



C#

# EM PROFUNDIDADE

TERCEIRA EDIÇÃO

Jon Skeet

PREFÁCIO DE ERIC LIPPERT



TRIPULAÇÃO

## Elogios à segunda edição

*Uma obra-prima sobre C#.*

—Kirill Osenkov, equipe de C# da Microsoft

*Se você deseja dominar C#, este livro é uma leitura obrigatória.*

—Tyson S. Maxwell  
Engenheiro de software sênior, Raytheon

*Apostamos que este será o melhor livro sobre C# 4.0 que existe.*

—Nikander Bruggeman e Margriet Bruggeman consultores .NET,  
Lois & Clark IT Services

*Uma visão útil e envolvente sobre a evolução do C# 4.*

—Joe Albahari  
Autor de LINQPad e C# 4.0 em poucas palavras

*Um dos melhores livros sobre C# que já li.*

—Aleksey Nudelman  
CEO, C# Computação, LLC

*Este livro deve ser leitura obrigatória para todos os desenvolvedores profissionais de C#.*

—Stuart Caborn  
Desenvolvedor Sênior, BNP Paribas

*Um recurso de nível mestre altamente focado em atualizações de linguagem em todas as principais versões do C#. Este livro é essencial para o desenvolvedor especialista que deseja se manter atualizado com os novos recursos da linguagem C#.*

—Sean Reilly, programador/analista  
Tecnologias Point2

*Por que ler o básico repetidamente? Jon se concentra nas coisas novas e mastigáveis!*

—Keith Hill, arquiteto de software da  
Agilent Technologies

*Tudo o que você não sabia que precisava saber sobre C#.*

—Jared Parsons  
Engenheiro de Desenvolvimento de Software Sênior  
Microsoft

## Elogios à Primeira Edição

*Simplificando, C# in Depth é talvez o melhor livro de informática que já li.*

—Craig Pelkie, autor, *System iNetwork*

*Tenho desenvolvido em C# desde o início e este livro trouxe algumas boas surpresas até para mim. Fiquei especialmente impressionado com a excelente cobertura de delegados, métodos anônimos, covariância e contravariância. Mesmo se você for um desenvolvedor experiente, C# em profundidade lhe ensinará algo novo sobre a linguagem C#... Este livro realmente tem uma profundidade que nenhum outro livro em linguagem C# pode alcançar.*

—Adam J. Wolf  
Grupo de usuários .NET do Vale do Sudeste

*Gostei de ler o livro inteiro; está bem escrito – os exemplos são fáceis de entender.*

*Na verdade, achei muito fácil me envolver em todo o tópico de expressões lambda e gostei muito do capítulo sobre expressões lambda.*

"Joseph Roland Guy Paz  
Desenvolvedor Web, Soluções CSW

*Este livro resume o grande conhecimento do autor sobre o funcionamento interno do C# e o entrega aos leitores em um livro bem escrito, conciso e utilizável.*

—Jim Holmes  
Autor de *ferramentas poderosas para desenvolvedores do Windows*

*Cada termo é usado apropriadamente e no contexto certo, cada exemplo é preciso e contém a menor quantidade de código que mostra toda a extensão do recurso...este é um deleite raro.*

—Franck Jeannin, revisor da Amazon UK

*Se você desenvolve usando C# há vários anos e gostaria de conhecer os detalhes internos, este livro é absolutamente certo para você.*

—Golo Roden  
Autor, palestrante e instrutor de .NET  
e tecnologias relacionadas

*O melhor livro de C# que já li.*

—Chris Mullins, MVP de C#

# C# em profundidade

TERCEIRA EDIÇÃO

JON SKEETS



TRIPULAÇÃO  
ILHA ABRIGO

Para informações on-line e pedidos deste e de outros livros de Manning, visite  
[www.manning.com](http://www.manning.com). A editora oferece descontos neste livro quando encomendado em grande quantidade.  
Para mais informações por favor entre em contato

Departamento de Vendas Especiais  
Manning Publicações Co.  
Estrada Baldwin, 20  
Caixa Postal 261  
Ilha Shelter, NY 11964.

©2014 por Manning Publications Co. Todos os direitos reservados.

Nenhuma parte desta publicação pode ser reproduzida, armazenada em um sistema de recuperação ou transmitida, de qualquer forma ou por meio eletrônico, mecânico, fotocópia ou outro, sem permissão prévia por escrito do editor.

Muitas das designações utilizadas pelos fabricantes e vendedores para distinguir os seus produtos são reivindicadas como marcas comerciais. Onde essas designações aparecem no livro, e a Manning Publications estava ciente de uma reivindicação de marca registrada, as designações foram impressas em letras maiúsculas ou todas em letras maiúsculas.

- ⊗ Reconhecendo a importância de preservar o que foi escrito, é política da Manning imprimir os livros que publicamos em papel sem ácido, e envidamos os nossos melhores esforços para esse fim.
- Reconhecendo também a nossa responsabilidade de conservar os recursos do nosso planeta, Manning livros são impressos em papel que é pelo menos 15% reciclado e processado sem o uso de cloro elementar.



Manning Publicações Co.  
Estrada Baldwin, 20  
Caixa Postal 261  
Ilha Abrigo, NY 11964

Editor de desenvolvimento Jeff Bleiel  
Editor de texto: Andy Carroll  
Revisora: Katie Tennant  
Tipógrafo: Dottie Marsico  
Designer da capa: Marija Tudor

ISBN 9781617291340  
Impresso nos Estados Unidos da América  
1 2 3 4 5 6 7 8 9 10 – VEZES – 18 17 16 15 14 13

*Para meus meninos, Tom, Robin e William*



## *breve conteúdo*

---

<b>PARTE 1 PREPARANDO-SE PARA A VIAGEM.....</b>	<b>1</b>
1 ♦ A face mutável do desenvolvimento do C# 3 2 ♦	
Fundamentos essenciais: construindo em C# 1 29	
<b>PARTE 2 C# 2: RESOLVENDO OS PROBLEMAS DO C# 1 .....</b>	<b>57</b>
3 ♦ Tipagem parametrizada com genéricos 59 4 ♦	
Não dizer nada com tipos anuláveis 105 5 ♦	
Delegados acelerados 133 6 ♦	
Implementando iteradores de maneira fácil 159 7 ♦	
Concluindo C# 2: os recursos finais 182	
<b>PARTE 3 C# 3: REVOLUCIONANDO O ACESSO A DADOS .....</b>	<b>205</b>
8 ♦ Eliminando problemas com um compilador	
inteligente 207 9 ♦ Expressões lambda e árvores de expressão 232	
10 ♦ Métodos de extensão 262	
11 ♦ Expressões de consulta e LINQ to Objects 285 12 ♦	
LINQ além das coleções 328	

PARTE 4 C# 4: JOGANDO BEM COM OS OUTROS.....	369
13 ÿ Pequenas alterações para simplificar o código	371
14 ÿ Vinculação dinâmica em uma linguagem estática	409
PARTE 5 C# 5: ASSINCRONIA SIMPLES .....	461
15 ÿ Assincronia com async/await	463
16 ÿ Recursos bônus do C# 5 e considerações finais	519

# conteúdo

---

*prefácio xix  
prefácio xxi  
agradecimentos xxii sobre  
este livro xxiv  
sobre o autor xxxx  
sobre a ilustração da capa xxx*

## **PARTE 1 PREPARANDO-SE PARA A VIAGEM .....1**

### **1 A face mutável do desenvolvimento C# 3**

#### 1.1 Começando com um tipo de dados simples 4

*O tipo de produto em C# 1 5 ↴ Coleções fortemente tipadas em C# 2 6*

*Propriedades implementadas automaticamente em C# 3 7 ↴*

*Argumentos nomeados em C# 4 8*

#### 1.2 Classificação e filtragem 9

*Classificando produtos por nome 9 ↴ Consultando coleções 12*

#### 1.3 Lidando com a ausência de dados 14

*Representando um preço desconhecido 14 ↴ Parâmetros opcionais e valores  
padrão 16*

#### 1.4 Apresentando o LINQ 16

*Expressões de consulta e consultas em processo 17 ↴ Consultando  
XML 18 ↴ LINQ para SQL 19*

## 1.5 COM e digitação dinâmica 20

*Simplificando a interoperabilidade COM 20 ↴ Interoperando com uma linguagem dinâmica 21*

## 1.6 Escrevendo código assíncrono sem dor de cabeça 22

## 1.7 Dissecando a plataforma .NET 23

*C#, a linguagem 24 ↴ Tempo de execução 24 ↴ Bibliotecas de estrutura 24*

## 1.8 Tornando seu código super incrível 25

*Apresentando programas completos como fragmentos 25 ↴ Código didático não é código de produção 26 ↴ Seu novo melhor amigo: a especificação da linguagem 27*

## 1.9 Resumo 28

# 2 Fundamentos principais: construindo em C# 1 29

## 2.1 Delegados 30

*Uma receita para delegados simples 30 ↴ Combinando e removendo delegados 35 ↴ Uma breve diversão nos eventos 36 ↴ Resumo dos delegados 37*

## 2.2 Características do sistema de tipo 38

*O lugar do C# no mundo dos sistemas de tipos 38 ↴ Quando o sistema de tipos do C# 1 não é rico o suficiente? 41 ↴ Resumo das características do sistema de tipos 44*

## 2.3 Tipos de valor e tipos de referência 44

*Valores e referências no mundo real 45 ↴ Fundamentos de valor e tipo de referência 46 ↴ Dissipando mitos 47 ↴ Encaixotamento e desembalagem 49 ↴ Resumo de tipos de valor e tipos de referência 50*

## 2.4 Além do C# 1: novos recursos em uma base sólida 51

*Recursos relacionados a delegados 51 ↴ Recursos relacionados ao sistema de tipos 53 ↴ Recursos relacionados a tipos de valor 55*

## 2.5 Resumo 56

# PARTE 2 C# 2: RESOLVENDO OS PROBLEMAS DO C# 1 .....57

# 3 Digitação parametrizada com genéricos 59

## 3.1 Por que os genéricos são necessários 60

## 3.2 Genéricos simples para uso diário 62

*Aprendendo pelo exemplo: um dicionário genérico 62 ↴ Tipos genéricos e parâmetros de tipo 64 ↴ Métodos genéricos e leitura de declarações genéricas 67*

3.3 Além do básico 70	
	Restrições de tipo 71 ÿ Inferência de tipo para argumentos de tipo de métodos genéricos 76 ÿ Implementando genéricos 77
3.4 Genéricos avançados 83	
	Campos estáticos e construtores estáticos 84 ÿ Como o compilador JIT lida com genéricos 85 ÿ Iteração genérica 87 ÿ Reflexão e genéricos 90
3.5 Limitações de genéricos em C# e outras linguagens 94	
	Falta de variância genérica 94 ÿ Falta de restrições de operador ou uma restrição "numérica" 99 ÿ Falta de propriedades genéricas, indexadores e outros tipos de membros 101 ÿ Comparação com modelos C++ 101 ÿ Comparação com genéricos Java 103
3.6 Resumo 104	
4 Não dizendo nada com tipos anuláveis 105	
4.1 O que você faz quando simplesmente não tem valor? 106	
	Por que variáveis de tipo de valor não podem ser nulas 106
	Padrões para representar valores nulos em C# 1 107
4.2 System.Nullable<T> e System.Nullable 109	
	Apresentando Nullable<T> 109 ÿ Boxing Nullable<T> e unboxing 112 ÿ Igualdade de instâncias Nullable<T> 113
	Suporte da classe Nullable não genérica 114
4.3 Açúcar sintático do C# 2 para tipos anuláveis 114	
	O ? modificador 115 ÿ Atribuindo e comparando com nulo 116
	Conversões e operadores anuláveis 118 ÿ Lógica anulável 121
	Usando o operador as com tipos anuláveis 123 ÿ O operador coalescente nulo 123
4.4 Novos usos de tipos anuláveis 126	
	Tentando uma operação sem usar parâmetros de saída 127
	Comparações indolores com o operador de coalescência nulo 129
4.5 Resumo 131	
5 Delegados acelerados 133	
5.1 Dizendo adeus à estranha sintaxe de delegado 134	
5.2 Conversões de grupos de métodos 136	
5.3 Covariância e contravariância 137	
	Contravariância para parâmetros delegados 138 ÿ Covariância de tipos de retorno delegado 139 ÿ Um pequeno risco de incompatibilidade 141

## 5.4 Ações de delegação inline com métodos anônimos 142

*Começando de forma simples: agindo sobre um parâmetro 142 ↴ Retornando valores de métodos anônimos 145 ↴ Ignorando parâmetros delegados 146*

## 5.5 Capturando variáveis em métodos anônimos 148

*Definindo fechamentos e diferentes tipos de variáveis 148  
Examinando o comportamento das variáveis capturadas 149 ↴ Qual é o objetivo das variáveis capturadas? 151 ↴ A vida útil estendida das variáveis capturadas 152 ↴ Instanciações de variáveis locais 153 Misturas de variáveis compartilhadas e distintas 155 ↴ Diretrizes e resumo das variáveis capturadas 156*

## 5.6 Resumo 158

# 6 **Implementando iteradores da maneira mais fácil 159**

## 6.1 C# 1: A dor dos iteradores manuscritos 160

## 6.2 C# 2: Iteradores simples com instruções de rendimento 163

*Apresentando blocos de iteradores e retorno de rendimento 163 ↴ Visualizando o fluxo de trabalho de um iterador 165 ↴ Fluxo de execução avançado do iterador 167 ↴ Peculiaridades na implementação 170*

## 6.3 Exemplos de iteradores da vida real 172

*Iterando sobre as datas em um horário 172 ↴ Iterando sobre linhas em um arquivo 173 ↴ Filtrando itens preguiçosamente usando um bloco iterador e um predicado 176*

## 6.4 Código pseudossíncrono com o tempo de execução de simultaneidade e coordenação 178

## 6.5 Resumo 180

# 7 **Concluindo C# 2: os recursos finais 182**

## 7.1 Tipos parciais 183

*Criando um tipo com vários arquivos 184 ↴ Usos de tipos parciais 186 ↴ Métodos parciais — somente C# 3! 188*

## 7.2 Classes estáticas 190

## 7.3 Acesso separado à propriedade getter/setter 192

## 7.4 Aliases de namespace 193

*Aliases de namespace qualificados 194 ↴ O alias de namespace global 195 ↴ Aliases externos 196*

**7.5 Diretrizes Pragma 197***Pragmas de aviso 197 ↴ Pragmas de soma de verificação 198***7.6 Buffers de tamanho fixo no código inseguro 199****7.7 Expondo membros internos a assemblies selecionadas 201***Assemblies amigos no caso simples 201 ↴ Por que usar**InternalsVisibleTo? 202 ↴ InternalsVisibleTo e assemblies assinados 203***7.8 Resumo 204****PARTE 3 C# 3: REVOLUCIONANDO O ACESSO A DADOS.....205****8 Cortando coisas com um compilador inteligente 207****8.1 Propriedades implementadas automaticamente 208****8.2 Tipagem implícita de variáveis locais 211***Usando var para declarar uma variável local 211 ↴ Restrições à digitação**implícita 213 ↴ Prós e contras da digitação implícita 214**Recomendações 215***8.3 Inicialização simplificada 216***Definindo alguns tipos de amostra 216 ↴ Definindo propriedades simples 217**Configurando propriedades em objetos incorporados 219 ↴**Inicializadores de coleção 220 ↴ Usos de recursos de inicialização 223***8.4 Matrizes digitadas implicitamente 224****8.5 Tipos anônimos 225***Primeiros encontros do tipo anônimo 225 ↴ Membros de tipos anônimos**227 ↴ Inicializadores de projeção 228 ↴ Qual é o objetivo? 229***8.6 Resumo 231****9 Expressões lambda e árvores de expressão 232****9.1 Expressões lambda como delegados 234***Preliminares: Apresentando os tipos de delegado Func<...> 234**Primeira transformação para uma expressão lambda 235 ↴ Usando uma única**expressão como corpo 236 ↴ Listas de parâmetros digitados implicitamente 236**Atalho para um único parâmetro 237***9.2 Exemplos simples usando List<T> e eventos 238***Filtragem, classificação e ações em listas 238 ↴ Registrando um manipulador**de eventos 240*

## 9.3 Árvores de expressão 241

*Construindo árvores de expressão programaticamente 242* ↗ *Compilando árvores de expressão em delegados 243* ↗ *Convertendo expressões lambda C# em árvores de expressão 244* ↗ *Árvores de expressão no coração do LINQ 248* ↗ *Árvores de expressão além do LINQ 249*

## 9.4 Mudanças na inferência de tipo e resolução de sobrecarga 251

*Razões para mudança: simplificando chamadas de métodos genéricos 252*  
*Tipos de retorno inferidos de funções anônimas 253* ↗ *Inferência de tipo de duas fases 254* ↗ *Escolhendo o método sobrecarregado correto 258*  
*Concluindo inferência de tipo e resolução de sobrecarga 260*

## 9.5 Resumo 260

# 10 Métodos de extensão 262

## 10.1 Vida antes dos métodos de extensão 263

## 10.2 Sintaxe do método de extensão 265

*Declarando métodos de extensão 265* ↗ *Chamando métodos de extensão 267* ↗ *Descoberta de método de extensão 268* ↗ *Chamando um método em uma referência nula 269*

## 10.3 Métodos de extensão no .NET 3.5 271

*Primeiros passos com Enumerable 271* ↗ *Filtrando com Where e encadeando chamadas de métodos 273* ↗ *Interlúdio: não vimos o método Where antes? 275* ↗ *Projeções usando o método Select e tipos anônimos 276* ↗ *Classificando usando o método OrderBy 277* ↗ *Exemplos de negócios envolvendo encadeamento 278*

## 10.4 Ideias e diretrizes de uso 280

*“Estendendo o mundo” e tornando as interfaces mais ricas 280* ↗ *Interfaces fluentes 280* ↗ *Usando métodos de extensão de maneira sensata 282*

## 10.5 Resumo 284

# 11 Expressões de consulta e LINQ to Objects 285

## 11.1 Apresentando o LINQ 286

*Conceitos fundamentais em LINQ 286* ↗ *Definindo o modelo de dados de amostra 291*

## 11.2 Começos simples: selecionando elementos 292

*Começando com uma fonte e terminando com uma seleção 293* ↗ *Traduções do compilador como base para expressões de consulta 293* ↗ *Variáveis de intervalo e projeções não triviais 296* ↗ *Cast, OfType e variáveis de intervalo digitadas explicitamente 298*

### 11.3 Filtrando e ordenando uma sequência 300

*Filtragem usando uma cláusula where 300 ↴ Expressões de consulta degeneradas 301 ↴ Ordenação usando uma cláusula orderby 302*

### 11.4 Cláusulas Let e identificadores transparentes 304

*Apresentando um cálculo intermediário com let 305  
Identificadores transparentes 306*

### 11.5 Junta-se a 307

*Junções internas usando cláusulas de junção 307 ↴ Junções de grupo com cláusulas join...into 311 ↴ Junções cruzadas e sequências de nivelamento usando múltiplas cláusulas from 314*

### 11.6 Agrupamentos e continuações 318

*Agrupando com a cláusula group...by 318 ↴ Consultar continuações 321*

### 11.7 Escolhendo entre expressões de consulta e notação de ponto 324

*Operações que requerem notação de ponto 324 ↴ Expressões de consulta onde a notação de ponto pode ser mais simples 325 ↴ Onde as expressões de consulta brilham 325*

### 11.8 Resumo 326

## 12 LINQ além das coleções 328

### 12.1 Consultando um banco de dados com LINQ to SQL 329

*Primeiros passos: o banco de dados e o modelo 330 ↴ Consultas iniciais 332 ↴ Consultas envolvendo junções 334*

### 12.2 Traduções usando IQueryable e IQueryProvider 336

*Apresentando IQueryable<T> e interfaces relacionadas 337 ↴ Fingindo: implementações de interface para registrar chamadas 338 ↴ Unindo expressões: os métodos de extensão Queryable 341 ↴ O provedor de consulta falso em ação 342 ↴ Concluindo IQueryable 344*

