite operações de atualização de ocorrer em qualquer ciclo de solicitação / resposta. Se um conversa executa uma operação de atualização em qualquer, mas a final transação, não é atômica e não podem sequer ser isolado. No entanto, sentimos que a transação prazo ainda é necessário, desde sistemas com esse tipo de conversa costumam ter funcionalidade ou um processo de negócio que permite ao usuário para compensar para esta falta de atomicidade (permitindo que o usuário reverta as etapas do conversa manualmente, por exemplo).

Agora que temos o nosso caso de uso, vamos olhar para as diferentes formas que podemos implementá-lo. Vamos começar com uma abordagem não recomendamos.

10.2.2 Fazê-lo da maneira mais difícil

Da maneira mais difícil de implementar conversas é descartar todas as instâncias persistentes entre cada solicitação.

A justificativa para essa abordagem é que, uma vez que a transação é finalizada, o persistente casos não são mais a garantia de estar em um estado que é consistente com o banco de dados. Quanto mais tempo o

administrador passa a decidir se aprova ou não o leilão, maior o risco de que algum outro usuário editou os detalhes do leilão e que o Item exemplo, passou a deter dados desatualizados.

Suponha que o nosso pedido executado pela primeira vez o seguinte código para recuperar os detalhes do leilão:

```
newItem item pública (itemId tempo) {
    retorno itemDAO.FindById (itemId);
}
```

Essa linha de pensamento que nos aconselham a descartar o retornado Item depois exibi-lo, armazenando somente os

valor de identificador para uso no próximo pedido. Parece razoável que superficialmente, devemos recuperar o Item exemplo de novo no início do segundo pedido. Poderíamos, então, estar certos de que o Item realizada não dados antigos para a duração da transação segundo banco de dados.

Há um problema com esta noção: O administrador já usou os dados possivelmente obsoletas para chegar à decisão de aprovar! Recarregar a Item no segundo pedido é inútil, já que o Estado recarregado não serão utilizados para qualquer coisa-em pelo menos, não pode ser usado para decidir se o leilão deve ser aprovado, que é o importante.

A fim de garantir que os detalhes que foram vistos e aprovados pelo administrador ainda são os detalhes atuais durante a transação segundo banco de dados, devemos realizar uma explícita versão manual cheque. O código a seguir demonstra como isso poderia ser implementado pela camada de negócio:

```
ApproveAuction public void (itemId prazo,
                             itemVersion int,
                             longo adminId) {

    Item item = itemDAO.FindById (itemId);

    if (itemVersion! item.Version ==)
        throw StaleItemException new ();

    Usuário admin = userDAO.FindById (adminId);
    item.Approve (admin);
}
```

Neste caso, a verificação de versão manual não é especialmente difícil de implementar.

Somos justificados em chamar esta abordagem difícil? Em casos mais complexos que envolvem relacionamentos, é tedioso para realizar todas as verificações manualmente para todos os objetos que estão a ser atualizado. Estes versão manual verificações devem ser considerados ruído que implementar uma preocupação puramente sistêmicas não expressos na problema de negócio.

Mais importante, o trecho de código anterior contém ruídos desnecessários outros. Nós já recuperados o Item e Usuário em pedidos anteriores. É necessário recarregá-las em cada pedido? Deve-se possível simplificar nosso código de controle para o seguinte:

```
usuários) {
    item.Approve (admin);
}
```

Se o fizer, não só poupa três linhas de código, mas também é objeto indiscutivelmente mais orientado-o nosso sistema é

trabalhando principalmente com instâncias do modelo de domínio em vez de passar em torno de valores identificador.

Além disso, este código seria mais rápido, uma vez que ele salva duas SQL SELEÇÃO consultas que inutilmente reload de dados. Como podemos alcançar esta simplificação usando NHibernate?

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

10.2.3 Usando destacado objetos persistentes

Suponha que manteve o Item como uma instância separada (esta abordagem é comum em aplicações Windows). Poderíamos reutilizá-lo na transação de banco de dados segundo por re-associando-o ao novo NHibernate sessão usando Lock () ou Update (). Vamos ver o que estas duas opções parecem.

No caso de Lock (), Ajustamos o ApproveAuction () método para ficar assim:

```
ApproveAuction public void (item Item, de administração dos
usuários) {
    try {
        NHibernateHelper.GetCurrentSession ()
            . Lock (item, LockMode.NONE);

    } Catch (HibernateException ex) {
        throw new InfrastructureException (ex);
    }

    item.Approve (admin);
}
```

A chamada para ISession.Lock () re-associa o item com a nova sessão NHibernate e garante que qualquer alteração subsequente ao estado do item é propagada para o banco de dados quando a sessão é lavada (para uma discussão dos diferentes LockModes, Ver capítulo 6, secção 6.1.8, "Usando pessimista bloqueio").

Desde Item é versionado (se um mapa <versão> propriedade), NHibernate irá verificar o número da versão quando a sincronização com o banco de dados, usando o mecanismo descrito no capítulo 6, secção 6.2.1, "Usando gerida de versões. "Portanto, você não tem que usar um bloqueio pessimista, enquanto seria permitido para transações simultâneas para ler o item em questão, enquanto a rotina de aprovação é executado.

Claro, seria melhor para ocultar códigos NHibernate em um método DAO novo, então adicionamos um novo Lock () método para o ItemDAO. Isso nos permite simplificar o ApproveAuction () método para

```
ApproveAuction públicos (item Item, de administração dos usuários) {
    itemDAO.Lock (item, false) // = false não ser pessimista
    item.Approve (admin);
}
```

Alternativamente, nós poderíamos usar Update (). Para o nosso exemplo, a única diferença real é que Update () pode ser chamado depois que o estado do item foi modificado, o que seria o caso se o administrador fez alterações antes de aprovar o leilão:

```
ApproveAuction públicos (item Item, de administração dos
usuários) {
    item.Approve (admin);
    itemDAO.MakePersistent (item);
}
```

Mais uma vez, NHibernate irá executar uma verificação de versão ao atualizar o item.