### 12.3 APIs compatíveis com LINQ e LINQ to XML 344

*Tipos principais em LINQ to XML 345 ↴ Construção declarativa 347  
Consultas em nós únicos 349 ↴ Operadores de consulta nivelados 351  
Trabalhando em harmonia com LINQ 352*

### 12.4 Substituindo LINQ por objetos com LINQ 353 paralelo

*Traçando o conjunto de Mandelbrot com um único thread 353 ↴ Apresentando ParallelEnumerable, ParallelQuery e AsParallel 354  
Ajustando consultas paralelas 356*

## 12.5 Invertendo o modelo de consulta com LINQ to Rx 357

*IObservable<T> e IObserver<T>* 358 ↴ *Começando de forma simples (de novo)* 360 ↴ *Consultando observáveis* 360 ↴ *Qual é o objetivo?* 363

## 12.6 Estendendo LINQ para Objetos 364

*Diretrizes de design e implementação* 364 ↴ *Extensão da amostra: selecionando um elemento aleatório* 365

## 12.7 Resumo 367

# PARTE 4 C# 4: JOGANDO BEM COM OS OUTROS .....369

## 13 Pequenas alterações para simplificar o código 371

### 13.1 Parâmetros opcionais e argumentos nomeados 372

*Parâmetros opcionais* 372 ↴ *Argumentos nomeados* 378 ↴ *Juntando os dois* 382

### 13.2 Melhorias para interoperabilidade COM 387

*Os horrores da automação do Word antes do C# 4* 387 ↴ *A vingança dos parâmetros opcionais e argumentos nomeados* 388 ↴ *Quando um parâmetro ref não é um parâmetro ref?* 389 ↴ *Chamando indexadores nomeados* 390 ↴ *Vinculando assemblies de interoperabilidade primários* 391

### 13.3 Variação genérica para interfaces e delegados 394

*Tipos de variação: covariância e contravariância* 394 ↴ *Usando variação em interfaces* 396 ↴ *Usando variação em delegados* 399  
*Situações complexas* 399 ↴ *Restrições e notas* 401

### 13.4 Pequenas mudanças em eventos de bloqueio e semelhantes a campos 405

*Bloqueio robusto* 405 ↴ *Mudanças em eventos semelhantes a campos* 406

### 13.5 Resumo 407

## 14 Vinculação dinâmica em uma linguagem estática 409

### 14.1 O quê? Quando? Por que? Como? 411

*O que é digitação dinâmica?* 411 ↴ *Quando a digitação dinâmica é útil e por quê?* 412 ↴ *Como o C# 4 fornece digitação dinâmica?* 413

### 14.2 O guia de cinco minutos para 414 dinâmico

### 14.3 Exemplos de digitação dinâmica 416

*COM em geral e Microsoft Office em particular* 417  
 ↴ *Linguagens dinâmicas como IronPython* 419  
*Digitação dinâmica em código puramente gerenciado* 423

**14.4 Olhando os bastidores 429**

*Apresentando o Dynamic Language Runtime 429* ↗ *Conceitos básicos do DLR 431* ↗ *Como o compilador C# lida com dinâmica 434*  
*O compilador C# fica ainda mais inteligente 438* ↗ *Restrições ao código dinâmico 441*

**14.5 Implementando comportamento dinâmico 444**

*Usando ExpandoObject 444* ↗ *Usando DynamicObject 448*  
*Implementando IDynamicMetaObjectProvider 455*

**14.6 Resumo 459****PARTE 5 C# 5: ASSINCRONIA SIMPLES .....461****15 Assincronia com async/await 463****15.1 Apresentando funções assíncronas 465**

*Primeiros encontros do tipo assíncrono 465* ↗ *Detalhando o primeiro exemplo 467*

**15.2 Pensando em assincronia 468**

*Fundamentos da execução assíncrona 468* ↗ *Modelagem de métodos assíncronos 471*

**15.3 Sintaxe e semântica 472**

*Declarando um método assíncrono 472* ↗ *Tipos de retorno de métodos assíncronos 473* ↗ *O padrão awaitable 474* ↗ *O fluxo de expressões de espera 477* ↗ *Retornando de um método assíncrono 481*  
*Exceções 482*

**15.4 Funções anônimas assíncronas 490****15.5 Detalhes de implementação: transformação do compilador 492**

*Visão geral do código gerado 493* ↗ *Estrutura do método esqueleto 495*  
↗ *Estrutura da máquina de estado 497* ↗ *Um ponto de entrada para governar todos eles 498* ↗ *Controle em torno de expressões de espera 500* ↗ *Acompanhar uma pilha 501* ↗ *Descobrindo mais 503*

**15.6 Usando async/await efetivamente 503**

*O padrão assíncrono baseado em tarefas 504* ↗ *Composição de operações assíncronas 507* ↗ *Código assíncrono de teste de unidade 511*  
*O padrão aguardável redux 515* ↗ *Operações assíncronas em WinRT 516*

**15.7 Resumo 517**

## 16 Recursos bônus do C# 5 e considerações finais 519

16.1 Mudanças nas variáveis capturadas em loops foreach 520

16.2 Atributos de informações do chamador 520

*Comportamento básico 521 ↗ Registro em log 522 ↗*

*Implementação de INotifyPropertyChanged 523 ↗ Uso de atributos de informações do chamador sem .NET 4.5 524*

16.3 Considerações finais 525

*apêndice A Operadores de consulta padrão LINQ 527*

*apêndice B Coleções genéricas no .NET 540*

*apêndice C Resumos da versão 554*

*índice 563*

---

## prefácio

Existem dois tipos de pianistas.

Há alguns pianistas que tocam, não porque gostem, mas porque os pais os obrigam a ter aulas. Depois há aqueles que tocam piano porque lhes agrada criar música. Eles não precisam ser forçados; pelo contrário, às vezes não sabem quando parar.

Deste último tipo, há alguns que tocam piano como hobby. Depois, há aqueles que jogam para viver. Isso requer um nível mais alto de dedicação, habilidade e talento.

Eles podem ter algum grau de liberdade sobre o gênero musical que tocam e as escolhas estilísticas que fazem ao tocá-lo, mas fundamentalmente essas escolhas são motivadas pelas necessidades do empregador ou pelos gostos do público.

Deste último tipo, há alguns que o fazem principalmente por dinheiro. Depois, há aqueles profissionais que gostariam de tocar piano em público mesmo que não fossem pagos. Eles gostam de usar suas habilidades e talentos para fazer música para outras pessoas. Que eles possam se divertir e serem pagos por isso é tanto melhor.

Deste último tipo, há alguns que são autodidatas, que tocam de ouvido, que podem ter grande talento e habilidade, mas não conseguem comunicar essa compreensão intuitiva aos outros, exceto através da própria música. Depois, há aqueles que têm formação formal tanto na teoria como na prática. Podem explicar que técnicas o compositor utilizou para alcançar o efeito emocional pretendido e utilizar esse conhecimento para moldar a sua interpretação da peça.

Deste último tipo, há alguns que nunca olharam para dentro dos seus pianos. Depois, há aqueles que ficam fascinados pelos escapamentos inteligentes que levantam os feltros amortecedores uma fração de segundo antes de os martelos atingirem as cordas. Eles possuem niveladores princip

e chaves de cabrestante. Eles ficam encantados e orgulhosos por serem capazes de compreender os mecanismos de um instrumento que possui de 5 a 10.000 peças móveis.

Deste último tipo, há alguns que se contentam em dominar o seu ofício e exercitar os seus talentos pelo prazer e lucro que isso traz. Depois há aqueles que não são apenas artistas, teóricos e técnicos; de alguma forma, eles encontram tempo para transmitir esse conhecimento a outros como mentores.

Não tenho ideia se Jon Skeet é pianista ou músico de algum tipo. Mas pelas minhas conversas por e-mail com ele como um dos profissionais mais valiosos da equipe C# ao longo do anos, lendo seu blog e lendo cada palavra de cada um de seus livros pelo menos três vezes, ficou claro para mim que Jon é esse último tipo de desenvolvedor de software: entusiasta, condescendente, talentoso, curioso, analítico - e um professor dos outros.

C# é uma linguagem altamente pragmática e em rápida evolução. Através da adição de compreensão de consulta, inferência de tipos mais rica, uma sintaxe compacta para funções anônimas e assim por diante, espero que tenhamos possibilitado um estilo totalmente novo de programação, ao mesmo tempo em que permanecemos fiéis à abordagem estaticamente tipada e orientada a componentes que tornou o C# um sucesso.

Muitos desses novos elementos estilísticos têm a qualidade paradoxal de parecerem muito antigos (as expressões lambda remontam aos fundamentos da ciência da computação na primeira metade do século XX) e, ao mesmo tempo, parecerem novos e desconhecidos para os desenvolvedores acostumados a um estilo mais abordagem moderna orientada a objetos.

Jon entende tudo isso. Este livro é ideal para desenvolvedores profissionais que precisam entender o *quê* e *como* da última revisão do C#. Mas é também para aqueles que desenvolvem operações cuja compreensão é enriquecida pela exploração do *porquê* dos princípios de design da linguagem.

Ser capaz de aproveitar todo esse novo poder requer novas formas de pensar sobre dados, funções e o relacionamento entre eles. Não é diferente de tentar tocar jazz depois de anos de treinamento clássico – ou vice-versa. De qualquer forma, estou ansioso para descobrir que tipos de composições funcionais a próxima geração de programadores de C# criará. Boa composição é obrigado por escolher a chave de C# para fazer isso.

ÉRIC LIPPERT  
ARQUITETO DE ANÁLISE C#  
COBERTURA

---

## prefácio

Oh garoto. Ao escrever este prefácio, comecei com o prefácio da segunda edição, que começou dizendo quanto tempo passou desde que escrevi o prefácio da primeira edição.

A segunda edição é agora uma memória distante, e a primeira edição parece uma vida totalmente diferente. Não tenho certeza se isso diz mais sobre o ritmo da vida moderna ou sobre minha memória, mas, de qualquer forma, é um pensamento preocupante.

O panorama do desenvolvimento mudou enormemente desde a primeira edição, e mesmo desde a segunda. Isto tem sido impulsionado por muitos fatores, sendo o aumento dos dispositivos móveis provavelmente o mais óbvio. Mas muitos desafios permaneceram os mesmos. Ainda é difícil escrever aplicações adequadamente internacionalizadas. Ainda é difícil lidar com erros normalmente em todas as situações. Ainda é bastante difícil escrever aplicativos multithread corretos, embora essa tarefa tenha se tornado significativamente mais simples graças às melhorias na linguagem e na biblioteca ao longo dos anos.

Mais importante ainda, no contexto deste prefácio, acredito que os desenvolvedores ainda precisam conhecer a linguagem que estão usando em um nível em que tenham confiança em como ela se comportará. Eles podem não conhecer os detalhes de cada chamada de API que estão usando, ou mesmo alguns dos casos obscuros da linguagem que por acaso não usam,<sup>1</sup> mas o núcleo da linguagem deve parecer um amigo sólido que o desenvolvedor pode confiar para se comportar de maneira previsível.

Além da letra do idioma que você está desenvolvendo, acredito que há um grande benefício em compreender seu espírito. Embora você possa ocasionalmente descobrir que tem uma briga nas mãos, por mais que tente, se tentar fazer seu código funcionar da maneira que os designers da linguagem pretendiam, sua experiência será muito mais agradável.

---

<sup>1</sup> Tenho uma confissão a fazer: sei muito pouco sobre códigos e ponteiros inseguros em C#. Eu simplesmente nunca precisei para descobrir mais sobre eles.

## agradecimentos

---

Você poderia esperar que montar uma terceira edição – e uma onde a principal mudança consiste em dois novos capítulos – seria simples. Na verdade, escrever o conteúdo do “campo verde” dos capítulos 15 e 16 foi a parte fácil. Mas há muito mais do que isso - ajustar pequenos trechos de linguagem ao longo do resto do livro, verificar quaisquer aspectos que estavam bem há alguns anos, mas que não fazem muito sentido agora, e geralmente garantir que todo o livro esteja correto. está de acordo com os altos padrões que espero que os leitores sigam. Felizmente, tive a sorte de ter um grande grupo de pessoas me apoiando e mantendo o livro no caminho certo.

Mais importante ainda, minha família tem sido maravilhosa como sempre. Minha esposa, Holly, também é autora infantil, então nossos filhos estão acostumados com o fato de termos que nos trancar por um tempo para cumprir os prazos editoriais, mas eles permaneceram alegremente encorajadores o tempo todo. A própria Holly leva tudo isso com calma, e sou grato por ela nunca ter me lembrado de quantos livros ela começou do zero e concluiu durante o tempo em que estou trabalhando nesta terceira edição.

Os revisores formais são listados mais adiante, mas gostaria de acrescentar uma nota de agradecimento pessoal a todos aqueles que solicitaram cópias de acesso antecipado desta terceira edição, encontrando erros de digitação e sugerindo alterações... também perguntando constantemente quando o livro estava chegando. fora. O próprio fato de eu ter leitores ansiosos para colocar as mãos no livro finalizado foi uma enorme fonte de encorajamento.

Sempre me dou bem com a equipe da Manning e tem sido um prazer trabalhar com alguns amigos familiares da primeira edição, bem como com os recém-chegados. Mike Stephens e Jeff Bleiel guiaram todo o processo sem problemas, enquanto decidimos o que mudar em relação às edições anteriores e o que manter. Eles geralmente colocam tudo em

a forma certa. Andy Carroll e Katie Tennant forneceram edição especializada e revisando, respectivamente, nunca expressando irritação com meu inglês, seletividade ou perplexidade geral. A equipe de produção fez sua mágica no histórico, como sempre, mas mesmo assim sou grato a eles: Dottie Marsico, Janet Vail, Marija Tudor e Mary Piergies. Por fim, gostaria de agradecer à editora, Marjan Bace, por me permitir uma terceira edição e explorar algumas opções futuras interessantes.

A revisão por pares é imensamente importante, não apenas para obter os detalhes técnicos o livro certo, mas também o equilíbrio e o tom. Às vezes, os comentários que recebemos apenas moldaram o livro como um todo; em outros casos, fiz alterações muito específicas em resposta. De qualquer forma, todos os comentários foram bem-vindos. Então, obrigado aos seguintes revisores por tornarem o livro melhor para todos nós: Andy Kirsch, Bas Pennings, Bret Colloff, Charles M. Gross, Dror Helper, Dustin Laine, Ivan Todorovic', Jon Parish, Sebastian Martín Aguilar, Tiaan Geldenhuys e Timo Bredenoort.

Gostaria de agradecer particularmente a Stephen Toub e Stephen Cleary, cujos primeiros as revisões do capítulo 15 foram inestimáveis. Assincronia é um tópico particularmente complicado de escrever sobre isso de forma clara, mas precisa, e seus conselhos de especialistas fizeram uma diferença muito significativa para o capítulo.

Sem a equipe C#, este livro não teria causa para existir, é claro. Sua dedicação à linguagem em design, implementação e testes é exemplar, e eu vejo ansioso para ver o que eles farão a seguir. Desde que a segunda edição foi publicada, Eric Lippert deixou a equipe de C# para uma nova e fabulosa aventura, mas estou imensamente grato por ele ainda ter sido capaz de atuar como revisor técnico desta terceira edição. EU agradecer-lhe também pelo prefácio que ele escreveu originalmente para a primeira edição e que é incluído novamente desta vez. Refiro-me aos pensamentos de Eric sobre vários assuntos ao longo do livro, e se você ainda não está lendo o blog dele (<http://ericlippert.com>), você realmente deveria estar.

# *sobre este livro*

---

Este é um livro sobre C# da versão 2 em diante — é simples assim. Eu mal abordo o C# 1 e apenas abordo as bibliotecas do .NET Framework e o Common Language Runtime (CLR) quando eles estão relacionados à linguagem. Essa é uma decisão deliberada e o resultado é um livro bem diferente da maioria dos livros sobre C# e .NET que já vi.

Ao presumir um conhecimento razoável de C# 1, evito gastar centenas de páginas cobrindo material que acho que a maioria das pessoas já entende. Isso me dá espaço para expandir os detalhes das versões posteriores do C#, e é por isso que espero que você esteja lendo o livro. Quando escrevi a primeira edição deste livro, até mesmo o C# 2 era relativamente desconhecido para alguns leitores. Até agora, quase todos os desenvolvedores de C# têm alguma experiência com os recursos introduzidos no C# 2, mas ainda mantive esse material nesta edição, pois ele é fundamental para o que virá depois.

## **Quem deveria ler esse livro?**

Este livro é direcionado diretamente a desenvolvedores que já conhecem um pouco de C#. Para obter o valor máximo absoluto, você conhece bem o C# 1, mas sabe muito pouco sobre versões posteriores. Não há mais muitos leitores nesse ponto ideal, mas acredito que ainda há muitos desenvolvedores que podem se beneficiar se aprofundarem no C# 2 e 3, mesmo que já os utilizem há algum tempo... e muitos desenvolvedores ainda não usaram C# 4 ou 5.

Se você não conhece C#, provavelmente este não é o livro para você. Você poderia lutar, procurando aspectos com os quais não está familiarizado, mas não seria uma forma muito eficiente de aprender. Seria melhor começar com um livro diferente e, em seguida, adicionar gradualmente *C# in Depth* à mistura. Há uma grande variedade de livros que cobrem C#

do zero, em muitos estilos diferentes. A série C# in a Nutshell (O'Reilly) sempre tem sido bom nesse aspecto, e Essential C# 5.0 (Addison-Wesley Professional) também é um boa introdução.

Não vou afirmar que a leitura deste livro fará de você um programador fabuloso. Há muito mais na engenharia de software do que conhecer a sintaxe da linguagem que você está usando. Dou algumas palavras de orientação, mas em última análise há uma muito mais instinto no desenvolvimento do que a maioria de nós gostaria de admitir. O que eu vou afirmação é que se você ler e compreender este livro, você deverá se sentir confortável com C# e livre para seguir seus instintos sem muita apreensão. Não é sobre ser capaz de escrever códigos que ninguém mais entenderá porque utiliza cantos desconhecidos da linguagem; trata-se de ter certeza de que você conhece as opções disponíveis para você e saiba qual caminho os idiomas C# estão incentivando você a seguir.

### Roteiro

A estrutura do livro é simples. São cinco partes e três apêndices. O primeiro Essa parte serve como uma introdução, incluindo uma atualização sobre tópicos em C# 1 que são importantes para a compreensão de versões posteriores da linguagem e que muitas vezes são mal compreendidos. A segunda parte cobre os novos recursos introduzidos no C# 2, a terceira parte cobre C# 3 e assim por diante.

Há ocasiões em que organizar o material desta forma significa que voltaremos a um tópico algumas vezes - em particular, os delegados são melhorados em C# 2 e depois novamente em C# 3 - mas há método na minha loucura. Prevejo que vários leitores usarão versões diferentes para projetos diferentes; por exemplo, você pode estar usando C# 4 no trabalho, mas experimentando C# 5 em casa. Isso significa que é útil esclarecer o que está em qual versão. Também proporciona uma sensação de contexto e evolução - mostra como a linguagem se desenvolveu ao longo do tempo.

O Capítulo 1 define o cenário pegando um trecho simples de código C# 1 e evoluindo-o, vendo como versões posteriores permitem que o código-fonte se torne mais legível e poderoso. Bem observe o contexto histórico em que o C# cresceu e o contexto técnico em que opera como parte de uma plataforma completa; C# como linguagem baseia-se em bibliotecas de estrutura e em um poderoso tempo de execução para transformar abstração em realidade.

O Capítulo 2 analisa o C# 1 e três aspectos específicos: delegados, as características do sistema de tipos e as diferenças entre tipos de valor e tipos de referência. Esses tópicos são frequentemente compreendidos "bem o suficiente" pelos desenvolvedores de C# 1, mas como evoluiu e os desenvolveu significativamente, é necessária uma base sólida para para aproveitar ao máximo os novos recursos.

O Capítulo 3 aborda o maior recurso do C# 2 e potencialmente o mais difícil de entender: genéricos. Métodos e tipos podem ser escritos genericamente, com parâmetros de tipo em pé in para tipos reais especificados no código de chamada. Inicialmente é tão confuso quanto isso descrição faz parecer, mas depois de entender os genéricos, você se perguntará como sobreviveu sem eles.

Se você sempre quis representar um número inteiro nulo, o capítulo 4 é para você. Ele introduz tipos anuláveis: um recurso, construído em genéricos, que aproveita o suporte na linguagem, no tempo de execução e na estrutura.

O Capítulo 5 mostra as melhorias nos delegados no C# 2. Até agora, você pode ter usado delegados apenas para lidar com eventos como cliques em botões. O C# 2 facilita a criação de delegados e o suporte da biblioteca os torna mais úteis para outras situações além de eventos.

No capítulo 6 examinaremos iteradores e a maneira fácil de implementá-los em C# 2.

Poucos desenvolvedores usam blocos iteradores, mas como o LINQ to Objects é construído sobre iteradores, eles se tornarão cada vez mais importantes. A natureza preguiçosa de sua execução também é uma parte fundamental do LINQ.

O Capítulo 7 mostra vários recursos menores introduzidos no C# 2, cada um tornando a vida um pouco mais agradável. Os designers da linguagem suavizaram alguns pontos difíceis no C# 1, permitindo uma interação mais flexível com geradores de código, melhor suporte para classes utilitárias, acesso mais granular às propriedades e muito mais.

O Capítulo 8 mais uma vez examina alguns recursos relativamente simples — mas desta vez em C# 3. Quase toda a nova sintaxe é voltada para o objetivo comum do LINQ, mas os blocos de construção também são úteis por si só. Com tipos anônimos, propriedades implementadas automaticamente, variáveis locais digitadas implicitamente e suporte de inicialização bastante aprimorado, o C# 3 fornece uma linguagem muito mais rica com a qual seu código pode expressar seu comportamento.

O Capítulo 9 analisa o primeiro tópico importante do C# 3 – expressões lambda. Não satisfeitos com a sintaxe razoavelmente concisa discutida no capítulo 5, os designers da linguagem tornaram os delegados ainda mais fáceis de criar do que no C# 2. Lambdas são capazes de mais — eles podem ser convertidos em árvores de expressão, uma forma poderosa de representar código como dados.

No capítulo 10 examinaremos os métodos de extensão, que fornecem uma maneira de enganar o compilador fazendo-o acreditar que os métodos declarados em um tipo na verdade pertencem a outro. À primeira vista, isso parece ser um pesadelo de legibilidade, mas com consideração cuidadosa pode ser um recurso extremamente poderoso – e vital para o LINQ.

O Capítulo 11 combina os três capítulos anteriores na forma de expressões de consulta, uma forma concisa, mas poderosa, de consultar dados. Inicialmente nos concentraremos em LINQ to Objects, mas você verá como o padrão de expressão de consulta é aplicado de uma forma que permite que outros provedores de dados se conectem perfeitamente.

O Capítulo 12 é um rápido tour pelos vários usos diferentes do LINQ. Primeiro, veremos os benefícios das expressões de consulta combinadas com árvores de expressão — como o LINQ to SQL é capaz de converter o que parece ser C# normal em instruções SQL. Em seguida, veremos como as bibliotecas podem ser projetadas para combinar bem com o LINQ, tomando o LINQ to XML como exemplo. Parallel LINQ e Reactive Extensions mostram duas abordagens alternativas para consultas em processo, e o capítulo termina com uma discussão sobre como você pode estender LINQ to Objects com seus próprios operadores LINQ.

A cobertura do C# 4 começa no capítulo 13, onde veremos argumentos nomeados e parâmetros opcionais, melhorias na interoperabilidade COM e variação genérica. De certa forma, esses são recursos muito separados, mas argumentos nomeados e parâmetros opcionais contribuem para a interoperabilidade COM , bem como para habilidades mais específicas que só estão disponíveis ao trabalhar com objetos COM .

O Capítulo 14 descreve o maior recurso do C# 4: tipagem dinâmica. A capacidade de vincular membros dinamicamente em tempo de execução, em vez de estaticamente em tempo de compilação, é um grande avanço para o C#, mas é aplicada seletivamente – apenas o código que envolve um valor dinâmico será executado dinamicamente.

O Capítulo 15 é sobre assincronia. O C# 5 contém apenas um recurso importante: a capacidade de escrever funções assíncronas. Esse único recurso é ao mesmo tempo extremamente complicado de entender completamente e incrivelmente elegante de usar. Finalmente, podemos escrever código assíncrono que não pareça espaguete.

Encerraremos no capítulo 16 com os recursos restantes do C# 5 (ambos minúsculos) e algumas reflexões sobre o futuro.

Os apêndices são todos materiais de referência. No apêndice A, abordo os operadores de consulta padrão LINQ , com alguns exemplos. O Apêndice B examina as principais classes e interfaces genéricas da coleção. O Apêndice C fornece uma breve visão das diferentes versões do .NET, incluindo os diferentes sabores, como Compact Framework e Silverlight.

### Terminologia, tipografia e downloads

A maior parte da

terminologia do livro é explicada à medida que avança, mas há algumas definições que valem a pena destacar aqui. Eu uso C# 1, C# 2, C# 3, C# 4 e C# 5 de uma maneira razoavelmente óbvia — mas você pode ver outros livros e sites referindo-se a C# 1.0, C# 2.0, C# 3.0, C# 4.0 e C# 5.0. O “.0” extra parece redundante para mim, e é por isso que o omiti – espero que o significado esteja claro.

Apropriei-me de alguns termos de um livro de C# de Mark Michaelis. Para evitar a confusão entre o *tempo de execução* ser um ambiente de execução (como em “o Common Language Runtime”) e um ponto no tempo (como em “a substituição ocorre em tempo de execução”), Mark usa o *tempo de execução* para o último conceito, geralmente em comparação com *tempo de compilação*. Esta parece-me ser uma ideia completamente sensata e que espero que seja difundida na comunidade em geral. Estou fazendo minha parte seguindo o exemplo dele neste livro.

Freqüentemente me refiro à “especificação da linguagem” ou apenas à “especificação” – a menos que eu indique o contrário, isso significa a especificação da linguagem C#. Entretanto, múltiplas versões da especificação estão disponíveis, em parte devido às diferentes versões da própria linguagem e em parte devido ao processo de padronização. Todos os números de seção fornecidos são da especificação da linguagem C# 5.0 da Microsoft.

Este livro contém vários trechos de código, que aparecem em uma fonte de largura fixa como esta; a saída das listagens aparece da mesma maneira. As anotações de código acompanham algumas listagens e, outras vezes, seções específicas do código são mostradas em negrito para destacar uma alteração, melhoria ou adição. Quase todo o código

aparece em forma de snippet, permitindo que ele permaneça compacto, mas ainda executável - dentro do ambiente certo. Esse ambiente é o Snippy, uma ferramenta personalizada introduzida em seção 1.8. Snippy está disponível para download, junto com todo o código do livro (na forma de snippets, soluções completas do Visual Studio ou, mais frequentemente, ambos) do site do livro em [csharpinprofundidade.com](http://csharpinprofundidade.com), bem como no site da editora em [man-ning.com/CSharpinDepthThirdEdition](http://man-ning.com/CSharpinDepthThirdEdition).

## Author Online e o site C# in Depth

A compra do *C# in Depth, Third Edition* inclui acesso gratuito a um fórum privado na web executado da Manning Publications, onde você pode fazer comentários sobre o livro, fazer perguntas técnicas e receber ajuda do autor e de outros usuários. Para acessar o fórum e assine-o, aponte seu navegador para [www.manning.com/CSharpinDepth-ThirdEdition](http://www.manning.com/CSharpinDepth-ThirdEdition). Esta página fornece informações sobre como entrar no fórum depois de estarem registrados, que tipo de ajuda está disponível e as regras de conduta no fórum.

O fórum Author Online e os arquivos de discussões anteriores estarão acessíveis disponível no site da editora enquanto o livro estiver sendo impresso.

Além do próprio site de Manning, criei um site complementar para o livro em [csharpinprofundidade.com](http://csharpinprofundidade.com), contendo informações que não se enquadram no livro, código-fonte para download de todas as listagens do livro e links para outros recursos.

## Sobre o autor

---

Não sou um desenvolvedor C# típico, acho justo dizer. Nos últimos cinco anos, quase todo o meu tempo trabalhando com C# foi para diversão – na verdade, como um hobby um tanto obsessivo. No trabalho, tenho escrito Java do lado do servidor no Google London e posso afirmar com segurança que poucas coisas ajudam você a apreciar mais os novos recursos da linguagem do que ter que codificar em uma linguagem que não os possui, mas que é semelhante o suficiente para lembrá-lo de sua ausência.

Tentei manter contato com o que outros desenvolvedores acham difícil sobre C#, mantendo um olhar atento sobre Stack Overflow, postando curiosidades em meu blog e, ocasionalmente, falando sobre C# e tópicos relacionados em qualquer lugar que forneça às pessoas para me ouvir. Além disso, estou desenvolvendo ativamente uma API de data e hora .NET de código aberto chamada Noda Time ([consulte `http://nodatime.org`](http://nodatime.org)). Resumindo, C# ainda corre em minhas veias com mais força do que nunca.

Apesar de todas essas estranhezas - e apesar do meu sempre surpreendente status de microcelebridade devido ao Stack Overflow - sou um desenvolvedor muito comum em muitos outros aspectos. Eu escrevo muitos códigos que me fazem fazer uma careta quando volto a eles. Meus testes unitários nem sempre vêm primeiro... e às vezes eles nem existem. Cometo erros pontuais de vez em quando. A seção de inferência de tipo da especificação C# ainda me confunde, e há alguns usos de curingas Java que me fazem querer descansar um pouco. Sou um programador profundamente falho.

É assim que deve ser. Nas próximas centenas de páginas, tentarei fingir o contrário: defenderei as melhores práticas como se eu mesmo sempre as tivesse seguido e desaprovarei atalhos sujos como se nunca sonhasse em segui-los. Não acredite em uma palavra disso. A verdade é que provavelmente sou igual a você. Acontece que eu sei um pouco mais sobre como o C# funciona, só isso... e mesmo esse estado de coisas só durará até você terminar o livro.

## sobre a ilustração da capa

---

A legenda da ilustração na capa de C# in Depth, Third Edition é “Musician”. A ilustração foi retirada de uma coleção de trajes do Império Otomano publicada em 1º de janeiro de 1802, por William Miller, da Old Bond Street, Londres. A página de título está faltando na coleção e não conseguimos localizá-la até o momento. O índice do livro identifica as figuras em inglês e francês, e cada ilustração traz os nomes de dois artistas que trabalharam nela, ambos os quais sem dúvida ficariam surpresos ao encontrar sua arte enfeitando a capa de um livro de programação de computador. ...duzentos anos depois.

A coleção foi comprada por um editor da Manning em um mercado de antiguidades na “Garage” na West 26th Street, em Manhattan. O vendedor era um americano radicado em Ancara, na Turquia, e a transação ocorreu no momento em que ele arrumava seu estande para o dia. O editor de Manning não tinha consigo a quantia substancial de dinheiro necessária para a compra e um cartão de crédito e um cheque foram educadamente recusados. Com o vendedor voltando para Ancara naquela noite, a situação estava ficando desesperadora. Qual foi a solução? Acabou sendo nada mais do que um acordo verbal antiquado selado com um aperto de mão. O vendedor simplesmente propôs que o dinheiro lhe fosse transferido por transferência bancária e o editor saiu com os dados bancários num pedaço de papel e o portfólio de imagens debaixo do braço. Escusado será dizer que transferimos os fundos no dia seguinte e continuamos gratos e impressionados com a confiança desta pessoa desconhecida em um de nós. Lembra algo que pode ter acontecido há muito tempo.

Nós da Manning celebramos a inventividade, a iniciativa e, sim, a diversão do negócio da informática com capas de livros baseadas na rica diversidade da vida regional de dois séculos atrás, trazidas de volta à vida pelas fotos desta coleção.

## Parte 1

# *Preparando-se para a viagem*

Cada leitor chegará a este livro com um conjunto diferente de expectativas e um nível diferente de experiência. Você é um especialista procurando preencher algumas lacunas, ainda que pequenas, em seu conhecimento atual? Talvez você se considere um desenvolvedor comum, com um pouco de experiência no uso de expressões genéricas e lambda, mas com desejo de entender melhor como elas funcionam. Talvez você esteja razoavelmente confiante com C# 2 e 3, mas não tenha experiência com C# 4 ou 5.

Como autor, não posso fazer com que todos os leitores sejam iguais – e não gostaria de fazer isso, mesmo que pudesse. Mas espero que todos os leitores tenham duas coisas em comum: o desejo de um relacionamento mais profundo com C# como linguagem e pelo menos um conhecimento básico de C# 1. Se você puder trazer esses elementos para a festa, eu fornecerei o resto.

A gama potencialmente enorme de níveis de habilidade é a principal razão pela qual esta parte do livro existe. Talvez você já saiba o que esperar das versões posteriores do C# — ou tudo pode ser totalmente novo para você. Você pode ter um conhecimento sólido do C# 1 ou pode estar enferrujado em alguns detalhes — alguns dos quais se tornarão cada vez mais importantes à medida que você aprender sobre as versões posteriores. Ao final da parte 1, não terei nivelado totalmente o campo de jogo, mas você deverá ser capaz de abordar o resto do livro com confiança e uma ideia do que está por vir.

Nos dois primeiros capítulos, olharemos para frente e para trás. Um dos temas principais do livro é a evolução. Antes de introduzir qualquer recurso na linguagem, a equipe de design do C# considera cuidadosamente esse recurso no contexto do que já está presente e dos objetivos gerais para o futuro. Isso traz uma sensação de consistência à linguagem mesmo em meio a mudanças. Para entender como e por que a linguagem está evoluindo, você precisa ver de onde ela vem e para onde vai.

O Capítulo 1 apresenta uma visão panorâmica do restante do livro, dando uma breve olhada em alguns dos maiores recursos do C# além da versão 1. Mostrarei uma progressão do código do C# 1 em diante, aplicando novos recursos um por um até que o código fique quase irreconhecível desde seu início humilde. Também veremos parte da terminologia que usarei no restante do livro, bem como o formato do código de exemplo.

O Capítulo 2 é fortemente focado em C# 1. Se você for um especialista em C# 1, poderá pular este capítulo, mas ele aborda algumas das áreas do C# 1 que tendem a ser mal compreendidas. Em vez de tentar explicar toda a linguagem, o capítulo concentra-se nos recursos que são fundamentais para as versões posteriores do C#. A partir dessa base sólida, você pode seguir em frente e examinar C# 2 na parte 2 do livro.

# A face mutável do desenvolvimento em C#

## Este capítulo cobre

- ÿ Um exemplo em evolução
- ÿ A composição do .NET
- ÿ Usando o código deste livro
- ÿ A especificação da linguagem C#

Você sabe o que eu realmente gosto em linguagens dinâmicas como Python, Ruby e Groovy? Eles sugam o conteúdo do seu código, deixando apenas a essência dele – os bits que realmente *fazem* alguma coisa. A formalidade tediosa dá lugar a recursos como geradores, expressões lambda e compreensão de lista.

O interessante é que poucos dos recursos que tendem a dar às linguagens dinâmicas uma sensação de leveza têm alguma coisa a ver com serem dinâmicas. Alguns o fazem, é claro — digitação rápida e um pouco da magia usada no Active Record, por exemplo — mas linguagens estaticamente digitadas não precisam ser desajeitadas e pesadas.

Insira C#. De certa forma, o C# 1 poderia ter sido visto como uma versão melhor da linguagem Java, por volta de 2001. As semelhanças eram muito claras, mas o C# tinha alguns extras: propriedades como um recurso de primeira classe na linguagem, delegados e eventos , cada

loops, instruções de uso , substituição explícita de métodos, sobrecarga de operadores e tipos de valores personalizados, para citar alguns. Obviamente, a preferência de linguagem é uma questão pessoal, mas o C# 1 definitivamente parecia um avanço em relação ao Java quando comecei a usá-lo.

Desde então, as coisas só melhoraram. Cada nova versão do C# adicionou recursos significativos para reduzir a angústia do desenvolvedor, mas sempre de uma forma cuidadosamente considerada e com pouca incompatibilidade com versões anteriores. Mesmo antes do C# 4 ganhar a capacidade de usar digitação dinâmica onde for genuinamente útil, muitos recursos tradicionalmente associados a linguagens dinâmicas e funcionais foram incluídos no C#, levando a um código mais fácil de escrever e manter. Da mesma forma, embora os recursos em torno da assincronia no C# 5 não sejam exatamente os mesmos do F#, parece-me que há uma influência definitiva.

Neste livro, mostrarei essas mudanças uma por uma, com detalhes suficientes para que você se sinta confortável com alguns dos milagres que o compilador C# está preparado para realizar em seu nome. Porém, tudo isso virá depois – neste capítulo, examinarei todos os recursos que puder, sem respirar. Definirei o que quero dizer quando falo sobre C# como linguagem em comparação com .NET como plataforma e oferecerei algumas notas importantes sobre o código de exemplo no restante do livro. Então podemos mergulhar nos detalhes.

Não veremos *todas* as alterações feitas em C# neste único capítulo, mas você verá genéricos, propriedades com diferentes modificadores de acesso, tipos anuláveis, métodos anônimos, propriedades implementadas automaticamente, inicializadores de coleção aprimorados, inicializadores de objeto aprimorados, lambda expressões, métodos de extensão, digitação implícita, expressões de consulta LINQ , argumentos nomeados, parâmetros opcionais, interoperabilidade COM mais simples , digitação dinâmica e funções assíncronas. Eles nos levarão do C# 1 até a versão mais recente, o C# 5. Obviamente, isso é muito para ser feito, então vamos começar.

## 1.1 Começando com um tipo de dados simples

Neste capítulo, deixarei o compilador C# fazer coisas incríveis sem lhe dizer como e apenas mencionar o quê ou o porquê. Este é o único momento em que não vou explicar como as coisas funcionam ou tentar dar um passo de cada vez. Muito pelo contrário, na verdade – o plano é impressionar e não educar. Se você leu esta seção inteira sem ficar pelo menos um pouco entusiasmado com o que o C# pode fazer, talvez este livro não seja para você. Com alguma sorte, porém, você estará ansioso para conhecer os detalhes de como esses truques de mágica funcionam, e é para isso que serve o resto do livro.

O exemplo que usarei é artificial: foi projetado para agrupar o maior número possível de novos recursos em um trecho de código tão curto quanto possível. Também é clichê, mas pelo menos isso o torna familiar. Sim, é um exemplo de produto/nome/preço, a alternativa de comércio eletrônico para “olá, mundo”. Veremos como diversas tarefas podem ser realizadas e como, à medida que avançamos nas versões do C#, você pode realizá-las de maneira mais simples e elegante do que antes. Você não verá nenhum dos benefícios do C# 5 até o final, mas não se preocupe — isso não o torna menos importante.

### 1.1.1 O tipo de produto em C# 1

Começaremos com um tipo que representa um produto e depois o manipularemos. Você não verá nada particularmente impressionante ainda — apenas o encapsulamento de algumas propriedades. Para tornar a vida mais simples para fins de demonstração, é aqui também que criaremos uma lista de produtos predefinidos.

A Listagem 1.1 mostra o tipo como ele pode ser escrito em C# 1. Em seguida, veremos como o código pode ser reescrito para cada versão posterior. Este é o padrão que seguiremos para cada uma das outras partes do código. Dado que estou escrevendo isso em 2013, é provável que você já esteja familiarizado com o código que usa alguns dos recursos que apresentarei, mas vale a pena olhar para trás para ver até onde a linguagem chegou.

#### Listagem 1.1 O tipo de produto (C# 1)

```
using System.Collections; classe pública
Produto
{
    nome da sequência;
    string pública Nome { get { return nome; } }

    preço decimal; preço
    decimal público { obter { preço de retorno; } }

    produto público(nome da string, preço decimal) {

        este.nome = nome;
        este.preço = preço;
    }

    public static ArrayList GetSampleProducts() {

        ListaArrayList = new ArrayList(); list.Add(novo
        Produto("West Side Story", 9,99m)); list.Add(novo Produto("Assassinos",
        14,99m)); list.Add(novo Produto("Rás", 13,99m)); list.Add(novo
        Produto("Sweeney Todd", 10,99m)); lista de retorno;

    }

    string de substituição pública ToString() {

        return string.Format("{0}: {1}", nome, preço);
    }
}
```

Nada na listagem 1.1 deve ser difícil de entender — afinal, é apenas código C# 1.