É esta implementação, utilizando objetos destacados realmente mais simples do que a maneira mais difícil? Nós ainda precisamos uma chamada explícita para o ItemDAO, Então a questão é discutível. Em um exemplo mais complexo que envolve associações, veríamos mais benefícios, já que a chamada para Lock () ou Update () pode cascata para associados instâncias. Além disso, não vamos esquecer que esta implementação é mais eficiente, evitando a desnecessário SELECCIONES.

No entanto, ainda não estamos satisfeitos. Existe uma maneira de evitar a necessidade de re-associação explícita com uma nova sessão? Uma maneira seria usar a mesma sessão NHibernate para ambos os banco de dados transações, um padrão que descrevemos no capítulo 6, session-per-conversation ou longa sessão.

10.2.4 Usando o pattern session-per-conversation

A longa sessão é uma sessão NHibernate que se estende por toda uma conversa, permitindo a reutilização de persistentes casos em transações de banco de dados múltiplos. Esta abordagem evita a necessidade de re-associar instâncias destacadas criados ou recuperados em operações de banco de dados anterior.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Uma sessão contém dois tipos importantes do Estado: Possui um cache de instâncias persistentes e um ADO.NET IDbConnection. Nós já salientou a importância de não manter recursos de banco de dados aberta em várias solicitações. Portanto, a sessão precisa liberar a conexão entre os pedidos, se você pretende mantê-la aberta por mais de um pedido.

Como explicado no capítulo 5, seção 651,4, "Entendendo os modos de conexão release", NHibernate 1,2 mantém a conexão aberta desde a primeira vez que é necessário para o momento em que a transação é cometido. Portanto, confirmar a transação, no final de cada pedido é suficiente para fechar a conexão.

No 10,5 listagem, foram utilizados os teia sessão atual contexto para armazenar a sessão. Essa estratégia utiliza

HttpContext. No entanto, neste contexto, não sobreviverá a solicitação da Web. Portanto, precisamos de uma diferente solução.

Note-se que, no caso de um aplicativo do Windows, este não é um problema. O contexto da sessão atual (HttpContext por exemplo) terá a mesma duração que o aplicativo. O único trabalho que é necessário é delimitar corretamente a conversa.

Em uma aplicação ASP.NET, é possível manter a sessão NHibernate na sessão ASP.NET Estado para que fique disponível a partir de um pedido para outro. Nós vamos usar essa abordagem na seguindo o exemplo.

Vamos escrever um módulo Web nova chamada NHibernateConversationWebModule. Ele armazenará os NHibernate sessão entre os pedidos, em vez de descartá-lo. Também irá lidar com a re-anexando da sessão para o novo contexto.

10.6 A listagem NHibernateConversationWebModule para conversas

```
NHibernateConversationWebModule public class: IHttpModule {  
    const string NHibernateSessionKey = "NHibernate.NHibernateSession"; | 1  
  
    const string EndOfConversationKey = "NHibernate.EndOfConversation";  
    EndConversationAtTheEndOfThisRequest public static void () { | 2  
        HttpContext.Current.Items [EndOfConversationKey] = true;  
    }  
    public void Init (contexto HttpApplication) { | 3  
        context.PreRequestHandlerExecute +=  
            novo EventHandler (OnRequestBeginning);  
        context.PostRequestHandlerExecute +=  
            novo EventHandler (OnRequestEnding);  
    }  
    public void Dispose () {  
    }  
    OnRequestBeginning private void (object sender, EventArgs e) { | 4  
        // Dando continuidade a uma conversa?  
        ISession currentSession =  
            (ISession) HttpContext.Current.Session [NHibernateSessionKey]; | 5  
  
        if (currentSession == null) { | 6  
            // Conversa / Nova  
            currentSession NHibernateHelper.OpenSession = ();  
            currentSession.FlushMode = FlushMode.Never; | 7  
        }  
        CurrentSessionContext.Bind (currentSession);  
        currentSession.BeginTransaction (); | 8  
    }  
    OnRequestEnding private void (object sender, EventArgs e) { | 9  
        // Session / Libertação após o processamento  
        CurrentSession ISession = | 10  
            CurrentSessionContext.Unbind (NHibernateHelper.SessionFactory);  
}
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        / / End ou manter a conversação de longa duração?
        if (HttpContext.Current.Items [EndOfConversationKey] = null) { | 11
            currentSession.Flush (); | 12
            currentSession.Transaction.Commit ();
            currentSession.Close ();
            HttpContext.Current.Session [NHibernateSessionKey] = null;
        }
        else { | 13
            currentSession.Transaction.Commit ();
            HttpContext.Current.Session [NHibernateSessionKey] = currentSession;
        }
    }

}

```

Como estaremos armazenando a sessão NHibernate em um mapa, precisamos de uma chave # 1 para definir sua localização. Como o lógica de negócios é responsável por terminar a conversa, ele precisa chamar o `EndConversationAtTheEndOfTheRequest ()` # método 2 para definir um valor no contexto atual que ser usado no final do pedido. A inicialização do módulo # 3 registros de eventos para o início e fim de pedidos. Note que não estamos usando os mesmos eventos como na listagem de 11,5 porque estes um nos permitem acessar o estado da sessão ASP.NET.