No entanto, existem três limitações que ele demonstra:

ÿ Um ArrayList não possui informações em tempo de compilação sobre o que está nele. Você poderia adicionar acidentalmente uma string à lista criada em GetSampleProducts e o compilador não piscaria.

ÿ Você forneceu propriedades getter públicas, o que significa que se você quiser setters correspondentes, eles também teriam que ser públicos.

💡 Há muita confusão envolvida na criação de propriedades e variáveis – código que complica a simples tarefa de encapsular uma string e um decimal.

Vamos ver o que o C# 2 pode fazer para melhorar a situação.

### 1.1.2 Coleções fortemente tipadas em C# 2

Nosso primeiro conjunto de alterações (mostradas na listagem a seguir) aborda os dois primeiros itens listados anteriormente, incluindo a alteração mais importante no C# 2: genéricos. As partes que são novas estão em negrito.

#### Listagem 1.2 Coleções fortemente tipadas e setters privados (C# 2)

```
classe pública Produto {

    nome da sequência;
    string pública Nome {

        obter { nome de retorno; }
        conjunto privado { nome = valor; }
    }

    preço decimal; preço
    decimal público {

        obter {preço de retorno; } conjunto
        privado { preço = valor; }
    }

    produto público(nome da string, preço decimal) {

        Nome = nome;
        Preço = preço;
    }

    public static List<Produto> GetSampleProducts() {

        Lista<Produto> lista = new Lista<Produto>(); lista.Add(novo
        Produto("West Side Story", 9,99m)); lista.Add(novo Produto("Assassinos", 14,99m));
        lista.Add(novo Produto("Rás", 13,99m)); lista.Add(novo Produto("Sweeney
        Todd", 10,99m)); lista de retorno;

    }

    string de substituição pública ToString() {

        return string.Format("{0}: {1}", nome, preço);
    }
}
```

Agora você tem propriedades com setters privados (que você usa no construtor) e não é preciso ser um gênio para adivinhar que `List<Product>` está informando ao compilador que a lista contém produtos. Tentar adicionar um tipo diferente à lista resultaria em um erro do compilador e você também não precisa converter os resultados ao buscá-los na lista.

As mudanças no C# 2 deixam apenas uma das três dificuldades originais sem resposta, e o C# 3 ajuda nisso.

### 1.1.3 Propriedades implementadas automaticamente em C# 3

Estamos começando com alguns recursos bastante simples do C# 3. As propriedades implementadas automaticamente e a inicialização simplificada mostradas na listagem a seguir são relativamente triviais em comparação com expressões lambda e similares, mas podem tornar o código muito mais simples.

#### Listagem 1.3 Propriedades implementadas automaticamente e inicialização mais simples (C# 3)

```
using System.Collections.Generic;

class Produto
{
    string pública Nome {obter; conjunto privado; } preço decimal público
    {obter; conjunto privado; }

    produto público(nome da string, preço decimal) {

        Nome = nome;
        Preço = preço;
    }

    Produtos() {}

    public static List<Produto> GetSampleProducts() {

        retornar nova lista<Produto>
        {
            novo produto { Name="West Side Story", Preço = 9,99 milhões }, novo produto
            { Name="Assassinos", Preço=14,99 milhões }, novo produto { Name="Frogs",
            Preço=13,99 milhões }, novo produto { Nome = "Sweeney Todd", Preço =
            10,99 milhões}
        };
    }

    string de substituição pública ToString() {

        return string.Format("{0}: {1}", Nome, Preço);
    }
}
```

Agora as propriedades não possuem nenhum código (ou variáveis visíveis!) associado a elas, e você está construindo a lista codificada de uma maneira muito diferente. Sem variáveis de nome e preço para acessar, você é forçado a usar as propriedades em todos os lugares da classe, melhorando a consistência. Agora você tem um construtor privado sem parâmetros para a nova inicialização baseada em propriedades. (Esse construtor é chamado para cada item antes que as propriedades sejam definidas.)

Neste exemplo, você poderia ter removido completamente o construtor público, mas nenhum código externo poderia ter criado outras instâncias de produto.

## 1.1.4 Argumentos nomeados em C# 4

Para C# 4, voltaremos ao código original no que diz respeito às propriedades e ao construtor, para que ele seja totalmente imutável novamente. Um tipo com apenas setters privados não pode sofrer mutação pública, mas pode ser mais claro se também não for mutável de forma privada.<sup>1</sup> Infelizmente, não há atalho para propriedades somente leitura, mas o C# 4 permite especificar nomes de argumentos para a chamada do construtor., conforme mostrado na listagem a seguir, que oferece a clareza dos inicializadores C# 3 sem a mutabilidade.

## Listagem 1.4 Argumentos nomeados para código de inicialização clara (C# 4)

```
usando System.Collections.Generic; classe pública
Produto {

    nome da string somente leitura;
    string pública Nome { get { return nome; } }

    preço decimal somente leitura; preço
    decimal público { obter { preço de retorno; } }

    produto público(nome da string, preço decimal) {

        este.nome = nome;
        este.preço = preço;
    }

    public static List<Produto> GetSampleProducts() {

        retornar nova lista<Produto>
        {
            novo produto (nome: "West Side Story", preço: 9,99 milhões), novo produto (nome:
            "Assassinos", preço: 14,99 milhões), novo produto (nome: "Frogs", preço: 13,99
            milhões), novo produto ( nome: "Sweeney Todd", preço: 10,99 milhões)

        };
    }

    string de substituição pública ToString() {

        return string.Format("{0}: {1}", nome, preço);
    }
}
```

Os benefícios de especificar explicitamente os nomes dos argumentos são relativamente mínimos neste exemplo específico, mas quando um método ou construtor tem vários parâmetros, isso pode tornar o significado do código muito mais claro - principalmente se eles forem do mesmo tipo ou se você está passando null para alguns argumentos. Você pode escolher quando usar esse recurso, é claro, especificando apenas os nomes dos argumentos quando isso tornar o código mais fácil de entender.

A Figura 1.1 resume como o tipo Produto evoluiu até agora. Incluirei um diagrama semelhante após cada tarefa, para que você possa ver o padrão de como a evolução do C#

<sup>1</sup>

O código C# 1 também poderia ter sido imutável – eu apenas o deixei mutável para simplificar as alterações para C# 2 e 3.

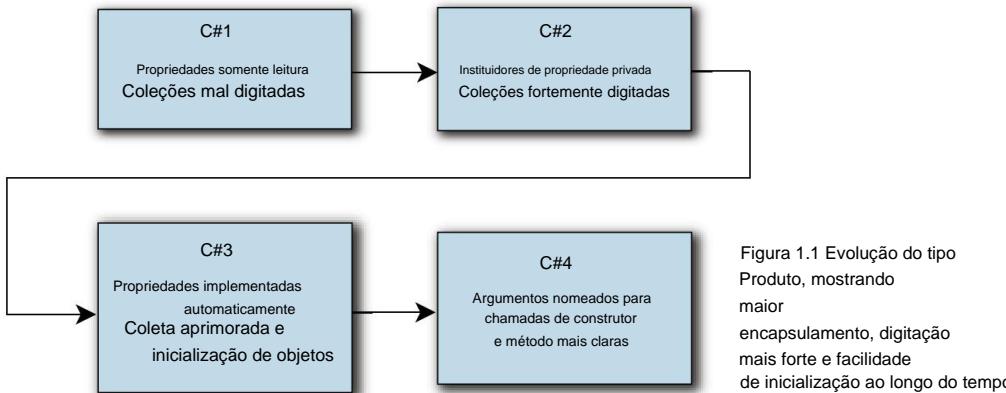


Figura 1.1 Evolução do tipo Produto, mostrando maior encapsulamento, digitação mais forte e facilidade de inicialização ao longo do tempo

melhora o código. Você notará que o C# 5 está faltando em todos os diagramas de blocos; isso porque a principal característica do C# 5 (funções assíncronas) é voltada para uma área que realmente não evoluiu muito em termos de suporte à linguagem. No entanto, daremos uma olhada nisso em breve.

Até agora, as mudanças são relativamente mínimas. Na verdade, a adição de genéricos (a sintaxe `List<Product>`) é provavelmente a parte mais importante do C# 2, mas você só viu parte de sua utilidade até agora. Ainda não há nada para acelerar o coração, mas apenas começamos. Nossa próxima tarefa é imprimir a lista de produtos em ordem alfabética.

## 1.2 Classificação e filtragem

Nesta seção, não

alteraremos o tipo de Produto; em vez disso, pegaremos os produtos de amostra e os classificaremos por nome, e então encontraremos os mais caros. Nenhuma dessas tarefas é exatamente difícil, mas você verá como elas se tornam mais simples com o tempo.

### 1.2.1 Classificando produtos por nome

A maneira mais fácil de exibir uma lista em uma ordem específica é classificá-la e, em seguida, percorrê-la, exibindo os itens. No .NET 1.1, isso envolvia o uso de `ArrayList.Sort` e, opcionalmente, o fornecimento de uma implementação `IComparer` para especificar uma comparação específica. Você poderia fazer com que o tipo `Produto` implementasse `IComparable`, mas isso permitiria definir apenas uma ordem de classificação, e não é exagero imaginar que você possa querer classificar por preço em algum momento, bem como por nome.

A listagem a seguir implementa `IComparer` e, em seguida, classifica a lista e a exibe.

#### Listagem 1.5 Classificando um `ArrayList` usando `IComparer` (C# 1)

```

class NomeDoProdutoComparer: IComparer {
    public int Comparar(objeto x, objeto y) {
        Produto primeiro = (Produto)x; Produto
        segundo = (Produto)y; retorne
        primeiro.Nome.CompareTo(segundo.Nome);
    }
}

```

```

        }
    }
    ...
ArrayList produtos = Product.GetSampleProducts(); produtos.Sort(new
ProductComparer()); foreach (produto produto em produtos) {

    Console.WriteLine (produto);
}

```

A primeira coisa a notar na listagem 1.5 é que você teve que introduzir um tipo extra para ajudar na classificação. Isso não é um desastre, mas é muito código se você quiser classificar apenas por nome em um só lugar. A seguir, observe as conversões no método Compare . Casts são uma forma de dizer ao compilador que você sabe mais informações do que ele, e isso geralmente significa que há uma chance de você estar errado. Se o ArrayList que você retornou de GetSampleProducts contivesse uma string, é aí que o código iria explodir - onde a comparação tenta converter a string em um Produto.

Você também tem uma conversão no código que exibe a lista classificada. Não é óbvio, porque o compilador o insere automaticamente, mas o loop foreach converte implicitamente cada elemento da lista em Produto. Novamente, essa conversão pode falhar em tempo de execução e, mais uma vez, os genéricos vêm em socorro no C# 2. A listagem a seguir mostra o código anterior com o uso de genéricos como a única alteração .

#### Listagem 1.6 Classificando um List<Product> usando IComparer<Product> (C# 2)

```

class ProductNameComparer: IComparer<Produto> {

    public int Compare(Produto x, Produto y) {

        return x.Nome.CompareTo(y.Nome);
    }
}

List<Produto> produtos = Product.GetSampleProducts(); produtos.Sort(new
ProductNameComparer()); foreach (produto produto em produtos) {

    Console.WriteLine(produto);
}

```

O código para o comparador na listagem 1.6 é mais simples porque você recebe produtos para começar. Nenhuma função é necessária. Da mesma forma, a conversão invisível no loop foreach efetivamente desapareceu agora. O compilador ainda deve considerar a conversão do tipo fonte da sequência para o tipo alvo da variável, mas sabe que neste caso ambos os tipos são Produto , portanto não precisa emitir nenhum código para a conversão.

Isso é uma melhoria, mas seria bom se você pudesse classificar os produtos simplesmente especificando a comparação a ser feita, sem a necessidade de implementar uma interface para fazer isso. A listagem a seguir mostra como fazer exatamente isso, informando ao método Sort como comparar dois produtos usando um delegado.

**Listagem 1.7 Classificando um List<Product> usando Comparison<Product> (C# 2)**

```
List<Produto> produtos = Product.GetSampleProducts();  
  
produtos.Sort(delegate(Produto x, Produto y)  
    { return x.Nome.CompareTo(y.Nome); }  
);  
foreach (produto produto em produtos) {  
  
    Console.WriteLine(produto);  
}
```

Veja a falta do tipo `ProductNameComparer`. A instrução em negrito cria uma instância delegada, que você fornece ao método `Sort` para realizar as comparações. Você aprenderá mais sobre esse recurso (*métodos anônimos*) no capítulo 5.

Agora você corrigiu todos os problemas identificados na versão C# 1. Isso não significa que o C# 3 não possa fazer melhor. Primeiro, você substituirá o método anônimo por uma maneira ainda mais compacta de criar uma instância delegada, conforme mostrado na listagem a seguir.

**Listagem 1.8 Classificando usando Comparison<Product> de uma expressão lambda (C# 3)**

```
List<Produto> produtos = Product.GetSampleProducts(); produtos.Sort((x, y) =>  
x.Nome.CompareTo(y.Nome)); foreach (produto produto em produtos) {  
  
    Console.WriteLine(produto);  
}
```

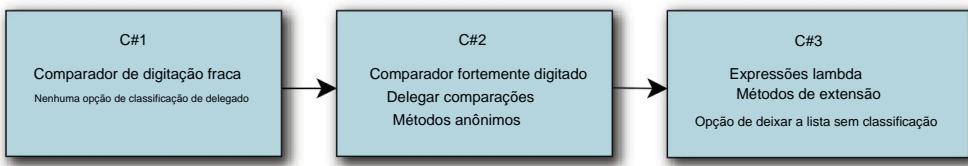
Você ganhou uma sintaxe ainda mais estranha (*uma expressão lambda*), que ainda cria um delegado `Comparison<Product>`, assim como a listagem 1.7 fez, mas desta vez com menos complicações. Você não precisou usar a palavra-chave delegada para apresentá-la, nem mesmo especificar os tipos de parâmetros.

Porém, há mais: com o C# 3, você pode imprimir facilmente os nomes em ordem, sem modificar a lista original de produtos. A próxima listagem mostra isso usando o método `OrderBy`.

**Listagem 1.9 Solicitando um List<Product> usando um método de extensão (C# 3)**

```
List<Produto> produtos = Product.GetSampleProducts(); foreach (Produto produto em  
produtos.OrderBy(p => p.Name) ) {  
  
    Console.WriteLine (produto);  
}
```

Nesta listagem, parece que você está chamando um método `OrderBy` na lista, mas se você procura no MSDN, verá que ele nem existe em `List<Product>`. Você pode chamá-lo devido à presença de um *método de extensão*, que você verá com mais detalhes no capítulo 10. Na verdade, você não está mais classificando a lista “no lugar”, apenas recuperando o conteúdo



**Figura 1.2 Recursos envolvidos na facilitação da classificação em C# 2 e 3**

da lista em uma ordem específica. Às vezes você precisará alterar a lista real; às vezes é melhor fazer um pedido sem quaisquer outros efeitos colaterais.

O importante é que esse código é muito mais compacto e legível (depois que você entender a sintaxe, é claro). Queríamos a lista ordenada por nome e é exatamente isso que diz o código. Ele não diz para classificar comparando o nome de um produto com o nome de outro, como fazia o código C# 2, ou para classificar usando uma instância de outro tipo que sabe como comparar um produto com outro. Diz apenas para pedir pelo nome. Essa simplicidade de expressão é um dos principais benefícios do C# 3. Quando as partes individuais de consulta e manipulação de dados são tão simples, transformações maiores podem permanecer compactas e legíveis em uma parte do código. Isso, por sua vez, incentiva uma forma de ver o mundo mais centrada nos dados.

Você viu mais do poder do C# 2 e 3 nesta seção, com muita sintaxe (ainda) inexplicável, mas mesmo sem entender os detalhes você pode ver o progresso em direção a um código mais claro e simples. A Figura 1.2 mostra essa evolução.

Isso é tudo para classificação.<sup>2</sup> Vamos fazer uma forma diferente de manipulação de dados agora: consulta.

## 1.2.2 Consultando coleções

Sua próxima tarefa é encontrar todos os elementos da lista que correspondam a um determinado critério – em particularmente, aqueles com um preço superior a US\$ 10. A listagem a seguir mostra como, em C# 1, você precisa fazer um loop, testando cada elemento e imprimindo-o quando apropriado.

### Listagem 1.10 Loop, teste, impressão (C# 1)

```

ArrayList produtos = Product.GetSampleProducts(); foreach (produto produto em
produtos) {

    if (produto.Preço > 10m) {

        Console.WriteLine(produto);
    }
}

```

Este código *não* é difícil de entender. Mas vale a pena ter em mente o quanto interligadas estão as três tarefas – fazer loop com foreach, testar o critério com if e

---

<sup>2</sup> O C# 4 fornece um recurso que pode ser relevante durante a classificação, chamado *variação genérica*, mas dar um exemplo aqui exigiria muita explicação. Você pode encontrar os detalhes perto do final do capítulo 13.

em seguida, exibindo o produto com `Console.WriteLine`. A dependência é óbvia por causa do aninhamento.

A listagem a seguir demonstra como o C# 2 permite simplificar um pouco as coisas.

#### Listagem 1.11 Separando teste de impressão (C# 2)

```
List<Produto> produtos = Product.GetSampleProducts();

Predicado<Produto> teste = delegado(Produto p) { return p.Preço > 10m; }; List<Produto> corresponde =
produtos.FindAll(teste);

Action<Produto> print = Console.WriteLine;
corresponde.ForEach(imprimir);
```

A variável de teste é inicializada usando o recurso de método anônimo que você viu na seção anterior. A inicialização da variável de impressão usa outro novo recurso do C# 2 chamado *conversões de grupo de métodos*, que facilita a criação de delegados a partir de métodos existentes.

Não vou afirmar que esse código é mais simples que o código C# 1, mas é *muito* mais poderoso.<sup>3</sup> Em

particular, a técnica de separar as duas preocupações como essa torna *muito* fácil alterar a condição que você está testando e a ação que você executa em cada uma das partidas de forma independente. As variáveis delegadas envolvidas (teste e impressão) poderiam ser passadas para um método, e esse mesmo método poderia acabar testando condições radicalmente diferentes e executando ações radicalmente diferentes. Claro, você poderia colocar todos os testes e impressões em uma única instrução, conforme mostrado na listagem a seguir.

#### Listagem 1.12 Separando teste de impressão redux (C# 2)

```
List<Produto> produtos = Product.GetSampleProducts();
produtos.FindAll(delegate(Produto p) { return p.Price > 10; })
    .ForEach(Console.WriteLine);
```

De certa forma, esta versão é melhor, mas o delegado (`Produto p`) está atrapalhando, assim como os colchetes. Eles estão adicionando ruído ao código, o que prejudica a legibilidade. Ainda prefiro a versão C# 1 nos casos em que desejo apenas usar o mesmo teste e realizar a mesma ação. (Pode parecer óbvio, mas vale lembrar que nada impede você de usar o código C# 1 com uma versão posterior do compilador. Você não usaria uma escavadeira para plantar bulbos de tulipas, que é o tipo de exagero usado na última listagem.)

A próxima listagem mostra como o C# 3 melhora drasticamente a situação, removendo muitos do boato que cerca a *lógica real* do delegado.

---

<sup>3</sup> De certa forma, isso é trapaça. Você poderia ter definido delegados apropriados em C# 1 e chamado-os dentro do loop. Os métodos `FindAll` e `ForEach` no .NET 2.0 apenas incentivam você a considerar a separação de interesses.

**Listagem 1.13 Testando com uma expressão lambda (C# 3)**

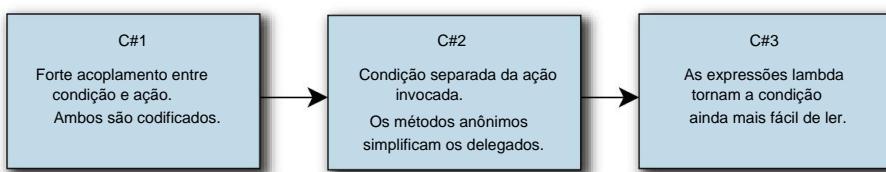
```
List<Produto> produtos = Product.GetSampleProducts(); foreach (Produto produto em
produtos.Where(p => p.Price > 10)) {
    Console.WriteLine(produto);
}
```

A combinação da expressão lambda colocando o teste no lugar certo e um método bem nomeado significa que você *quase* pode ler o código em voz alta e entendê-lo sem pensar. Você ainda tem a flexibilidade do C# 2 – o argumento para Where pode vir de uma variável e você pode usar um Action<Product> em vez da chamada Console.WriteLine embutida em código , se desejar.

Esta tarefa enfatizou o que você já sabia sobre classificação: métodos anônimos simplificam a escrita de um delegado e as expressões lambda são ainda mais concisas.

Em ambos os casos, essa brevidade significa que você pode incluir a operação de consulta ou classificação dentro da primeira parte do loop foreach sem perder a clareza.

A Figura 1.3 resume as mudanças que acabamos de observar. C# 4 não oferece nada coisa para simplificar ainda mais esta tarefa.



**Figura 1.3** Métodos anônimos e expressões lambda em C# 2 e 3 auxiliam na separação de interesses e na legibilidade.

Agora que você exibiu a lista filtrada, vamos considerar uma mudança nas suas suposições iniciais sobre os dados. O que acontece se você nem sempre sabe o preço de um produto? Como você pode lidar com isso na classe Produto ?

## 1.3 Lidando com a ausência de dados

Veremos duas formas diferentes de dados ausentes. Primeiro, lidaremos com o cenário em que você realmente não possui as informações e, em seguida, veremos como *remover* ativamente as informações das chamadas de método, usando valores padrão.

### 1.3.1 Representando um preço desconhecido

Não apresentarei muito código desta vez, mas tenho certeza de que será um problema familiar para você, especialmente se você já trabalhou muito com bancos de dados. Imagine que sua lista de produtos contém não apenas produtos à venda no momento, mas também produtos que ainda não estão disponíveis. Em alguns casos, você pode não saber o preço. Se decimal fosse um tipo de referência, você poderia usar null para representar o preço desconhecido, mas como é um tipo de valor, não é possível. Como você representaria isso em C# 1?

Existem três alternativas comuns:

- ↳ Crie um wrapper de tipo de referência em torno de decimal.
- ↳ Mantenha um sinalizador booleano separado indicando se o preço é conhecido.
- ↳ Use um “valor mágico” (decimal.MinValue, por exemplo) para representar o preço desconhecido.

Espero que você concorde que nada disso tem muito apelo. É hora de um pouco de mágica: você pode resolver o problema adicionando um único caractere nas declarações de variáveis e propriedades. O .NET 2.0 torna as coisas muito mais simples ao introduzir a estrutura Nullable<T>, e o C# 2 fornece algum açúcar sintático adicional que permite alterar a declaração de propriedade para este bloco de código:

```
decimal? preço; decimal
público? Preço {
    obter {preço de retorno; } conjunto
    privado { preço = valor; }
}
```

O parâmetro do construtor muda para decimal? e então você pode passar null como argumento ou dizer Price = null; dentro da classe. O significado do nulo muda de “uma referência especial que não se refere a nenhum objeto” para “um valor especial de qualquer tipo anulável representando a ausência de outros dados”, onde todos os tipos de referência e todos os tipos baseados em Nullable <T> contam como *tipos anuláveis*.

Isso é muito mais expressivo do que qualquer uma das outras soluções. O restante do código funciona como está: um produto com preço desconhecido será considerado menos caro que US\$ 10, devido à forma como os valores anuláveis são tratados em comparações maiores que. Para verificar se um preço é conhecido, você pode compará-lo com null ou usar a propriedade HasValue , então para mostrar todos os produtos com preços desconhecidos em C# 3, você escreveria o seguinte código.

#### Listagem 1.14 Exibindo produtos com preço desconhecido (C# 3)

```
List<Produto> produtos = Product.GetSampleProducts(); foreach (Produto produto em
produtos.Where(p => p.Price == null)) {
    Console.WriteLine(produto.Nome);
}
```

O código C# 2 seria semelhante ao da listagem 1.12, mas você precisaria verificar se há null no método anônimo:

```
List<Produto> produtos = Product.GetSampleProducts(); produtos.FindAll(delegate(Produto
p) { return p.Price == null; })
.ForEach(Console.WriteLine);
```

O C# 3 não oferece nenhuma alteração aqui, mas o C# 4 tem um recurso que está pelo menos tangencialmente relacionado.

### 1.3.2 Parâmetros opcionais e valores padrão

Às vezes você não quer dizer a um método tudo o que ele precisa saber, como quando você quase sempre usa o mesmo valor para um parâmetro específico. Tradicionalmente, a solução tem sido sobrecarregar o método em questão, mas o C# 4 introduziu *parâmetros opcionais* para tornar isso mais simples.

Na versão C# 4 do tipo `Produto`, você tem um construtor que leva o nome e o preço. Você pode tornar o preço um decimal anulável, assim como em C# 2 e 3, mas vamos supor que a *maioria* dos produtos não tenha preços. Seria bom poder inicializar um produto como este:

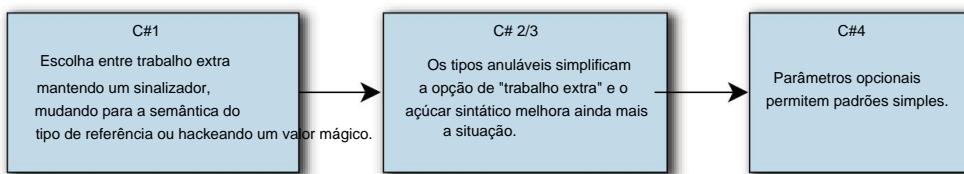
```
Produto p = novo Produto("Produto não lançado");
```

Antes do C# 4, você teria que introduzir uma nova sobrecarga no construtor `Product` para essa finalidade. C# 4 permite declarar um valor padrão (neste caso, nulo) para o parâmetro `price`:

```
produto público (nome da string, decimal? preço = nulo) {  
    este.nome = nome;  
    este.preço = preço;  
}
```

Você sempre deve especificar um valor constante ao declarar um parâmetro opcional. Não precisa ser nulo; esse é o padrão apropriado nesta situação. O requisito de que o valor padrão seja uma constante se aplica a qualquer tipo de parâmetro, embora para tipos de referência diferentes de strings você *está* limitado a null como o único valor constante disponível.

A Figura 1.4 resume a evolução que observamos nas diferentes versões do C#.



**Figura 1.4 Opções para trabalhar com dados faltantes**

Até agora, os recursos têm sido úteis, mas talvez nada digno de nota. A seguir veremos algo um pouco mais interessante: LINQ.

### 1.4 Apresentando o LINQ O LINQ

(Consulta Integrada à Linguagem) está no centro das mudanças no C# 3. Como o próprio nome sugere, o LINQ tem tudo a ver com consultas — o objetivo é facilitar a gravação de consultas em diversas fontes de dados com sintaxe consistente e recursos, de forma legível e combinável.

Embora os recursos do C# 2 sejam indiscutivelmente mais voltados para corrigir aborrecimentos no C# 1 do que colocar fogo no mundo, quase tudo no C# 3 é baseado no LINQ, e o resultado é bastante especial. Já vi recursos em outras linguagens que abordam *algumas* das mesmas áreas do LINQ, mas nada tão completo e flexível.

### 1.4.1 Expressões de consulta e consultas em processo

Se você já viu algum código LINQ antes, provavelmente já viu *expressões de consulta* que permitem usar um estilo declarativo para criar consultas em diversas fontes de dados. A razão pela qual nenhum dos exemplos deste capítulo usou expressões de consulta até agora é que todos os exemplos foram mais simples *sem* usar a sintaxe extra. Isso não quer dizer que você não possa usá-lo de qualquer maneira — a listagem a seguir, por exemplo, é equivalente à listagem 1.13.

#### Listagem 1.15 Primeiros passos com expressões de consulta: filtrando uma coleção

```
List<Produto> produtos = Product.GetSampleProducts(); var filtrado = do produto p em
produtos
    onde p.Price > 10 selecione p;

foreach (Produto produto filtrado) {

    Console.WriteLine(produto);
}
```

Pessoalmente, acho a listagem anterior mais fácil de ler – o único benefício desta versão da expressão de consulta é que a cláusula `where` é mais simples. Eu incluí um recurso extra *aqui: variáveis locais digitadas implicitamente*, que são declaradas usando a palavra-chave contextual `var`. Isso permite que o compilador infira o tipo de uma variável a partir do valor inicialmente atribuído a ela — nesse caso, o tipo de filtrado é `IEnumerable<Product>`. Usarei `var` bastante extensivamente no restante dos exemplos deste capítulo; é particularmente útil em livros, onde o espaço nas listagens é escasso.

Mas se as expressões de consulta não são boas, por que todo mundo faz tanto barulho com elas e com o LINQ em geral? A primeira resposta é que, embora as expressões de consulta não sejam particularmente benéficas para tarefas simples, elas são *muito* boas para situações mais complicadas que seriam difíceis de ler se escritas nas chamadas de método equivalentes (e seriam diabólicas em C# 1 ou 2). Vamos tornar as coisas um pouco mais difíceis introduzindo outro tipo: Fornecedor.

Cada fornecedor possui um `Nome` (`string`) e um `SupplierID` (`int`). Também adicionei `SupplierID` como uma propriedade em `Product` e adaptei os dados de amostra adequadamente. É certo que essa não é uma maneira muito orientada a objetos de atribuir um fornecedor a cada produto — é muito mais próxima de como os dados seriam representados em um banco de dados. Isso torna esse recurso específico mais fácil de demonstrar por enquanto, mas você verá no capítulo 12 que o LINQ também permite usar um modelo mais natural.

Agora vamos dar uma olhada no código (listagem 1.16) que une os produtos de amostra com os fornecedores de amostra (obviamente com base no ID do fornecedor), aplica o mesmo filtro de preço de antes aos produtos, classifica por nome do fornecedor e, em seguida, nome do produto, e imprime fora

o nome do fornecedor e do produto para cada correspondência. Isso foi complicado e, nas versões anteriores do C#, teria sido um pesadelo implementar. No LINQ, é quase trivial.

#### Listagem 1.16 Unindo, filtrando, ordenando e projetando (C# 3)

```
List<Produto> produtos = Product.GetSampleProducts(); Lista<Fornecedor> fornecedores
= Fornecedor.GetSampleSuppliers(); var filtrado = de p em produtos junta-se a s em fornecedores

        em p.SupplierID é igual a s.SupplierID
        onde p.Price > 10 orderby
        s.Name, p.Name select new
        { SupplierName = s.Name, ProductName = p.Name };
foreach (var v em filtrado) {

    Console.WriteLine("Fornecedor={0}; Produto={1}",
                      v.NomeFornecedor, v.NomeProduto);
}
```

Você deve ter notado que isso se parece muito com SQL. Na verdade, a reação de muitas pessoas ao ouvirem falar do LINQ pela primeira vez (mas antes de examiná-lo de perto) é rejeitá-lo como uma mera tentativa de colocar SQL na linguagem para conversar com bancos de dados. Felizmente, o LINQ pegou emprestada a sintaxe e algumas ideias do SQL, mas como você viu, você não precisa estar perto de um banco de dados para usá-lo. Nenhum código que você viu até agora tocou um banco de dados. Na verdade, você poderia obter dados de inúmeras fontes: XML, por exemplo.

### 1.4.2 Consultando XML

Suponha que, em vez de codificar seus fornecedores e produtos, você tenha usado o seguinte arquivo XML :

```
<?xml versão="1.0"?>
<Dados>
    <Produtos>
        <Nome do produto="West Side Story" Preço="9,99" SupplierID="1" /> <Nome do produto="Assassins"
        Price="14,99" SupplierID="2" /> <Nome do produto="Rás" Price="13,99" SupplierID="1" />
        <Product Name="Sweeney Todd" Price="10,99" SupplierID="3" />

    </Produtos>
    <Fornecedores>
        <Nome do fornecedor="Solely Sondheim" SupplierID="1" />
        <Nome do fornecedor="CD-por-CD-por-Sondheim" SupplierID="2" />
        <Nome do fornecedor="CDs de barbearia" SupplierID="3" />
    </Fornecedores>
</Dados>
```

O arquivo é bastante simples, mas qual é a melhor maneira de extrair os dados dele? Como você consulta isso? Participe? Certamente será um pouco mais difícil do que o que você fez na listagem 1.16, certo? A listagem a seguir mostra quanto trabalho você precisa fazer no LINQ to XML.

### Listagem 1.17 Processamento complexo de um arquivo XML com LINQ to XML (C# 3)

Essa abordagem não é tão simples, porque você precisa informar ao sistema como ele deve entender os dados (em termos de quais atributos devem ser usados e quais tipos), mas não está muito longe. Em particular, existe uma relação óbvia entre cada parte das duas listagens. Se não fosse pelas limitações de comprimento de linha dos livros, você veria uma correspondência exata linha por linha entre as duas consultas.

Impressionado ainda? Não está muito convencido? Vamos colocar os dados onde é muito mais provável estar - em um banco de dados.

### 1.4.3 LINQ para SQL

Há algum trabalho envolvido em deixar o LINQ to SQL saber o que esperar em qual tabela, mas é tudo bastante simples e muito disso pode ser automatizado. Iremos direto para o código de consulta, que é mostrado na listagem a seguir. Se você quiser ver os detalhes do `LinqDemoDataContext`, eles estão todos no código-fonte para download.

Listagem 1.18 Aplicando uma expressão de consulta a um banco de dados SQL (C# 3)

```
usando (LinqDemoDataContext db = new LinqDemoDataContext()) { var filtrado = de p em

db.Products
    junte-se a s em db.Fornecedores
        em p.SupplierID é igual a s.SupplierID
        onde p.Price > 10 orderby
            s.Name, p.Name select new
            { SupplierName = s.Name, ProductName = p.Name };
foreach (var v em filtrado) {
    Console.WriteLine("Fornecedor={0}; Produto={1}",
        v.NomeFornecedor, v.NomeProduto);
}
}
```

A esta altura, isso deve parecer incrivelmente familiar. Tudo abaixo da linha de junção é recortado e colado diretamente da listagem 1.16, sem alterações.

Isso é bastante impressionante, mas se você estiver preocupado com o desempenho, poderá estar se perguntando por que deseja extrair todos os dados do banco de dados e depois aplicar essas consultas e ordenações do .NET. Por que não fazer com que o banco de dados faça isso? É nisso que é bom, não é? Bem, de fato — e é exatamente isso que o LINQ to SQL faz. O código da listagem 1.18 emite uma solicitação ao banco de dados, que é basicamente a consulta traduzida em SQL.

Mesmo que você tenha *expressado* a consulta em código C#, ela foi *executada* como SQL.

Você verá mais tarde que existe uma maneira mais orientada para o relacionamento de abordar esse tipo de junção quando o esquema e as entidades conhecem o relacionamento entre fornecedores e produtos. O resultado é o mesmo, porém, e mostra quão semelhantes podem ser o LINQ to Objects (o LINQ na memória operando em coleções) e o LINQ to SQL.

LINQ é extremamente flexível — você pode escrever seu próprio provedor para se comunicar com um serviço web ou traduzir uma consulta em sua representação específica. No capítulo 13, veremos o quanto amplo o termo *LINQ* realmente é e como ele pode ir além do que você poderia esperar em termos de consulta a coleções.

## 1.5 COM e digitação dinâmica

A seguir,

gostaria de demonstrar alguns recursos específicos do C# 4. Enquanto o LINQ era o foco principal do C# 3, a interoperabilidade era o tema principal do C# 4. Isso inclui trabalhar com a tecnologia antiga do COM e também o admirável mundo novo de linguagens dinâmicas executadas no *Dynamic Language Runtime* (DLR). Começaremos exportando a lista de produtos para uma planilha Excel.

### 1.5.1 Simplificando a interoperabilidade COM

Existem várias maneiras de disponibilizar dados para o Excel, mas usar COM para controlá-los oferece mais poder e flexibilidade. Infelizmente, as encarnações anteriores do C# dificultaram bastante o trabalho com COM; VB teve um suporte muito melhor. O C# 4 corrige amplamente essa situação.

A listagem a seguir mostra algum código para salvar seus dados em uma nova planilha.

#### Listagem 1.19 Salvando dados no Excel usando COM (C# 4)

```
var app = new Application { Visível = false }; Pasta de trabalho pasta de
trabalho = app.Workbooks.Add(); Planilha de planilha =
app.ActiveSheet; linha interna = 1; foreach (var produto em
Product.GetSampleProducts()
    .Where(p => p.Price != null))
{
    planilha.Células[linha, 1].Valor = produto.Nome; planilha.Células[linha,
    2].Valor = produto.Preço; linha++;
}

} pasta de trabalho.SaveAs(Nome do arquivo: "demo.xls",
FileFormat: XlFileFormat.xlWorkbookNormal);
app.Application.Quit();
```

Isso pode não ser tão bom quanto você gostaria, mas é muito *melhor* do que seria usando versões anteriores do C#. Na verdade, você já conhece alguns dos recursos do C# 4 mostrados aqui — mas há alguns outros que não são tão óbvios. Aqui está a lista completa:

↳ A chamada SaveAs usa argumentos nomeados. ↳

Várias chamadas omitem argumentos para parâmetros opcionais – em particular, SaveAs normalmente teria 10 argumentos extras!

↳ C# 4 pode incorporar as partes relevantes do *conjunto de interoperabilidade primário* (PIA) no chamando o código, então você não precisa mais implantar o PIA separadamente.

↳ Em C# 3, a atribuição à planilha falharia sem conversão, porque o tipo da propriedade ActiveSheet é representado como objeto. Ao usar o recurso PIA incorporado , o tipo de ActiveSheet torna-se dinâmico, o que leva a um recurso totalmente diferente.

Além disso, o C# 4 dá suporte a indexadores nomeados ao trabalhar com COM — um recurso não demonstrado neste exemplo.

Já mencionei o recurso final: digitação dinâmica em C# usando o tipo dinâmico .

### 1.5.2 Interoperando com uma linguagem dinâmica

A digitação dinâmica é um tópico tão importante que todo o capítulo 14 é dedicado a ele. Vou apenas mostrar um pequeno exemplo do que ele pode fazer aqui.

Suponha que seus produtos não estejam armazenados em um banco de dados, nem em XML, nem na memória. Eles são acessíveis por meio de uma espécie de serviço da web, mas você só tem código Python para acessá-los, e esse código usa a natureza dinâmica do Python para construir resultados sem declarar um tipo contendo todas as propriedades que você precisa acessar em cada resultado. Em vez disso, os resultados permitem solicitar qualquer propriedade e tentar descobrir o que você quer dizer no momento da execução. Em uma linguagem como Python, não há nada de incomum nisso. Mas como você pode acessar seus resultados em C#?

A resposta vem na forma de dinâmico – um novo tipo<sup>4</sup> que o compilador C# permite usar dinamicamente. Se uma expressão for do tipo dinâmico, você pode chamar métodos nela, acessar propriedades, passá-la como um argumento de método e assim por diante — e a maior parte do processo normal de vinculação acontece em tempo de execução, em vez de tempo de compilação. Você pode converter implicitamente um valor de dinâmico para qualquer outro tipo (é por isso que a planilha lançada na listagem 1.19 funcionou) e fazer todo tipo de outras coisas divertidas.

Esse comportamento também pode ser útil mesmo em código C# puro, sem nenhuma interoperabilidade envolvida, mas é divertido vê-lo funcionando com outras linguagens. A lista a seguir mostra como você pode obter a lista de produtos do IronPython e imprimi-la. Isso inclui todo o código de configuração para executar o código Python no mesmo processo.