Ao começar um novo pedido # 4, se uma conversa já está rodando, podemos extrair a sua destacada NHibernate sessão a partir do estado da sessão ASP.NET # 5. Caso contrário, # 6, começamos uma nova conversa por abertura de uma nova sessão. Vamos explicar por que definir o seu modo de lavar nunca # 7 na próxima seção. Uma vez que tenhamos a sessão NHibernate da conversa em execução, que vinculá-lo ao contexto atual # 8 e começamos uma nova transação. Neste ponto, a conversa está pronto para ser usado em qualquer lugar dentro este pedido web. Quando o tempo para terminar o pedido vem # 9, destacamos sua sessão NHibernate do contexto atual # 10. Se o valor para terminar a conversa foi set # 11, que lave manualmente a sessão # 12 a processar todas as as alterações feitas na conversa, nós nos comprometemos essas alterações, fechar a sessão, e nós removemos -la do estado da sessão ASP.NET. Se a conversa é apenas suspenso # 13, nós nos comprometemos a operação para fechar sua conexão com o banco e nós armazenamos a sessão no estado da sessão ASP.NET. Este conversa será retomada quando começa o próximo pedido.

Esta implementação não é completa porque a parte de tratamento de exceção está faltando. Referem-se ao código-fonte do CaveatEmptor para um exemplo. Basicamente, estes métodos devem estar dentro de um `try / catch` declaração. Quando pegar uma exceção, devemos reverter a transação corrente e retire em seguida, fechar a sessão atual. Há também outra questão que será abordada na próxima seção. Agora, vamos ver como poderíamos usar o módulo de conversação (não se esqueça de registrá-lo). Em nossa exemplo, o administrador está vendo, e depois aprovar um leilão. Portanto, teremos uma conversa que se estende por dois pedidos de web: O primeiro a ver a ação ea segunda para aprová-lo. A seguinte página web ASP.NET exibição de itens do leilão ao carregar (primeira solicitação) e fornece um botão para aprovar este leilão (segundo pedido):

```

public partial class ApproveItem: System.Web.UI.Page {
    ...
    const string ItemKey = "NHiA.ItemKey";

    protected void Page_Load (object sender, EventArgs e) {
        if (! IsPostBack) {
            longo itemId = long.Parse (Context.Request.QueryString ["Id"]);

            / / Primeiro pedido implicitamente iniciar a conversa
            Item item = itemDAO.FindById (itemId);
            Session [ItemKey] = item;
            editItemName.Text item.Name = / / ... Mostrar o item
            btnApprove.Click += new EventHandler (btnApprove_Click);
        }
    }
}

```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        }

        protected void btnApprove_Click (object sender, EventArgs e) {

            / / Segundo pedido usa a conversa em execução e termina-lo
            Item item = Session [Item] [ItemKey];
            item.Approve (loggedUser);
            NHibernateConversationWebModule.EndConversationAtTheEndOfThisRequest ();

            Context.Response.Redirect ("Default.aspx");
        }
    }
}

```

Neste exemplo, nós armazenamos o item no estado da sessão ASP.NET entre os pedidos. Não se esqueça de habilitá-lo.

No caso do aplicativo do Windows, a implementação de uma conversa seria mais simples, porque nós pode guardar tudo localmente. Também é possível imitar este exemplo usando o `CallContext` classe.

Também é possível para apoiar o cancelamento de uma conversa. Tudo que você precisa fazer é substituir o `EndConversationAtTheEndOfTheRequest ()` método por dois métodos: Um para aceitar as alterações e outro para cancelá-las. Então, você terá de distinguir esses valores quando o fim da conversa. Cancelamento de uma conversão é feita por fechar a sessão sem rubor-lo.

Há uma exceção a esta solução. A fim de compreender este problema, precisamos explicar algumas A teoria por trás de nossa implementação de conversas. Então, vamos explicar como lidar com este problema.

Garantindo atomicidade e compensação mudanças

Como você viu na listagem 10.6, vamos definir o modo de lavar da sessão aberta para nunca. É hora de explicar o razão por trás disso.

Como explicado anteriormente, uma conversa é suposto a se comportar como uma transação de banco de dados. Um dos seus requisitos deve ser atômica. , A fim de garantir a atomicidade de uma conversa, todas as mudanças feito pelos pedidos deve ser cometido quando terminar a conversa.

No entanto, por padrão, cometendo uma transação faz NHibernate confirmar as alterações detectadas. Isto é porque a sessão é lavada naquele momento: A sessão de coleta de todas as mudanças feitas desde que está aberto e executa os comandos SQL correspondente.

A solução para alterar esse comportamento é definir a sessão NHibernate para `FlushMode.Never` e explicitamente nivelá-lo no final da conversa. Todas as alterações são guardadas na memória (na verdade, na NHibernate sessão) até o esvaziamento explícito. Observe que as consultas não estar ciente destas un-lavada mudanças e podem retornar dados obsoletos.

Agora, a exceção: Ao salvar uma entidade cujo identificador é gerado pelo banco de dados (por exemplo, ao usar nativo ou identidade), NHibernate deve salvar essa entidade imediatamente, a fim de recuperar seu identificador. A razão por trás deste comportamento é que o método `ISession.Save ()` preciso devolver este identificador. Dependendo do comportamento do seu banco de dados, esta economia pode ter lado adicionais efeitos.

Se qualquer alteração permanente como o que acontece em uma solicitação antecipada e que a conversa deve ser cancelada, será necessário executar algumas ações de compensação para reverter essas mudanças. Isto é também o caso quando a sessão NHibernate gera uma exceção: A conversa deve ser cancelado ea sessão fechada.

Você pode dar uma olhada no `IInterceptor` API (usado no capítulo 9, seção 9.4) para manter o controle dessas mudanças permanentes e revertê-los se necessário. Alternativamente, você pode simplesmente evitar estes geradores.

Note que, você pode querer conversas para manter as alterações em cada pedido. Ele pode permitir que você fornecer um recurso de recuperação em caso de falha do sistema, por exemplo. No entanto, lembre-se que você vai também têm de fornecer ações de compensação sempre que uma conversa pode ser cancelado.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Há uma complicação final potenciais relacionadas com a abordagem longa sessão. NHibernate ISession implementação não é thread-safe. Assim, se um ambiente permite que várias solicitações a partir do mesmo usuário a ser processadas simultaneamente, é possível que essas solicitações simultâneas poderiam obter o NHibernate mesmo ISession exemplo. Isso poderia resultar em um comportamento imprevisível. Este problema também afeta a abordagem anterior, onde usamos objetos individual, desde objetos destacados também não são thread-safe. Na verdade, este problema afeta qualquer aplicativo que mantém estado mutável em um não thread-safe cache.

Como este é um problema genérico, vamos deixar isso para encontrar uma solução adequada se você estiver confrontado com ele. Uma boa solução para algumas aplicações pode ser de rejeitar qualquer nova solicitação se um pedido já está sendo processada para o mesmo usuário. Outras aplicações pode precisar serializar solicitações do mesmo usuário. Note que este não é um problema para aplicações Web utilizando o ASP.NET o estado da sessão porque solicitações simultâneas (do mesmo usuário) são automaticamente serializado, o segundo pedidos de esperar que o primeiro for concluído.

Agora que nós cobrimos três maneiras diferentes de lidar com as conversas, você pode ser confundido

10.2.5 Escolhendo uma abordagem para conversas discutimos relevante?

Você provavelmente pode adivinhar o fato de que nós chamamos alguma coisa da maneira mais difícil que nós não acho que é uma

boa técnica. Nós não iríamos usar essa abordagem em nossos próprios aplicativos. No entanto, se sua arquitetura específica que a camada web nunca deve acessar o modelo de domínio diretamente (e por isso o modelo de domínio é

completamente escondido da camada de apresentação por trás de uma camada intermediária de abstração DTO), e se

você não conseguir manter o estado associado com o usuário porque você está usando um quadro apátridas, em seguida,

você tem essencialmente nenhuma outra escolha. É possível construir aplicações NHibernate desta maneira, e NHibernate foi projetado para suportar esta abordagem. Pelo menos essa abordagem liberta-o de ter que considerar a diferença entre instâncias persistentes e individual, e isso elimina a possibilidade de LazyInitializationExceptions lançadas por objetos soltos.

Atualmente, a maioria das aplicações NHibernate escolher a abordagem objetos individual, com uma nova sessão por transação de banco de dados. Em particular, este é o método de escolha para uma aplicação onde o negócio acesso a lógica e os dados executados no Data Access Layer, mas onde o modelo de domínio também é usado na camada de apresentação, evitando a necessidade de DTOs tedioso. Esta abordagem é mesmo sendo usado com sucesso em

Aplicações do Windows. Estamos inclinados a pensar que há muitos casos em que não é o melhor abordagem, no entanto.

Há muitos casos (a maioria deles bastante complexos) em que usaria a abordagem longa sessão na um aplicativo do Windows. Até agora, descobrimos que esta abordagem difícil de explicar, e não é bem compreendidas na comunidade NHibernate. Nós supor que este é porque a noção de uma conversa não é bem compreendido na comunidade de desenvolvedores, ea maioria dos desenvolvedores não estão acostumados a pensar em

problemas em termos de conversas. Esperamos que esta situação mude em breve, porque essa idéia é útil mesmo se você não usar a abordagem longa sessão.