---

<sup>4</sup> Mais ou menos, de qualquer maneira. É um tipo no que diz respeito ao compilador C#, mas o CLR não sabe nada sobre isso.

**Listagem 1.20 Executando IronPython e extraindo propriedades dinamicamente (C# 4)**

```
Mecanismo ScriptEngine = Python.CreateEngine(); Escopo ScriptScope =
engine.ExecuteFile("FindProducts.py"); produtos dinâmicos = escopo.GetVariable("produtos");
foreach (produto dinâmico em produtos) {

    Console.WriteLine("{0}: {1}", produto.ProductName, produto.Price);
}
```

Tanto os produtos quanto o produto são declarados dinâmicos, portanto o compilador fica feliz em permitir que você itere a lista de produtos e imprima as propriedades, mesmo que não saiba se funcionará. Se você cometer um erro de digitação, usando product.Name em vez de product.ProductName, por exemplo, isso só apareceria em tempo de execução.

Isso é completamente contrário ao restante do C#, que é digitado estaticamente. Mas a digitação dinâmica só entra em ação quando estão envolvidas expressões com um tipo de dinâmica ; a maior parte do código C# provavelmente permanecerá digitado estaticamente.

## 1.6 Escrevendo código assíncrono sem dor de cabeça

Finalmente você verá o grande recurso do C# 5: funções assíncronas, que permitem pausar a execução do código sem bloquear um thread.

Este tópico é grande – muito grande – mas vou dar apenas um trecho por enquanto. Como tenho certeza que você sabe, existem duas regras de ouro quando se trata de threading no Windows Forms: você não deve bloquear o thread da UI e não deve acessar elementos da UI em qualquer outro thread, exceto em algumas maneiras bem especificadas. A listagem a seguir mostra um único método que lida com um clique de botão em um aplicativo Windows Forms e exibe informações sobre um produto, de acordo com seu ID.

**Listagem 1.21 Exibindo produtos em Windows Forms usando uma função assíncrona**

```
private async void CheckProduct(objeto remetente, EventArgs e) {

    tentar
    {
        produtoCheckButton.Enabled = falso; string id = idInput.Text;

        Task<Produto> productLookup = diretório.LookupProductAsync(id); Tarefa<int> stockLookup =
        warehouse.LookupStockLevelAsync(id); Produto produto = aguardar productLookup; if (produto ==
        nulo) {

            retornar;

        } nomeValue.Text = produto.Nome; preçoValue.Text
        = produto.Price.ToString("c");

        int estoque = aguarda stockLookup; stockValue.Text
        = stock.ToString(); }
```

```
finalmente  
  
{produtoCheckButton.Enabled = verdadeiro; }  
  
}
```

O método completo é um pouco mais longo que o mostrado na listagem 1.22, exibindo mensagens de status e limpando os resultados no início, mas esta listagem contém todas as partes importantes. As novas partes da sintaxe estão em negrito – o método possui o novo modificador `async` e há duas expressões de espera.

Se você apertar os olhos e ignorá-los por enquanto, provavelmente poderá entender o fluxo geral do código. Ele começa realizando pesquisas no diretório de produtos e no depósito para descobrir os detalhes do produto e o estoque atual. O método então espera até obter as informações do produto e é encerrado se o diretório não tiver nenhuma entrada para o ID fornecido. Caso contrário, ele preenche os elementos da interface do usuário com o nome e o preço e, em seguida, espera para obter as informações do estoque e as exibe também.

As pesquisas de produtos e de estoque são assíncronas – podem ser operações de banco de dados ou chamadas de serviço web. Não importa: quando você aguarda os resultados, na verdade não está bloqueando o thread da UI, mesmo que todo o código do método seja executado nesse thread. Quando os resultados voltarem, o método continua de onde parou.

O exemplo também demonstra que o controle de fluxo normal (`tentativa/finalmente`) funciona exatamente como você espera. O que é realmente surpreendente sobre esse método é que ele conseguiu atingir exatamente o tipo de assincronia que você deseja, sem qualquer confusão normal ao iniciar outros threads ou `BackgroundWorkers`, chamar `Control.BeginInvoke` ou anexar retornos de chamada a eventos assíncronos. É claro que você ainda precisa pensar: a assincronia não se torna fácil usando `async/await`, mas se torna menos tediosa, com muito menos código clichê para distraí-lo da complexidade inerente que você está tentando controlar.

Você já está tonto? Relaxe, vou desacelerar consideravelmente pelo resto do livro. Em particular, explicarei alguns dos casos extremos, entrando em mais detalhes sobre por que vários recursos foram introduzidos e dando algumas orientações sobre quando é apropriado usá-los.

Até agora, mostrei recursos do C#. Alguns desses recursos requerem assistência de biblioteca e alguns deles requerem assistência de tempo de execução. Direi muito esse tipo de coisa, então vamos esclarecer o que quero dizer.

## 1.7 Dissecando a plataforma .NET

Quando foi originalmente introduzido, o .NET era usado como um termo genérico para uma vasta gama de tecnologias provenientes da Microsoft. Por exemplo, o Windows Live ID era chamado de .NET Passport, apesar de não haver uma relação clara entre ele e o que atualmente conhecemos como .NET. Felizmente, as coisas se acalmaram um pouco desde então. Nesta seção, veremos as diversas partes do .NET.

Em vários lugares deste livro, me referirei a três tipos diferentes de recursos: recursos do C# como linguagem, recursos do tempo de execução que fornece o “mecanismo”, por assim dizer, e recursos das bibliotecas do .NET Framework. Este livro é fortemente focado na linguagem C#, e geralmente discutirei apenas os recursos de tempo de execução e de estrutura quando eles estiverem relacionados aos recursos do próprio C#. Freqüentemente, os recursos se sobrepõem, mas é importante entender onde estão os limites.

### 1.7.1 C#, a linguagem

A linguagem C# é definida por sua especificação, que descreve o formato do código-fonte C#, incluindo sintaxe e comportamento. Ele não descreve a plataforma na qual a saída do compilador será executada, além de alguns pontos-chave onde os dois interagem. Por exemplo, a linguagem C# requer um tipo chamado `System.IDisposable`, que contém um método chamado `Dispose`. Eles são necessários para definir a instrução `using`.

Da mesma forma, a plataforma precisa ser capaz de suportar (de uma forma ou de outra) tipos de valor e tipos de referência, juntamente com a coleta de lixo.

Em teoria, qualquer plataforma que suporte os recursos necessários poderia ter um compilador C# direcionado a ela. Por exemplo, um compilador C# poderia legitimamente produzir saída em um formato diferente da Linguagem Intermediária (IL), que é a saída típica no momento em que este livro foi escrito. Um tempo de execução poderia interpretar a saída de um compilador C# ou converter tudo em código nativo em uma única etapa, em vez de compilá-lo JIT. Embora essas opções sejam relativamente incomuns, elas existem à solta; por exemplo, o Micro Framework usa um interpretador, assim como o Mono (<http://mono-project.net>). No outro extremo do espectro, a compilação antecipada é usada pelo NGen e pelo Xamarin.iOS (<http://xamarin.com/ios>) — uma plataforma para criar aplicativos para iPhone e outros dispositivos iOS.

### 1.7.2 Tempo de execução

O aspecto de tempo de execução da plataforma .NET é a quantidade relativamente pequena de código responsável por garantir que os programas escritos em IL sejam executados de acordo com a especificação Common Language Infrastructure (CLI) (ECMA-335 e ISO/IEC 23271), partições I a III. A parte do tempo de execução da CLI é chamada Common Language Runtime (CLR).

Quando me refiro ao CLR no restante do livro, quero dizer a implementação da Microsoft.

Alguns elementos da linguagem C# nunca aparecem no nível do tempo de execução, mas outros ultrapassam a linha divisória. Por exemplo, os enumeradores não são definidos no nível do tempo de execução e nem há nenhum significado específico associado à interface `IDisposable`, mas matrizes e delegados são importantes para o tempo de execução.

### 1.7.3 Bibliotecas de estrutura

As bibliotecas fornecem código que está disponível para seus programas. As bibliotecas de estrutura no .NET são em grande parte construídas como IL, com código nativo usado apenas quando necessário.

Isso é uma marca da força do tempo de execução: não se espera que seu próprio código seja um cidadão de segunda classe — ele pode fornecer o mesmo tipo de poder e desempenho que o

bibliotecas que utiliza. A quantidade de código nas bibliotecas é muito maior que a do tempo de execução, da mesma forma que um carro envolve muito mais do que o motor.

As bibliotecas da estrutura são parcialmente padronizadas. A Partição IV da especificação CLI fornece vários perfis diferentes (*compacto* e *kernel*) e bibliotecas. A Partição IV vem em duas partes – uma descrição textual geral das bibliotecas identificando, entre outras coisas, quais bibliotecas são necessárias em quais perfis, e outra parte contendo os detalhes das próprias bibliotecas em formato XML. Esta é a mesma forma de documentação produzida quando você usa comentários XML em C#.

Há muitas coisas no .NET que *não* estão nas bibliotecas básicas. Se você escrever um programa que use *apenas* bibliotecas da especificação, e as usar corretamente, descobrirá que seu código funciona perfeitamente em qualquer implementação — Mono, .NET ou qualquer outra. Mas, na prática, quase todos os programas de qualquer tamanho usarão bibliotecas que não são padronizadas — Windows Forms ou ASP.NET, por exemplo. O projeto Mono possui bibliotecas próprias que não fazem parte do .NET, como GTK#, e implementa muitas das bibliotecas não padronizadas.

O termo .NET refere-se à combinação de tempo de execução e bibliotecas fornecidas pela Microsoft e também inclui compiladores para C# e VB.NET. Pode ser visto como uma *plataforma de desenvolvimento* completa construída sobre o Windows. Cada aspecto do .NET tem versões separadas, o que pode ser uma fonte de confusão. O Apêndice C fornece um rápido resumo de qual versão do que foi lançada, quando e com quais recursos.

Se tudo estiver claro, tenho que fazer uma última tarefa doméstica antes de realmente comece a mergulhar em C#.

## **1.8 Tornando seu código super incrível** Peço desculpas

pelo título enganoso. Esta seção (por si só) *não* tornará seu código super incrível. Não vai nem torná-lo refrescantemente mentolado. No entanto, isso o ajudará a aproveitar ao máximo este livro – e é por isso que eu queria ter certeza de que você realmente o lerá.

Há mais coisas desse tipo no início (um pouco antes da página 1), mas sei que muitos leitores pulam isso, indo direto para o cerne do livro. Eu posso entender isso, então farei isso o mais rápido possível.

### **1.8.1 Apresentando programas completos como trechos**

Um dos desafios ao escrever um livro sobre uma linguagem de computador (além das linguagens de script) é que programas completos – aqueles que o leitor pode compilar e executar sem nenhum código-fonte além do apresentado – ficam longos muito rapidamente. Eu queria contornar isso, fornecer um código que você pudesse digitar e experimentar facilmente. Acredito que realmente *tentar* algo é uma maneira muito melhor de aprender do que apenas ler sobre isso.

Com as referências de assembly corretas e o uso correto de diretivas, você pode realizar muitas coisas com uma quantidade relativamente pequena de código C#, mas o matador é o fluff envolvido em escrever aquelas que usam diretivas, declarar uma classe e declarar um método Main antes. você escreveu a primeira linha de código *útil*. Meus exemplos são principalmente na forma de

*trechos*, que ignoram os boatos que atrapalham programas simples, concentrando-se nas partes importantes. Os snippets podem ser executados diretamente em uma pequena ferramenta que criei, chamada *Snippy*.

Se um trecho não contém reticências (...), então todo o código deve ser considerado o corpo do método Main de um programa. Se houver *reticências*, tudo antes dela será tratado como declarações de métodos e tipos aninhados, e tudo depois das reticências será tratado no método Main . Por exemplo, considere este trecho:

```
string estática reversa (entrada de string) {

    char[] chars = input.ToCharArray(); Array.Reverse(caracteres);
    retornar nova string (caracteres);

}

...
Console.WriteLine(Reverse("dlrow olleH"));

Isso é expandido por Snippy no seguinte:
```

usando o sistema;

trecho de classe pública {

```
    string estática reversa (entrada de string) {

        char[] chars = input.ToCharArray(); Array.Reverse(caracteres);
        retornar nova string (caracteres);

    }

    [STAThread]
    static void Main() {

        Console.WriteLine(Reverse("dlrow olleH"));
    }
}
```

Na realidade, o Snippy inclui muito mais diretivas de uso , mas a versão expandida já estava demorando. Observe que a classe que a contém sempre será chamada de Snippet e quaisquer tipos declarados no snippet serão aninhados nessa classe.

Há mais detalhes sobre como usar o Snippy no site do livro (<http://mng.bz/Lh82>), junto com todos os exemplos como trechos e versões expandidas em soluções do Visual Studio. Além disso, há suporte para LINQPad (<http://www.linqpad.net>) — uma ferramenta semelhante desenvolvida por Joe Albahari, com recursos particularmente úteis para explorar o LINQ.

A seguir, vamos ver o que há de errado com o código que acabamos de ver.

## 1.8.2 Código didático não é código de produção

Seria ótimo se você pudesse pegar todos os exemplos deste livro e usá-los diretamente em suas próprias aplicações, sem pensar mais... mas eu sugiro fortemente que você não faça isso. A maioria dos exemplos são apresentados para demonstrar um ponto específico – e isso geralmente é

aliado ao limite da intenção. A maioria dos snippets não inclui validação de argumentos, modificadores de acesso, testes unitários ou documentação. Eles também podem falhar quando usados fora do contexto pretendido.

Por exemplo, vamos considerar o corpo do método mostrado anteriormente para reverter uma linha. Eu uso esse código diversas vezes no decorrer do livro:

```
char[] chars = input.ToCharArray();
Array.Reverse(caracteres);
retornar nova string (caracteres);
```

Deixando de lado a validação de argumentos, isso consegue reverter a sequência de pontos de código UTF-16 dentro de uma string, mas em alguns casos isso não é bom o suficiente. Por exemplo, se um único glifo exibido for composto por um e seguido por um caractere de combinação representando um acento agudo, você não desejará alterar a sequência dos pontos de código; o acento acabará no personagem errado. Ou suponha que sua string contenha um caractere fora do plano multilíngue básico, formado a partir de um par substituto – reordenar os pontos de código levará a uma string que é efetivamente UTF-16 inválida. A correção desses problemas levaria a um código muito mais complicado, desviando a atenção do ponto que deveria ser demonstrado.

Você pode usar o código do livro, mas tenha esta seção em mente se fizer isso - seria muito melhor inspirar-se nele do que copiá-lo literalmente e presumir que atenderá às suas necessidades específicas. requisitos.

Finalmente, há outro livro que você deve baixar para aproveitar ao máximo este.

### **1.8.3 Seu novo melhor amigo: a especificação da linguagem**

Tentei ao máximo ser preciso neste livro, mas ficaria surpreso se não houvesse nenhum erro – na verdade, você encontrará uma lista de erros conhecidos no site do livro (<http://mng.bz/m1Hh>). Se você acha que encontrou um erro, ficaria grato se você pudesse me enviar um e-mail ([skeet@pobox.com](mailto:skeet@pobox.com)) ou adicionar uma nota no fórum do autor (<http://mng.bz/TQmF>). Mas você pode não querer esperar que eu responda ou pode ter uma pergunta que não foi abordada no livro. Em última análise, a fonte definitiva para o comportamento pretendido do C# é a especificação da linguagem.

Existem duas formas importantes de especificação: o padrão internacional da ECMA e a especificação da Microsoft. No momento em que escrevo isto, a especificação ECMA (ECMA- 334 e ISO/IEC 23270) cobre apenas C# 2, apesar de ser a quarta edição. Não está claro se ou quando isso será atualizado, mas a versão da Microsoft está completa e disponível gratuitamente. O site deste livro tem links para todas as versões disponíveis de ambos os tipos de especificação (<http://mng.bz/8s38>), e o Visual Studio também vem com uma cópia.<sup>5</sup> Quando me refiro às seções da especificação neste livro, eu 'usarei a numeração de

<sup>5</sup> A localização exata da especificação dependerá do seu sistema, mas na minha instalação do Visual Studio 2012 Professional, ela está em C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC#\Specifications\1033.

a especificação Microsoft C# 5, mesmo quando estou falando de versões anteriores da linguagem. Eu recomendo fortemente que você baixe esta versão e a tenha em mãos sempre que estiver ansioso para conferir um caso estranho.

Um dos meus objetivos é tornar a especificação *redundante* para os desenvolvedores – fornecer um formulário mais orientado ao desenvolvedor, cobrindo tudo o que você provavelmente verá no código do dia a dia, sem o enorme nível de detalhe exigido pelos autores do compilador. Dito isto, é extremamente legível no que diz respeito às especificações e você não deve se assustar com isso. Se você achar a especificação interessante, existem versões anotadas disponíveis para C# 3 e C# 4.

Ambos contêm comentários fascinantes da equipe C# e de outros colaboradores. (Isenção de responsabilidade: sou um dos “outros contribuidores” da edição C# 4...mas todos os *outros* os comentários são ótimos!)

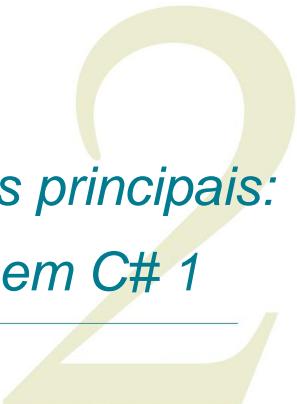
## 1.9 Resumo

Neste capítulo, mostrei (mas não expliquei) alguns dos recursos que serão abordados em profundidade no restante do livro. Há muito mais que não mostrei aqui, e muitos dos recursos que você viu até agora têm outros subrecursos associados a eles.

Esperamos que o que você viu aqui tenha aguçado seu apetite pelo resto do livro.

Embora os recursos tenham ocupado a maior parte do capítulo, também examinamos algumas áreas que devem ajudá-lo a aproveitar ao máximo o livro. Esclareci o que quero dizer quando me refiro à linguagem, ao tempo de execução e às bibliotecas, e também expliquei como o código será apresentado no livro.

Há mais uma área que precisamos abordar antes de nos aprofundarmos nos recursos do C# 2, que é o C# 1. Obviamente, como autor, não tenho ideia do quanto você conhece o C# 1, mas tenho alguma compreensão de quais áreas do C# 1 . C# geralmente causa problemas conceituais. Algumas dessas áreas são essenciais para obter o máximo das versões posteriores do C#, portanto, no próximo capítulo irei abordá-las com mais detalhes.



## Fundamentos principais: construindo em C# 1

### Este capítulo cobre

- ÿ Delegados
- ÿ Tipo de características do sistema
- ÿ Tipos de valor/referência

Isto não é uma atualização de todo o C# 1. Vamos tirar isso do caminho imediatamente. Eu não conseguia fazer justiça a *nenhum* tópico em C# se tivesse que cobrir toda a primeira versão em um único capítulo. Escrevi este livro presumindo que você seja pelo menos razoavelmente competente em C# 1. O que conta como “razoavelmente competente” é, obviamente, um tanto subjetivo, mas presumo que você pelo menos ficaria feliz em entrar *em* uma entrevista para uma função de desenvolvedor C# júnior e responda a perguntas técnicas apropriadas para esse trabalho. Você pode muito bem ter mais experiência, mas esse é o nível de conhecimento que estou assumindo.

Neste capítulo, focaremos em três áreas do C# 1 que são particularmente importantes para compreender os recursos de versões posteriores. Isso deve aumentar um pouco o mínimo denominador comum, para que eu possa fazer suposições um pouco maiores mais adiante no livro. Dado que se trata *de* um mínimo denominador comum, você pode descobrir que já possui um entendimento perfeito de todos os conceitos deste capítulo. Se você acredita que esse é o caso, mesmo sem ler o capítulo, sinta-se à vontade para ignorá-lo.

Você sempre pode voltar mais tarde se descobrir que algo não é tão simples quanto você pensava.

Se você não tiver certeza de que sabe tudo neste capítulo, talvez queira dar uma olhada no resumo no final de cada seção, que destaca os pontos importantes – se algum deles lhe parecer estranho, vale a pena ler essa seção detalhadamente.

Começaremos examinando os delegados, depois consideraremos como o sistema de tipos C# se compara a algumas outras possibilidades e, finalmente, examinaremos as diferenças entre tipos de valor e tipos de referência. Para cada tópico descreverei as ideias e comportamentos, além de aproveitar para definir termos para poder utilizá-los posteriormente. Depois de vermos como o C# 1 funciona, mostrarei uma rápida prévia de quantos dos novos recursos em versões posteriores estão relacionados aos tópicos examinados neste capítulo.

## 2.1 Delegados

Tenho certeza de que você já tem uma ideia intuitiva sobre o que é um delegado, mesmo que seja difícil articular. Se você estiver familiarizado com C e tiver que descrever delegados para outro programador C, o termo *ponteiro de função* sem dúvida surgirá. Essencialmente, os delegados fornecem um nível de indireção: em vez de especificar o comportamento a ser executado imediatamente, o comportamento pode de alguma forma ser “contido” em um objeto. Esse objeto pode então ser usado como qualquer outro, e uma operação que você pode realizar com ele é executar a ação encapsulada. Como alternativa, você pode pensar em um tipo delegado como uma interface de método único e em uma instância delegada como um objeto que implementa essa interface.

Se isso é apenas uma bobagem para você, talvez um exemplo ajude. É um pouco móbido, mas capta a essência dos delegados. Considere a sua vontade – a sua última vontade e testamento. É um conjunto de instruções: “pagar as contas, fazer uma doação para caridade, deixar o resto do meu patrimônio para o gato”, por exemplo. Você o escreve *antes de morrer* e o deixa em um local devidamente seguro. *Após* sua morte, seu advogado (você espera!) agirá de acordo com essas instruções.

Um delegado em C# age como sua vontade no mundo real – ele permite que você especifique uma sequência de ações a serem executadas no momento apropriado. Os delegados são normalmente usados quando o código que deseja executar as ações não conhece os detalhes do que está acontecendo.

essas ações deveriam ser. Por exemplo, a única razão pela qual a classe Thread sabe o que executar em um novo thread quando você o inicia é porque você fornece ao construtor uma instância delegada ThreadStart ou ParameterizedThreadStart .

Começaremos nosso tour pelos delegados com os quatro princípios básicos absolutos, sem os quais ninguém do resto faria sentido.

### 2.1.1 Uma receita para delegados simples

Para que um delegado faça qualquer coisa, quatro coisas precisam acontecer: *ÿ O tipo do delegado* precisa ser declarado. *ÿ O código a ser executado* deve estar contido em um método.

*ÿ* Uma *instância delegada* deve ser criada. *ÿ* A instância delegada deve ser *invocada*.

Vamos dar cada passo desta receita por vez.

## DECLARANDO O TIPO DE DELEGADO

Um tipo delegado é efetivamente uma lista de tipos de parâmetros e um tipo de retorno. Especifica que tipo de ação pode ser representada por instâncias do tipo.

Por exemplo, considere um tipo delegado declarado assim:

```
delegar void StringProcessor (entrada de string);
```

O código diz que se você quiser criar uma instância de StringProcessor, você precisará de um método com um parâmetro (uma string) e um tipo de retorno void (o método não retorna nada).

É importante entender que StringProcessor é realmente um tipo derivado de System.MulticastDelegate, que por sua vez deriva de System.Delegate. Tem métodos, você pode criar instâncias dele e passar referências para instâncias, tudo funciona. Obviamente, existem alguns recursos especiais, mas se você estiver pensando no que acontecerá em uma situação específica, primeiro pense no que aconteceria se você estivesse usando um tipo de referência normal.

**FONTE DE CONFUSÃO: O TERMO AMBÍGUO DELEGADO** Delegados podem ser mal compreendidos porque a palavra delegado é freqüentemente usada para descrever um tipo de delegado e uma instância de delegado. A distinção entre esses dois é exatamente a mesma que entre qualquer outro tipo e instâncias desse tipo — o tipo string em si é diferente de uma sequência específica de caracteres, por exemplo. Usei os termos tipo de delegado e instância de delegado ao longo deste capítulo para tentar deixar claro exatamente do que estou falando a qualquer momento.

Usaremos o tipo delegado StringProcessor quando considerarmos o próximo ingrediente.

## ENCONTRAR UM MÉTODO ADEQUADO PARA A AÇÃO DA INSTÂNCIA DELEGADA

O próximo ingrediente é encontrar (ou escrever) um método que faça o que você deseja e tenha a mesma assinatura do tipo de delegado que você está usando. A ideia é ter certeza de que, quando você tentar invocar uma instância delegada, todos os parâmetros usados corresponderão e você poderá usar o valor de retorno (se houver) da maneira esperada - como um valor normal. chamada de método.

Considere essas cinco assinaturas de método como candidatas a serem usadas para uma String Instância de processador:

```
void PrintString(string x) void  
PrintInteger(int x) void  
PrintTwoStrings(string x, string y) int GetStringLength(string x)  
void PrintObject(objeto x)
```

O primeiro método tem tudo certo, então você pode usá-lo para criar uma instância delegada.

O segundo método possui um parâmetro, mas não é string, portanto é incompatível com StringProcessor. O terceiro método possui o primeiro tipo de parâmetro correto, mas também possui outro parâmetro, portanto ainda é incompatível. O quarto método possui a lista de parâmetros correta, mas um tipo de retorno não nulo. (Se o seu tipo delegado tivesse um tipo de retorno, o tipo de retorno do método também teria que corresponder a ele.)

O quinto método é interessante: sempre que você invocar uma instância de StringProcessor , poderá chamar o método PrintObject com os mesmos argumentos, porque string deriva de object. Faria sentido poder usá-lo para uma instância de StringProcessor, mas em C# 1 o delegado deve ter *exatamente* os mesmos tipos de parâmetros.<sup>1</sup> O C# 2 muda essa situação — consulte o capítulo 5 para obter mais detalhes. De certa forma, o quarto método é semelhante, pois você sempre pode ignorar o valor de retorno indesejado.

Mas os tipos de retorno void e nonvoid são atualmente sempre considerados incompatíveis.

Isso ocorre em parte porque outros aspectos do sistema (particularmente o JIT) precisam saber se um valor será deixado na pilha como valor de retorno quando um método for executado.<sup>2</sup> Vamos

supor que você tenha um corpo de método para a assinatura compatível (PrintString ) e passe para o próximo ingrediente – a própria instância delegada.

#### CRIANDO UMA INSTÂNCIA DELEGADA

Agora que você tem um tipo delegado e um método com a assinatura correta, você pode criar uma instância desse tipo delegado, especificando que esse método seja executado quando a instância delegada for invocada. Nenhuma terminologia oficial foi definida para isso, mas neste livro chamarei isso de *ação* da instância delegada.

A forma exata da expressão usada para criar a instância delegada depende se a ação usa um método de instância ou um método estático. Suponha que PrintString seja um método estático em um tipo chamado StaticMethods e um método de instância em um tipo chamado InstanceMethods.

Aqui estão dois exemplos de criação de uma instância de String-

Processador:

```
StringProcessor proc1, proc2; proc1 = novo
StringProcessor(StaticMethods.PrintString); Instância de InstanceMethods = new
InstanceMethods(); proc2 = novo StringProcessor(instance.PrintString);
```

Quando a ação é um método estático, você só precisa especificar o nome do tipo. Quando a ação é um método de instância, você precisa de uma instância do tipo (ou de um tipo derivado), como faria normalmente. Este objeto é chamado de *alvo* da ação e, quando a instância delegada for invocada, o método será chamado naquele objeto. Se a ação estiver dentro da mesma classe (como costuma acontecer, especialmente quando você está escrevendo manipuladores de eventos no código da UI ), você não precisa qualificá-la de qualquer maneira — a referência this é usada implicitamente para métodos de instância.<sup>3</sup> Novamente , essas regras agem como se você estivesse chamando o método diretamente.

**LIXO COMPLETO! (OU NÃO, CONFORME O CASO)** Vale a pena estar ciente de que uma instância delegada impedirá que seu destino seja coletado como lixo se a própria instância delegada não puder ser coletada. Isto pode resultar em aparente

<sup>1</sup> Além dos tipos de parâmetros, você deve combinar se o parâmetro é in (o padrão), out ou ref. No entanto , é razoavelmente raro usar parâmetros out e ref com delegados.

<sup>2</sup> Este é um uso deliberadamente vago da palavra *pilha* para evitar entrar em muitos detalhes irrelevantes. Veja Eric Lippert do blog de pert “O vazio é invariante” para obter mais informações (<http://mng.bz/4g58>).

<sup>3</sup> Obviamente, se a ação for um método de instância e você estiver tentando criar uma instância delegada a partir de um método estático, ainda precisará fornecer uma referência para ser o alvo.

vazamentos de memória, especialmente quando um objeto de curta duração se inscreve em um evento em um objeto de longa duração, usando a si mesmo como alvo. O objeto de vida longa mantém indiretamente uma referência ao objeto de vida curta, prolongando sua vida útil.

Não faz muito sentido criar uma instância delegada se ela não for invocada em algum momento. Vejamos a última etapa: a invocação.

#### INVOCANDO UMA INSTÂNCIA DELEGADA

Invocar uma instância delegada é a parte realmente fácil:<sup>4</sup> é apenas uma questão de chamar um método na instância delegada. O método em si é chamado `Invoke` e está sempre presente em um tipo delegado com a mesma lista de parâmetros e tipo de retorno especificado pela declaração do tipo delegado. Em nosso exemplo contínuo, existe um método como este:

```
void Invoke (entrada de string)
```

Chamar `Invoke` executará a ação da instância delegada, passando quaisquer argumentos que você especificou na chamada para `Invoke` e (se o tipo de retorno não for `void`) retornando o valor de retorno da ação.

Por mais simples que seja, C# torna tudo ainda mais fácil; se você tiver uma variável<sup>5</sup> cujo tipo é delegado, você pode trate-o como se fosse um método em si. É mais fácil ver isso acontece como uma cadeia de eventos que ocorrem em momentos diferentes, como mostra a figura 2.1.

Como você pode ver, isso também é simples. Todos os ingredientes estão agora no lugar, então você pode pré-aquecer seu CLR a 200°C, misturar tudo e ver o que acontece.

#### UM EXEMPLO COMPLETO E ALGUMA MOTIVAÇÃO

É mais fácil ver tudo isso em ação em um exemplo completo – finalmente, algo que você pode realmente executar! Como há muitos pedaços envolvidos, desta vez inclui todo o código-fonte, em vez de usar trechos. Não há nada de surpreendente na listagem a seguir, então não espere ficar surpreso – é apenas útil ter um código concreto para discutir.

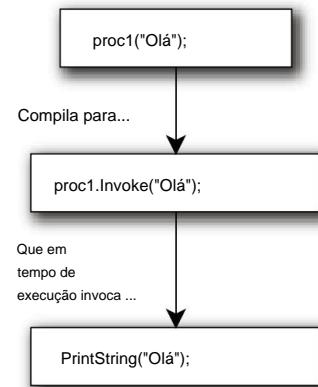


Figura 2.1 Processando uma chamada para uma instância delegada que usa a sintaxe abreviada C#

#### Listagem 2.1 Usando delegados de diversas maneiras simples

```
usando o sistema;
delegar void StringProcessor (entrada de string); classe Pessoa
{
    nome da sequência;
    public Person(string nome) { this.name = nome; }
```

← B Declara o tipo de delegado

<sup>4</sup> Pelo menos para invocação síncrona. Você pode usar `BeginInvoke` e `EndInvoke` para invocar uma instância delegada de forma assíncrona, mas isso está além do escopo deste capítulo.

<sup>5</sup> Ou qualquer outro tipo de expressão – mas geralmente é uma variável.

```

public void Say (mensagem de string)
{
    Console.WriteLine("{0} diz: {1}", nome, mensagem);
}

fundo da classe
{
    Nota pública estática void (nota de string)
    {
        Console.WriteLine("{0}", nota);
    }
}
classe SimpleDelegateUse
{
    vazio estático Principal()
    {
        Pessoa jon = new Pessoa("Jon");
        Pessoa tom = new Pessoa("Tom");
        StringProcessor jonsVoice, tomsVoice, background;
        jonsVoice = new StringProcessor(jon.Say);
        tomsVoice = new StringProcessor(tom.Say);
        background = new StringProcessor(Background.Note);
        jonsVoice("Olá, filho.");
        tomsVoice.Invoke("Olá, papai!");
        background("Um avião passa voando.");
    }
}

```

**A** Declara instância compatível

**B** Declara método estático compatível

**C** Cria três instâncias delegadas

**D** Invoca instâncias delegadas

Para começar, você declara o tipo delegado B. Em seguida, você cria dois métodos (C) e (D) que sejam compatíveis com o tipo delegado. Você tem uma instância método (Person.Say) e um método estático (Background.Note), então você verá como eles são usados de maneira diferente quando você cria as instâncias delegadas E. Listagem 2.1 inclui duas instâncias da classe Person , para que você possa ver a diferença que o destino de um delegado faz.

Quando jonsVoice é invocado F, ele chama o método Say no objeto Person com o nome Jon; da mesma forma, quando tomsVoice é invocado, ele usa o objeto com o nome Tom. Este código inclui as duas maneiras de invocar instâncias delegadas que você viu— chamando Invoke explicitamente e usando a abreviatura C# — apenas por uma questão de interesse. Normalmente você usaria a abreviatura.

O resultado da listagem 2.1 é bastante óbvio:

```

Jon diz: Olá, filho.
Tom diz: Olá, papai!
(Um avião passa voando.)

```

Francamente, há uma enorme quantidade de código na listagem 2.1 para exibir três linhas de saída. Até se você quiser usar a classe Person e a classe Background , não há necessidade real de use delegados aqui. Então qual é o objetivo? Por que não chamar os métodos diretamente? O A resposta está em nosso exemplo original de um advogado executando um testamento - só porque você querer que algo aconteça não significa que você está sempre lá na hora certa e

lugar para fazer isso acontecer. Às vezes você precisa dar instruções – *delegar* responsabilidades, por assim dizer.

Devo enfatizar que, no mundo do software, não se trata de objetos deixando desejos de morrer. Freqüentemente, o objeto que primeiro cria uma instância delegada ainda está ativo e bem quando a instância delegada é invocada. Em vez disso, trata-se de especificar algum código para ser executado em um determinado momento, quando você pode não ser capaz (ou não querer) altere o código que está sendo executado naquele ponto. Se eu quiser que algo aconteça quando um botão for clicado, não quero ter que alterar o código do *botão* - só quero diga ao botão para chamar um dos meus métodos, que tomará a ação apropriada. É um questão de adicionar um nível de *indireção*, como acontece com grande parte da programação orientada a objetos. Como você viu, isso adiciona complexidade (veja quantas linhas de código foram necessárias para produzir tão pouca saída!), mas também flexibilidade.

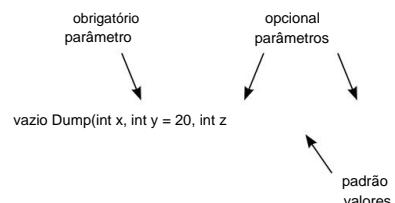
Agora que você entende mais sobre delegados simples, daremos uma breve olhada em combinando delegados para executar um monte de ações em vez de apenas uma.

### 2.1.2 Combinando e removendo delegados

Até agora, todas as instâncias de delegação que analisamos tiveram uma única ação. Na realidade, A vida é um pouco mais complicada: uma instância delegada na verdade tem uma lista de ações associadas a ela chamada *lista de invocação*. Os métodos estáticos Combine e Remove do O tipo System.Delegate é responsável por criar novas instâncias delegadas, unindo respectivamente as listas de invocação de duas instâncias delegadas ou removendo o lista de invocação de uma instância delegada de outra.

**DELEGADOS SÃO IMUTÁVEIS** Depois de criar uma instância delegada, nada nela poderá ser alterado. Isso torna seguro passar referências a delegar instâncias e combiná-las com outras sem se preocupar com consistência, segurança do thread ou qualquer pessoa tentando mudar suas ações. Isso é como strings, que também são imutáveis, e Delegate.Combine é exatamente como String.Concat - ambos combinam instâncias existentes para formar um novo sem alterar o original objetos em tudo. No caso do delegado instâncias, as listas de invocação originais são concatenados juntos. Observe que se você já tente combinar null com um delegado Por exemplo, o nulo é tratado como se fosse um instância delegada com uma lista de invocação vazia.

Você raramente verá uma chamada explícita para Delegate .Combine em código C# — geralmente os operadores + e += são usados. A Figura 2.2 mostra a tradução processo, onde x e y são variáveis do mesmos tipos de delegados (ou compatíveis). Tudo isso é feito pelo compilador C#.



**Figura 2.2** O processo de transformação usado para a sintaxe abreviada C# para combinar instâncias delegadas

Como você pode ver, é uma transformação simples, mas torna o código muito mais organizado. Assim como você pode combinar instâncias delegadas, você pode remover uma da outra com o método `Delegate.Remove`, e o C# usa a abreviação dos operadores - `e -=` da maneira óbvia. `Delegate.Remove(source, value)` cria um novo delegado cuja lista de invocação é aquela da fonte, com a lista do valor tendo sido removida. Se o resultado tiver uma lista de invocação vazia, será retornado nulo.

Quando uma instância delegada é invocada, todas as suas ações são executadas em ordem. Se a assinatura do delegado tiver um tipo de retorno não nulo, o valor retornado por `Invoke` será o valor retornado pela última ação executada. É raro ver uma instância delegada não nula com mais de uma ação em sua lista de invocação porque isso significa que os valores de retorno de todas as outras ações nunca são vistos, a menos que o código de chamada execute explicitamente as ações, uma de cada vez, usando `Delegate.GetInvocationList` para buscar a lista de ações.

Se alguma das ações na lista de invocação gerar uma exceção, isso impedirá que qualquer uma das ações subsequentes seja executada. Por exemplo, se uma instância delegada com uma lista de invocação `[a, b, c]` for invocada e a ação `b` lançar uma exceção, a exceção será propagada imediatamente e a ação `c` não será executada.

Combinar e remover instâncias delegadas é particularmente útil quando se trata de eventos. Agora que você entende o que envolve combinar e remover, podemos falar sobre eventos.

### 2.1.3 Um breve desvio para os acontecimentos

Você provavelmente tem uma ideia intuitiva sobre o objetivo geral dos eventos, principalmente se tiver escrito alguma UI. A ideia é que um evento permita que o código reaja quando algo acontecer – salvando um arquivo quando o botão apropriado for clicado, por exemplo. Neste caso, o evento é o clique do botão e a ação é o salvamento do arquivo.

No entanto, compreender a razão do conceito não é o mesmo que entender como o C# define eventos em termos de linguagem.

Os desenvolvedores muitas vezes confundem eventos e instâncias de delegação, ou eventos e campos declarados com tipos de delegação. A diferença é importante: eventos não são campos. A razão para a confusão é que C# fornece uma abreviação na forma de eventos semelhantes a campos. Chegaremos a isso em um minuto, mas primeiro vamos considerar em que consistem os eventos no que diz respeito ao compilador C#.

É útil pensar nos eventos como semelhantes às propriedades. Para começar, ambos eles são declarados como sendo de um determinado tipo – um evento é迫使ido a ser do tipo delegado.

Quando você usa propriedades, parece que você está buscando ou atribuindo valores diretamente aos campos, mas na verdade você está chamando métodos (getters e setters). A implementação da propriedade pode fazer o que quiser dentro desses métodos - acontece que a maioria das propriedades são implementadas com campos simples que as apoiam, às vezes com alguma validação no setter e às vezes com alguma segurança de thread incluída para sempre. medir.

Da mesma forma, quando você assina ou cancela a assinatura de um evento, parece que você está usando um campo cujo tipo é delegado, com os operadores `+ =` e `- =`. De novo,

porém, na verdade você está chamando métodos (adicionar e remover).<sup>6</sup> Isso é tudo que você pode fazer com um evento - inscreva-se nele (adicone um manipulador de eventos) ou cancele a assinatura dele (remova um manipulador de eventos). Cabe aos métodos de evento fazer algo útil, como tirar aviso dos manipuladores de eventos que você está tentando adicionar e remover e disponibilizá-los em outro lugar da classe.