O próximo passo é ver como podemos levar esse código e adaptá-lo para rodar em Serviços Corporativos aplicação. Obviamente, gostaríamos de mudar o mínimo possível. Nós estivemos discutindo o tempo todo que um vantagem de Poços e persistência transparente é a portabilidade entre diferentes runtime ambientes. Se agora temos que reescrever todo o código para fazer um lance, vamos olhar um bocado parvo.

Usando 10.3 NHibernate em um aplicativo do Enterprise Services

Por Serviços de aplicativos corporativos, queremos dizer uma aplicação que tira proveito da distribuição serviço de transação de. NET Enterprise Services. O Capítulo 5 contém uma breve explicação dos passos obrigado a fazer uma aplicação NHibernate participar de uma transação distribuída. Agora, vamos para implementar esta abordagem.

Antes disso, devemos cobrir uma questão ligada à inter-processo pedidos. Sempre que você tem fisicamente separados os componentes / camadas, você deve minimizar a comunicação entre eles. Isto é importante porque a latência é adicionado por cada requisição processo de inter-, aumentando o tempo de resposta das aplicações e

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

redução da concorrência, devido à necessidade para qualquer banco de dados mais transações ou operações mais.

Além disso, influencia a escalabilidade do seu aplicativo.

Portanto, é essencial que todo acesso a dados relacionados a um pedido de usuário único ocorrer dentro de um único

pedido para a camada de persistência. Isto significa que você não pode usar uma abordagem preguiçosa, onde a apresentação

camada de puxar os dados, conforme necessário. Em vez disso, a camada de negócios deve aceitar a responsabilidade de buscar todos os dados que serão necessários posteriormente pela camada de apresentação.

O onipresente de transferência de dados objeto (DTO) fornece uma maneira padrão de embalagens junto aos dados

que a camada de apresentação irá precisar. Um DTO é uma classe que contém o estado de uma entidade

10.3.1 Particularidades dos objetos de transferência de dados

Pode pensar em um DTO como um POCO sem métodos de negócio. Mas como DTOs tendem a duplicar as entidades, nós

certos argumentos são válidos para entender o uso de DTOs. No entanto, DTOs não deve confundir essas argumentos para a verdadeira razão pela qual DTOs são tão úteis.

A idéia por trás do padrão DTO é que de grão fino de acesso remoto é lento e não-escalável. Ele também passou a ser útil quando o modelo de domínio não pode ser feita serializável. Neste caso, outra objeto deve ser usado para empacotar e carregar o estado dos objetos de negócios entre as camadas.

Agora há justificativas duplo para o uso de DTOs: primeiro, implementar DTOs externalização de dados entre camadas, em segundo lugar, reforçar DTOs separação da camada de apresentação da lógica de negócios

tier. Apenas a segunda justificativa se aplica a nós, eo benefício dessa separação é questionável quando pesado contra o seu custo. Nós não vamos dizer-lhe para nunca usar DTOs (em outros lugares, estamos às vezes menos

reticentes). Em vez disso, vamos listar alguns argumentos a favor e contra o uso do padrão DTO em um aplicativo que usa NHibernate e pedir-lhe para pesar cuidadosamente os argumentos no contexto de sua própria aplicação.

É verdade que o DTO remove a dependência direta da camada de apresentação sobre o modelo de domínio. Se as partições do seu projeto o papel de .NET e web designer, isso pode ser de algum valor.

Em particular, o DTO permite que você alise associações modelo de domínio, transformando os dados em um formato

que talvez seja mais conveniente para fins de apresentação (que pode facilitar bastante ligação de dados).

No entanto, em nossa experiência, é normal para todas as camadas da aplicação a ser altamente acoplado ao modelo de domínio, com ou sem o uso de DTOs. Não vemos nada de errado com isso, e nós sugerem que ela pode ser possível aceitar o fato.

O primeiro indício de que algo está errado com DTOs é que, ao contrário do seu título, elas não são objetos em tudo. DTOs definir o estado sem comportamento. Esta é imediatamente suspeito no contexto do objeto desenvolvimento orientado. Pior ainda, o estado definido pelo DTO é muitas vezes idêntico ao estado definido nos objetos de negócio do domínio modelo de separação o suposto alcançado pelo padrão DTO também poderia ser visto como mero duplicação.

O padrão DTO exibe dois dos cheiros código descrito em [Fowler 1999]. O primeiro é o mudança de espingarda cheiro, onde uma pequena alteração para alguns requisitos de sistema requer mudanças a vários

classes. O segundo é o hierarquias de classe paralela cheiro, onde duas diferentes hierarquias de classe conter classes semelhantes em uma correspondência one-to-one. A hierarquia de classes paralelas é evidente neste caso de sistemas que usam o padrão DTO têm Item e ItemDTO, Usuário e UserDTOE assim por diante. O cheiro mudança shotgun manifesta-se quando adicionamos uma nova propriedade Item. Temos que mudar não só a camada de apresentação ea Item classe, mas também a ItemDTO eo código que monta a ItemDTO exemplo das propriedades de um Item (Este último pedaço de código é especialmente tedioso e frágil).

Claro, DTOs não são todos ruins. O código que apenas referido como "tedioso e frágil"-o assembler-se ter algum valor, mesmo no contexto do NHibernate. DTO montagem fornece-lhe com um ponto em que conveniente para garantir que todos os dados da camada de apresentação irá necessitar é totalmente improvável

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

antes de retornar o controle para a camada de apresentação. Se você se encontra lutando com NHibernate LazyInitializationExceptions na camada de apresentação, uma possível solução é tentar o DTO padrão, o que naturalmente impõe disciplina extra, exigindo que todos os dados necessários são copiados explicitamente

dos objetos de negócio (que não acho que precisamos desta disciplina, mas sua experiência pode variar).

Finalmente, DTOs pode ter um lugar na transferência de dados entre aplicativos de baixo acoplamento (nossa discussão centrou-se sobre a sua utilização na transferência de dados entre as camadas da mesma aplicação). No entanto,

DataSet digitado parece ser melhor adaptada para este problema.

Um DataSet digitado pode ser visto como um tipo especial de DTO. Há razões definitivamente bom para usar DataSet: Há um conjunto de ferramentas disponíveis e extensa DataSet use muitas das actuais bibliotecas. No entanto,

escrever classes personalizadas como DTO dá um melhor controle sobre o design do seu aplicativo mesmo que seja tediosa de escrever.

Não vamos usar DTOs na aplicação CaveatEmptor. Agora que nós cobrimos o problema potencial que podem ocorrer quando se lida com camadas separadas fisicamente, podemos voltar para distribuição transações.

10.3.2 Implementação de um transações distribuídas habilitado NHibernateHelper classe

Há algumas mudanças que devem ser aplicados para a classe NHibernateHelper anterior, a fim de permitir transações distribuídas. Primeiro, devemos adicionar uma referência ao conjunto System.EnterpriseServices

e alterar a definição de classe como esta:

```
[Transaction (TransactionOption.Supported)]
public class NHibernateHelper: ServicedComponent {
    ...
}
```

Nós também deve adicionar os seguintes métodos para simplesmente gestão da operação:

BeginTransaction (), CommitTransaction () e RollbackTransaction (). Eles serão utilizados para criar, commit / rollback e fechar a transação distribuída. Aqui estão os dois primeiros métodos:

```
BeginTransaction public static void () {
    ServiceConfig sc = ServiceConfig new ();
    sc.Transaction = TransactionOption.RequiresNew;
    ServiceDomain.Enter (sc);
}

CommitTransaction public static void () {
    try {
        ContextUtil.SetComplete ();
        ServiceDomain.Leave ();
    }
    catch (HibernateException ex) {
        throw new InfrastructureException (ex);
    }
}
```

A fim de tomar parte para a transação distribuída, a transação a sessão NHibernate deve ser alistados. Note que temos que usar o Reflection. NET aqui, porque o método utilizado não é parte do IDbConnection interface.

```
TryEnlistDistributedTransaction private static void (sessão ISession) {
    if (ContextUtil.IsInTransaction) {
        IDbConnection conn = session.Connection;
        MethodInfo mi = conn.GetType () . GetMethod (
            "EnlistDistributedTransaction",
            BindingFlags.Public | BindingFlags.Instance);

        if (mi! = null)
            mi.Invoke (conn,
                novo objeto [] {
                    (System.EnterpriseServices.ITransaction)
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        ContextUtil.Transaction));
    }
}

```

Note-se que, no caso que o `EnlistDistributedTransaction()` método não está disponível, este código falhará silenciosamente para alistar-se a transação. Se ela representa um erro no seu caso de uso, você deve lançar uma exceção. Você pode até mesmo evitar a reflexão, se você sabe que tipo de conexão do banco de dados é utilizado.

Assumindo que o aplicativo deve sempre tentar recorrer a transação distribuída, você pode alterar a implementação do método `OpenSession()` para:

```

openSession ISession sessionFactory.openSession = ();
TryEnlistDistributedTransaction (sessão);
retorno da sessão;
}

```

Tudo o que resta a fazer é atualizar o código para usar estes novos métodos. Além disso, não se esqueça de registrar o COM + resultantes de montagem (que deve ser assinado) antes de usá-lo. O instrumento utilizado para esta operação é linha de comando `RegSvcs` executável.

Nós chegamos ao fim da nossa discussão sobre a camada de persistência. Vamos rever o que nós cobrimos em

Resumo 10.4

Neste capítulo, focado na concepção da camada de persistência. Nós introduzido pela primeira vez o `NHibernateHelper` classe que é útil para a inicialização do resumo do NHibernate. Também mostramos como passar de um método monolítico mistura todas as preocupações para uma arquitetura limpa com uma clara separação entre as camadas.

Ilustramos um modelo de domínio "inteligentes" por lógica de negócios de execução no `CaveatEmptor` Item classe. Este foi o primeiro passo da nossa série de refatoração.

Nós usamos o padrão DAO para criar uma fachada para a camada de persistência, escondendo internals NHibernate das outras camadas. Nós também introduziu o `ISessionFactory.GetCurrentSession()` API e os noção de contexto. Nós usamos esse recurso para melhorar significativamente a nossa DAOs, tornando-os compartilhar o mesma sessão, sem ter que passá-lo como um parâmetro e sem usar uma sessão global estática.

Depois disso, nós alavancou a .NET 2,0 genéricos para reduzir os despedimentos na camada de persistência. Nós também projetamos nosso camada de persistência para que as outras camadas são completamente inconscientes da framework de persistência que é usado. Nós mesmo, foi possível mudar de uma implementação para outro por mudar uma única linha de código.

Também explicou como fazer uma sessão ao vivo para um pedido de web inteiro. Isso é útil para garantir a que uma única sessão é usada para todo o processamento e carregamento lento que sempre funciona transparente.

Capítulo 5 introduziu a noção de conversa (também chamado de application / negócios transações). Em Neste capítulo, desde três formas de implementá-las: O maneira mais difícil, que pode ser sua única escolha em alguns ambientes limitados; a abordagem usando destacado objetos persistentes, útil em ambiente apátridas ea abordagem utilizando longa vida sessões. Esta última abordagem é ainda relativamente desconhecido. No entanto, é muito poderosa, especialmente em um ambiente rico.

Este capítulo termina com a melhoria da `NHibernateHelper` classe a fim de apoiar empresas Operações de serviços. Discutimos a questão da latência do potencial e utilidade do padrão DTO. Embora este padrão pode ser útil quando dissociar o modelo de domínio é importante, nós concordamos que tal situação é rara e que o custo de manter o DTO é muito alto.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Neste ponto, você deve ter todo o conhecimento técnico que pode ser necessário para alavancar o recursos do NHibernate. É uma ferramenta poderosa, mas requer uma profunda compreensão do seu comportamento a ser correctamente utilizados.

Observe que você não terminar de ler este livro ainda. Você vai descobrir, nos apêndices, mais alguns ferramentas e idéias que irão melhorar sua experiência de trabalho com NHibernate.

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

A

Fundamentos SQL

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Este livro assume um conhecimento básico de bancos de dados relacionais e os Structured Query Language (SQL). Além disso, será mais fácil para você aprender alguns dos recursos avançados do NHibernate se já tem um bom conhecimento de SQL. Este apêndice fornece uma breve visão geral dos fundamentos da SQL. Portanto, recomendamos que você encontrar um livro para aprender mais sobre ele. Há alguns recomendações no capítulo 1, seção 1.1.1.

Uma tabela, com suas linhas e colunas, é uma visão familiar para quem já trabalhou com um SQL banco de dados. Às vezes, você verá as tabelas referidas como relações, linhas como tuplas e colunas como atributos. Esta é a linguagem do modelo de dados relacional, o modelo matemático que SQL bases de dados (imperfeitamente) implementar.

O modelo relacional permite definir estruturas de dados e restrições que garantem a integridade dos seus dados (por exemplo, não permitindo que os valores não estão de acordo com suas regras de negócio).

O modelo relacional também define as operações relacional de restrição, projeção, produto cartesiano, e relacional juntar [Codd 1970]. Essas operações permitem que você faça coisas úteis com os seus dados, tais como resumindo ou navegando-lo.

Cada uma das operações produz uma nova tabela a partir de uma determinada tabela ou uma combinação de tabelas. SQL é um linguagem para expressar estas operações em sua aplicação (por isso chamado de linguagem de dados) e para definir as tabelas de base sobre a qual as operações são executadas.

Você escreve instruções SQL DDL para criar e gerenciar as tabelas. Dizemos que define o DDL esquema de banco de dados. Afirmações como CREATE TABLE, ALTER TABLE E CREATE SEQUENCE pertencem a DDL.

Você escreve instruções SQL DML a trabalhar com os seus dados em tempo de execução. Vamos descrever estes DML

operações no contexto das tabelas da aplicação CaveatEmptor.

Em CaveatEmptor, nós naturalmente temos entidades como item, usuário, e lance. Assumimos que o SQL esquema de banco de dados para esta aplicação inclui um ITEM mesa e uma BID tabela, como mostrado na figura A.1.

Os tipos de dados, tabelas e restrições para este esquema ter sido criado com o SQL DDL (CRIAR e ALTER operações).

ITEM		
ITEM_ID	NAME	INITIAL_PRICE
1	Foo	2.00
2	Bar	50.00
3	Baz	1.00

BID		
BID_ID	ITEM_ID	AMOUNT
1	1	10.00
2	1	20.00
3	2	55.50

Figura A.1 O ITEM e BID tabelas de um aplicativo de leilão

Inserção é a operação de criar uma nova tabela a partir de uma tabela antiga, adicionando uma linha. Bancos de dados SQL

executar esta operação no local, para que a nova linha é adicionada à tabela existente:
inserir valores ITEM (4, 'Fum', 45, 0)

Uma atualização SQL modifica uma linha existente:

```
ITEM atualização do conjunto INITIAL_PRICE = 47,0 onde item_id = 4
```

A eliminação remove uma linha:

```
excluir ITEM onde item_id = 4
```

O poder real do SQL está em consulta de dados. Uma única consulta pode realizar muitas operações relacionais em várias tabelas. Vamos olhar para as operações básicas.

Em primeiro lugar, restrição é a operação de escolha de linhas de uma tabela que correspondem a um critério particular. Em

SQL, este critério é a expressão que ocorre no onde cláusula:

```
select * from ITEM onde NAME como 'F%'
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Projeção é a operação de escolha de colunas de uma tabela e eliminar linhas duplicadas da resultado. Em SQL, as colunas a serem incluídos são listados na selecionar cláusula. Você pode eliminar duplicados linhas especificando o distinto palavra-chave:

```
seleccione NOME distinta ITEM
```

A Produto cartesiano (Também chamado de junção cruzada) produz uma nova tabela composta por todas as possíveis combinações de linhas de duas tabelas existentes. Em SQL, você expressa um produto cartesiano de tabelas de listagem no ~~a partir de~~ cláusula:

A relacionais juntar produz uma nova tabela, combinando as linhas de duas tabelas. Para cada par de linhas para que um condição de junção É verdade, a nova tabela contém uma linha com todos os valores de campo de ambos os juntou linhas. No ANSI SQL, o juntar cláusula especifica uma tabela de junção, a condição de junção segue o em palavra-chave.

Por exemplo, para recuperar todos os itens que têm propostas, se juntar à ITEM eo BID tabela na sua comum Item_id atributo:

```
select * from ITEM i BID junção interna em i.ITEM_ID b = b.ITEM_ID
```

A junção é equivalente a um produto cartesiano seguido de uma restrição. Assim, junta-se muitas vezes ao invés expressa em estilo teta, com um produto no a partir de cláusula e condição de junção na onde cláusula. Este SQL juntar theta estilo é equivalente ao anterior ANSI-estilo join:

```
select * from ITEM i b BID, onde i.ITEM_ID = b.ITEM_ID
```

Juntamente com estas operações básicas, bancos de dados relacionais definir operações para agregar linhas (GROUP

POR) E ordenação das linhas (ORDER BY):

```
selecionar b.ITEM_ID, max (b.AMOUNT)
de b BID
grupo por b.ITEM_ID
ter max (b.AMOUNT) > 15
por fim b.ITEM_ID asc
```

SQL foi chamado de estruturado linguagem de consulta em referência a um recurso chamado subselects. Uma vez que cada

operação relacional produz uma nova tabela de uma tabela existente ou tabelas, uma consulta SQL pode operar na tabela resultado de uma consulta anterior. SQL permite que você expressa isso usando uma única consulta, pelo assentamento:

```
select *
a partir de (
    selecionar b.ITEM_ID como ITEM, max (b.AMOUNT) como VALOR
    de b BID
    grupo por b.ITEM_ID
)
onde VALOR > 15
por fim ITEM asc
```

O resultado dessa consulta é equivalente ao anterior.

Um subselect pode aparecer em qualquer lugar em uma instrução SQL; o caso de um subselect na onde cláusula

é o mais interessante:

```
select * from b BID onde b.AMOUNT >= (select max (c.AMOUNT) de c BID)
```

Esta consulta retorna o maior lance no banco de dados. onde subselects cláusula são muitas vezes combinados com

quantificação . A consulta a seguir é equivalente:

```
select * from b BID onde b.AMOUNT >= all (select c.AMOUNT do BID c)
```

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Um critério de restrição SQL é expressa em uma linguagem de expressão sofisticada que suporta expressões matemáticas, chamadas de função, strings, e ainda mais funcionalidades sofisticadas, tais como pesquisas de texto completo:

```
select * from ITEM i  
onde inferiores (i.NAME) like '% ba%'  
ou inferior (i.NAME) '% para% "como
```

Existem muitas outras operações em SQL. Você pode dar uma olhada no capítulo 8 para ter uma idéia das operações que você quer aprender.

D

Seguir em Frente

Por favor, postar comentários ou correções para o fórum on-line em Autor
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Você chegou ao final deste livro. O que resta para nós a fazer é dar-lhe alguma orientação e aconselhamento para começar e para dominar NHibernate.

Neste apêndice, vamos enumerar os requisitos para usar NHibernate. Então, nós damos-lhe um roteiro progressivamente mestre NHibernate e manter-se up-to-date. Finalmente, recomendamos que você descobrir as partes internas do NHibernate, contribuir e ajudar a melhorar o NHibernate.

Este livro pressupõe que você tenha alguma experiência .NET. Portanto, antes de começar usar NHibernate, você já deve ter o quadro .NET e um Integrated Development

Ambiente (IDE) como o Visual Studio ou SharpDevelop. Note que você também pode usar Mono ([Http://www.mono-project.com/](http://www.mono-project.com/)) que funciona em outros sistemas operacionais como o Linux.

NHibernate binários, código fonte e documentação estão disponíveis em seu site SourceForge: <http://sourceforge.net/projects/nhibernate/>. SourceForge.net é um site oferecendo hospedagem gratuita para Open Source Software (OSS) projetos de desenvolvimento.

Antes NHibernate 1.2, havia dois pacotes disponíveis: nhibernate contendo o núcleo binários com seu código fonte e documentação e NHibernateContrib contendo opcional útil add-ons para o NHibernate. Agora, eles são mesclados. Além disso, você vai precisar NHibernate 1.2 ou mais tarde para aproveitar .NET 2.0 genéricos e nullables.

Você pode usar NHibernate com sistemas de banco de dados mais popular. Você pode encontrar a lista completa aqui:

<http://www.hibernate.org/361.html>.

Isso é tudo que você precisa para começar a usar NHibernate. O próximo passo é praticar usando NHibernate, a partir de

o exemplo simples Olá Mundo do capítulo 2 para uma aplicação mais complexa como CaveatEmptor.

Nós encorajamos você a desenvolver suas próprias aplicações para testar os principais recursos do NHibernate e depois integrar os recursos avançados que lhe interessam. Neste ponto, este livro vai servir como um livro de referência.

Você deve se certificar que você entendeu o mapeamento de entidades e seu ciclo de vida de persistência ea forma como NHibernate sessões de trabalho. Tome também cuidado especial com a forma como você usa NHibernate

cache. Finalmente, os três últimos capítulos explicam a importância da arquitetura da aplicação é.

Muitas vezes você vai encontrar alguns problemas ao usar o NHibernate. Na maioria dos casos, sua fonte é o mau uso de alguns recursos, devido a uma falta de compreensão. Para mais conselhos sobre resolução de problemas técnicas, leia o capítulo 9, seção 9.3. Note-se que um projeto bem pensado ajuda muito a evitar e resolver problemas, por isso levará algum tempo para pensar sobre as características que você deseja usar (algoritmos) e os arquitetura de sua aplicação (camadas, separação de preocupação, etc).

No caso de você ainda não conseguir superar suas dificuldades, não hesite em pedir ajuda na NHibernate fórum: <http://forum.hibernate.org/viewforum.php?f=25>. Certifique-se de explicar a sua problema em detalhes, com os logs e mensagens de erro.

NHibernate está em constante evolução. Portanto, uma vez que você se sentir confortável a usá-lo, você deve manter yourself up-to-date porque alguns novos recursos pode melhorar as suas capacidades de aplicações e desempenho.

A melhor maneira de fazer isso é ler regularmente e participar do Fórum NHibernate. É também um ótimo lugar para compartilhar seu ponto de vista sobre vários problemas relacionados com NHibernate, obter feedback sobre

-los e aprender como as outras pessoas resolvê-los.

Há também muitos recursos informativos e úteis (documentação, amostras de código aberto projetos) disponíveis na Internet. Você pode encontrar uma lista completa aqui: <http://www.hibernate.org/365.html>.

NHibernate tem um site de rastreamento de bugs: <http://jira.nhibernate.org/>. Você pode se cadastrar neste site e reportar os bugs que você encontrar (caso você não tem certeza de que é um bug, use o NHibernate primeiro fórum), você também pode solicitar novos recursos.

Desde NHibernate é uma OSS, seu código fonte está disponível gratuitamente. Em vez de usar o compilado biblioteca, use o código fonte para obter mais detalhes quando a depuração do seu aplicativo.

Além disso, sinta-se livre para modificá-lo sempre que precisar adicionar um novo recurso que a sua aplicação necessidades ou para corrigir um bug existente. Não se esqueça de fazer esta adição à disposição do público para que outras

NHibernate usuário pode usar e até mesmo melhorá-lo. Você pode fazer isso através da apresentação de um remendo no-bug

site de rastreamento (é um arquivo contendo as alterações feitas no código fonte).

O site de acompanhamento de bugs, também dá uma idéia dos recursos e correções de bugs que já estão disponíveis na versão atual do NHibernate (que os desenvolvedores estão trabalhando em NHibernate). O código fonte desta versão é hospedado por Sourceforge.net em um repositório SVN. Se você está interessado em usando esta versão, leia "Introdução ao Código Fonte NHibernate" no <http://www.hibernate.org/428.html> e vá para http://sourceforge.net/svn/?group_id=73818. Note-se que, embora esta versão é geralmente estável, ele pode temporariamente tornar-se instável por tempo ao tempo.

Depois de começar a usar a versão SVN do NHibernate, há apenas um último passo para abraçar NHibernate completamente: Juntar-se à lista de desenvolvimento. É uma lista de endereços usados pelos desenvolvedores de

NHibernate para discutir sua evolução. Você pode começar por se registrar e ler o arquivo aqui: <http://lists.sourceforge.net/mailman/listinfo/nhibernate-development>.

Este é o fim do livro e, esperançosamente, o início de uma experiência maravilhosa com NHibernate.

Bon voyage!