A razão para ter eventos em primeiro lugar é muito parecida com a razão para ter propriedades - elas adicionam uma camada de encapsulamento, implementando o recurso de publicação/assinatura padrão (veja meu artigo, "Delegados e Eventos", aqui: <http://mng.bz/HPx6>). Assim como você não deseja que outro código seja capaz de definir valores de campo sem que o proprietário tenha pelo menos a opção de validar o novo valor, muitas vezes você não deseja que o código fora de uma classe ser capaz de alterar (ou chamar) arbitrariamente os manipuladores de um evento. É claro que uma classe *pode* adicione métodos para fornecer acesso extra – por exemplo, para redefinir a lista de manipuladores para um evento, ou para acionar o evento (em outras palavras, para chamar seus manipuladores de eventos). Por exemplo, BackgroundWorker.OnProgressChanged apenas chama os manipuladores de eventos ProgressChanged . Mas se você expor apenas o evento em si, o código fora da classe só terá a capacidade para adicionar e remover manipuladores.

*Eventos semelhantes a campo* tornam a implementação de tudo isso muito mais simples de se observar – uma declaração única e pronto. O compilador transforma a declaração em um evento com implementações padrão de adição/remoção e um campo privado do mesmo tipo. O código dentro da classe vê o campo; o código fora da classe vê apenas o evento. Esse faz *parecer* que você pode invocar um evento, mas o que você realmente faz para chamar o evento manipuladores é invocar a instância delegada armazenada no campo.

Os detalhes dos eventos estão fora do escopo deste capítulo – os próprios eventos não mudou muito nas versões posteriores do C#,<sup>7</sup> mas gostaria de chamar a atenção para o diferença entre instâncias de delegação e eventos agora, para evitar confusão mais tarde.

## 2.1.4 Resumo dos delegados

Vamos resumir o que abordamos sobre delegados:

- ÿ Os delegados encapsulam o comportamento com um tipo de retorno específico e um conjunto de parâmetros ters, semelhante a uma interface de método único.
- ÿ A assinatura de tipo descrita por uma declaração de tipo delegado determina quais métodos podem ser usados para criar instâncias delegadas e a assinatura para invocação.
- ÿ Criar uma instância delegada requer um método e (para métodos de instância) um target para chamar o método.
- ÿ As instâncias delegadas são imutáveis.
- ÿ Cada instância delegada contém uma lista de invocação – uma lista de ações.

---

<sup>6</sup> Estes não são seus nomes no código compilado; caso contrário, você poderia ter apenas um evento por tipo. O compilador cria dois métodos com nomes que não são usados em nenhum outro lugar e inclui uma parte especial de metadados para deixe outros tipos saberem que há um evento com o nome fornecido e como são chamados seus métodos de adição/remoção.

<sup>7</sup> Há mudanças muito pequenas em eventos semelhantes a campos no C# 4. Consulte a seção 4.2 para obter detalhes.

↳ As instâncias delegadas podem ser combinadas e removidas umas das outras. ↳ Eventos não são instâncias delegadas — eles são apenas pares de métodos de adição/remoção (pense em getters/setters de propriedades).

Os delegados são um recurso específico do C# e do .NET — um detalhe no grande esquema das coisas. Ambas as outras seções de lembretes deste capítulo tratam de tópicos muito mais amplos. Primeiro, consideraremos o que significa falar sobre C# ser uma linguagem *de tipo estaticamente* e as implicações que isso tem.

## 2.2 Características do sistema de tipo

Quase toda linguagem de programação possui algum tipo de sistema de tipos. Com o tempo, estes foram classificados como fortes/fracos, seguros/inseguros, estáticos/dinâmicos e, sem dúvida, algumas variações mais esotéricas. Obviamente, é importante entender o sistema de tipos com o qual você está trabalhando, e é razoável esperar que conhecer as categorias nas quais uma linguagem se enquadra forneceria muitas informações sobre esse assunto. Mas como os termos são usados por pessoas diferentes para significar coisas um tanto diferentes, a má comunicação é quase inevitável. Tentarei explicar *exatamente* o que quero dizer com cada termo para minimizar confusão.

Uma coisa importante a observar é que esta seção só se aplica a código *seguro*, o que significa todo código C# que não esteja explicitamente dentro de um contexto inseguro. Como você pode julgar pelo nome, o código dentro de um contexto inseguro pode fazer várias coisas que o código seguro não pode, e isso pode violar alguns aspectos da segurança normal de tipos, embora o sistema de tipos ainda seja seguro de muitas outras maneiras. É improvável que a maioria dos desenvolvedores precise escrever código inseguro, e as características do sistema de tipos são muito mais simples de descrever e entender quando apenas o código seguro é considerado.

Esta seção mostra quais restrições são ou não aplicadas no C# 1 ao definir alguns termos para descrever esse comportamento. Veremos então algumas coisas que você não pode fazer com o C# 1 — primeiro do ponto de vista do que você *não pode* dizer ao compilador e depois do ponto de vista do que você gostaria de não dizer. *tem* que informar o compilador.

Vamos começar com o que o C# 1 faz e com a terminologia normalmente usada para descrever esse tipo de comportamento.

### 2.2.1 O lugar do C# no mundo dos sistemas de tipos

É mais fácil começar fazendo uma declaração e depois esclarecer o que ela significa e quais poderiam ser as alternativas:

O sistema de tipos do C# 1 é *estático, explícito e seguro*.

Você poderia esperar que a palavra *forte* aparecesse na lista, e eu estava pensando em incluí-la. Mas embora a maioria das pessoas possa concordar razoavelmente sobre se uma linguagem tem as características que listei, decidir se uma linguagem é *fortemente tipada* pode causar um debate acalorado porque as definições variam muito. Alguns significados (aqueles que impedem quaisquer conversões, explícitas ou implícitas) excluiriam claramente o C#, enquanto outros são bastante próximos (ou até mesmo iguais) de tipo estaticamente, o que incluiria C# 1. A *maioria* dos

os artigos e livros que li que descrevem C# como uma linguagem fortemente tipada estão efetivamente usando “strongly typed” para significar tipagem estaticamente.

Vamos pegar os termos da definição, um de cada vez, e lançar alguma luz sobre eles.

#### DIGITAÇÃO ESTÁTICA VERSUS DIGITAÇÃO DINÂMICA

C# 1 é tipado estaticamente: cada variável é de um tipo específico e esse tipo é conhecido em tempo de compilação.<sup>8</sup> Somente operações conhecidas para esse tipo são permitidas, e isso é imposto pelo compilador. Considere este exemplo de aplicação:

```
objeto o = "olá";
Console.WriteLine(o.Comprimento);
```

Ao observar o código, é óbvio que o valor de o se refere a uma string e que o tipo string tem uma propriedade Length, mas o compilador só pensa em o como sendo do tipo object. Se você quiser chegar à propriedade Length, deverá informar ao compilador que o valor de o se refere a uma string:

```
objeto o = "olá";
Console.WriteLine(((string)o).Length);
```

O compilador é então capaz de encontrar a propriedade Length de System.String. Ele usa isso para validar se a chamada está correta, emitir o IL apropriado e calcular o tipo da expressão maior. O tipo de tempo de compilação de uma expressão também é conhecido como tipo estático – então você pode dizer: “O tipo estático de o é System.Object”.

**POR QUE É CHAMADA DIGITAÇÃO ESTÁTICA?** A palavra estática é usada para descrever esse tipo de digitação porque a análise de quais operações estão disponíveis é realizada usando dados imutáveis: os tipos de expressões em tempo de compilação. Suponha que uma variável seja declarada como sendo do tipo Stream; o tipo da variável não muda, mesmo que o valor da variável varie de uma referência a um MemoryStream, a um FileStream ou a nenhum fluxo (com uma referência nula).

Mesmo em sistemas do tipo estático, pode haver algum comportamento dinâmico; a implementação real executada por uma chamada de método virtual dependerá do valor em que é chamado. A ideia de informação imutável também é a motivação por trás do modificador estático, mas geralmente é mais simples pensar em um membro estático como pertencente ao próprio tipo do que a qualquer instância específica do tipo. Para fins mais práticos, você pode pensar nos dois usos da palavra como não relacionados.

A alternativa à digitação estática é a digitação dinâmica, que pode assumir vários disfarces. A essência da tipagem dinâmica é que as variáveis possuem apenas valores – elas não estão restritas a tipos específicos, portanto o compilador não pode realizar o mesmo tipo de verificações. Em vez disso, o ambiente de execução tenta compreender as expressões de maneira adequada aos valores envolvidos. Por exemplo, se C# 1 fosse digitado dinamicamente, você poderia fazer o seguinte:

---

<sup>8</sup> Isso também se aplica à maioria das expressões, mas não a todas. Certas expressões não têm um tipo, como invocações de método void, mas isso não afeta o status do C# 1 de ser digitado estaticamente. Usei a palavra variável ao longo desta seção para evitar esforço cerebral desnecessário.

E SE?

```

o = "olá";
Console.WriteLine(o.Comprimento);
o = nova string[] {"oi", "lá"};
Console.WriteLine(o.Comprimento);

```

Isso invocaria duas propriedades Length completamente não relacionadas - String.Length e Array.Length — examinando os tipos dinamicamente em tempo de execução. Como muitos aspectos dos sistemas de tipos, existem diferentes níveis de tipagem dinâmica. Algumas línguas permitem que você especifique os tipos onde desejar - possivelmente ainda tratando-os dinamicamente além da atribuição, mas permite usar variáveis não digitadas em outro lugar.

Embora eu tenha especificado o C# 1 repetidamente nesta descrição, o C# foi totalmente digitado estaticamente até e incluindo o C# 3. Você verá mais tarde que o C# 4 introduziu algumas digitação dinâmica, embora a grande maioria do código na maioria dos aplicativos C# 4 ainda use digitação estática.

#### DIGITAÇÃO EXPLÍCITA VERSUS DIGITAÇÃO IMPLÍCITA

A distinção entre *tipagem explícita* e *tipagem implícita* só é relevante em situações estaticamente linguagens digitadas. Com digitação explícita, o tipo de cada variável deve ser explicitamente indicado na declaração. A digitação implícita permite ao compilador inferir o tipo do variável com base em seu uso. Por exemplo, a linguagem poderia ditar que o tipo de variável é o tipo de expressão usada para atribuir o valor inicial.

Considere uma linguagem hipotética que usa a palavra-chave var para indicar inferência de tipo.<sup>9</sup> A Tabela 2.1 mostra como o código dessa linguagem poderia ser escrito em C# 1. A o código na coluna da esquerda *não* é permitido em C# 1, mas o código na coluna da direita é o código válido equivalente.

**Tabela 2.1 Um exemplo mostrando as diferenças entre digitação implícita e explícita**

C# 1 inválido – digitação implícita	C# 1 válido – digitação explícita
vars = "olá";	strings = "olá";
var x = s.Comprimento;	int x = s.Comprimento;
var duas vezesX = x *2;	int duas vezesX = x *2;

Esperamos que esteja claro por que isso só é relevante para situações de tipo estaticamente: para ambos tipagem implícita e explícita, o tipo da variável é *conhecido* em tempo de compilação, mesmo que não está explicitamente declarado. Em um contexto dinâmico, a variável nem sequer *tem* um tipo de tempo de compilação para declarar ou inferir.

#### TIPO SEGURO VERSUS TIPO INSEGURO

A maneira mais fácil de descrever um sistema com segurança de tipo é descrever seu oposto. Algumas linguagens (estou pensando particularmente em C e C++) permitem que você faça algumas ações realmente tortuosas. coisas. Eles são potencialmente poderosos nas situações certas, mas com grande poder vem uma caixa de donuts grátis, ou como diz a expressão, e as situações certas

<sup>9</sup> Ok, não é tão hipotético. Consulte a seção 8.2 para conhecer os recursos de variáveis locais digitadas implicitamente do C# 3.

são relativamente raros. Algumas dessas coisas tortuosas podem atirar no seu pé se você errar. Abusar do sistema de tipos é um deles.

Com os rituais vodu corretos, você pode persuadir essas linguagens a tratar um valor de um tipo como se fosse um valor de um tipo *completamente* diferente, sem aplicar quaisquer conversões. Não me refiro apenas a chamar um método que tenha o mesmo nome, como no exemplo de digitação dinâmica anterior. Quero dizer, código que analisa os bytes brutos dentro de um valor e os interpreta da maneira “errada”. A listagem a seguir fornece um exemplo simples em C do que quero dizer.

#### Listagem 2.2 Demonstrando um sistema sem segurança de tipo com código C

```
#include <stdio.h>
int principal(int argc, char**argv) {

    char *primeiro_arg = argv[1]; int
    *first_arg_as_int = (int *)first_arg; printf("%d", *first_arg_as_int);

}
```

Se você compilar a listagem 2.2 e executá-la com um argumento simples de "hello", você verá um valor de 1819043176 - pelo menos em uma arquitetura little-endian com um compilador tratando int como 32 bits e char como 8 bits, e onde o texto é representado em ASCII ou UTF-8. O código está tratando o ponteiro char como um ponteiro int , portanto, desreferenciando ele retorna os primeiros 4 bytes de texto, tratando-os como um número.

Na verdade, este pequeno exemplo é inofensivo em comparação com outros abusos potenciais – a conversão entre estruturas completamente não relacionadas pode facilmente resultar em caos total. Não é que isso aconteça na vida real com muita frequência, mas alguns elementos do sistema de digitação C geralmente exigem que você diga ao compilador o que fazer, não deixando outra opção a não ser confiar em você mesmo no tempo de execução.

Felizmente, nada disso ocorre em C#. Sim, há muitas conversões disponíveis, mas você não pode fingir que os dados de um tipo específico de objeto são, na verdade, dados de um tipo diferente. Você pode *tentar* adicionar uma conversão para fornecer ao compilador essas informações extras (e incorretas), mas se o compilador detectar que é realmente *impossível* que essa conversão funcione, isso causará um erro de compilação - e se for teoricamente permitido, mas na verdade totalmente incorreto em tempo de execução, o CLR lançará uma exceção.

Agora que você sabe um pouco sobre como o C# 1 se encaixa no panorama geral dos sistemas de tipos, gostaria de mencionar algumas desvantagens de suas escolhas. Isso não quer dizer que as escolhas sejam erradas – elas são apenas limitantes em alguns aspectos. Freqüentemente, os designers de linguagem precisam escolher entre diferentes caminhos que acrescentam diferentes limitações ou têm outras consequências indesejáveis. Começarei com o caso em que você *deseja* fornecer mais informações ao compilador, mas não há como fazê-lo.

#### 2.2.2 Quando o sistema de tipos do C# 1 não é rico o suficiente?

Existem duas situações comuns em que você pode querer expor mais informações ao chamador de um método, ou talvez forçar o chamador a limitar o que ele fornece em seu método.

argumentos. O primeiro envolve coleções e o segundo envolve herança e substituição de métodos ou implementação de interfaces. Examinaremos cada um por vez.

## COLEÇÕES, FORTES E FRACAS

Tendo evitado os termos *fortes* e *fracas* para o sistema do tipo C# em geral, vou usá-los ao falar sobre coleções. Os termos são usados em quase todos os lugares neste contexto, com pouco espaço para ambiguidade. Em termos gerais, três tipos de coleção são integrados ao .NET 1.1:

- ÿ Arrays — fortemente tipados — tanto na linguagem quanto no tempo de execução
  - ÿ Coleções com tipagem fraca no namespace System.Collections
  - ÿ Coleções com tipagem forte no System.Collections.Specialized
- espaço para nome

Arrays são fortemente tipados,<sup>10</sup> portanto em tempo de compilação você não pode definir um elemento de uma `string[]` como um `FileStream`, por exemplo. Mas matrizes de tipo de referência também suportam *covariância*, que fornece uma conversão implícita de um tipo de matriz para outro, desde que haja uma conversão entre os tipos de elementos. As verificações ocorrem em tempo de execução para garantir que o tipo errado de referência não esteja sendo armazenado, conforme mostrado na listagem a seguir.

### Listagem 2.3 Demonstração de covariância de array e verificação de tempo de execução

```
string[] strings = nova string[5]; objeto[] objetos =
strings; objetos[0] = novo botão();
```

Se você executar a listagem 2.3, verá que um `ArrayTypeMismatchException` é lançado em C. Isso ocorre porque a conversão de `string[]` para `object[]` **B** retorna a referência original – tanto `strings` quanto `objetos` referem-se ao mesmo array. O próprio array sabe que é um array de `strings` e rejeitará tentativas de armazenar referências a não-`strings`. A covariância de array é ocasionalmente útil, mas tem o custo de implementar parte do tipo de segurança em tempo de execução, em vez de tempo de compilação.

Vamos comparar isso com a situação em que coleções de tipo fraco, como `ArrayList` e `Hashtable`, colocam você. A API dessas coleções usa `objeto` como tipo de chaves e valores. Quando você escreve um método que usa um `ArrayList`, por exemplo, não há como garantir, em tempo de compilação, que o chamador passará uma lista de `strings`. Você pode documentá-lo, e a segurança de tipo do tempo de execução irá aplicá-la se você converter cada elemento da lista em `string`, mas não obterá segurança de tipo em tempo de compilação.

Da mesma forma, se você retornar um `ArrayList`, poderá indicar na documentação que ele conterá apenas `strings`, mas os chamadores terão que confiar que você está dizendo a verdade e terão que inserir conversões quando acessarem os elementos da lista.

<sup>10</sup> Pelo menos, a linguagem permite que sejam. No entanto, você pode usar o tipo `Array` para acesso de tipo fraco a arrays.

Finalmente, considere coleções fortemente tipadas, como `StringCollection`. Eles fornecem uma API fortemente tipada, para que você possa ter certeza de que, ao receber uma `StringCollection` como parâmetro ou valor de retorno, ela conterá apenas strings e você não precisará lançar ao buscar elementos da coleção. Parece ideal, mas há dois problemas. Primeiro, ele implementa `IList`, então você ainda pode tentar adicionar não-strings a ele (embora você falhe no tempo de execução). Em segundo lugar, trata apenas de strings. Existem outras coleções especializadas, mas no geral não cobrem muito terreno. Existe o tipo `CollectionBase`, que pode ser usado para construir suas próprias coleções fortemente tipadas, mas isso significa criar um novo tipo de coleção para cada tipo de elemento, o que também não é o ideal.

Agora que você viu o problema com coleções, vamos considerar o problema que pode ocorrer quando você substitui métodos e implementa interfaces. Está relacionado à ideia de covariância, que já vimos com arrays.

#### FALTA DE TIPOS DE RETORNO COVARIANTES

`ICloneable` é uma das interfaces mais simples do framework. Possui um único método, `Clone`, que deve retornar uma cópia do objeto que o método é chamado. Agora, deixando de lado a questão se esta deve ser uma cópia profunda ou superficial, vejamos a assinatura do método `Clone`:

```
objeto Clone()
```

É uma assinatura direta, certamente — mas como eu disse, o método deve retornar uma cópia do objeto que é chamado. Isso significa que ele precisa retornar um objeto do mesmo tipo, ou pelo menos compatível (onde esse significado irá variar dependendo do tipo).

Faria sentido poder substituir o método por uma assinatura que fornecesse uma descrição mais precisa do que o método realmente retorna. Por exemplo, em uma classe `Person` seria bom poder implementar `ICloneable` com

```
Clone de pessoa pública()
```

Isso não quebraria nada — o código que espera que qualquer objeto antigo ainda funcione bem. Esse recurso é chamado de *covariância de tipo de retorno*, mas, infelizmente, a implementação da interface e a substituição de métodos não o suportam. Em vez disso, a solução normal para interfaces é usar a *implementação explícita da interface* para obter o efeito desejado:

```
Clone de pessoa pública() {
```

```
[A implementação vai aqui]
```

```
} objeto ICloneable.Clone() {
```

← Implementa interface explicitamente

```
    retornar Clone();  
}
```

← Chama método sem interface

Qualquer código que chame `Clone()` em uma expressão com um tipo estático `Person` chamará o método `top`; se o tipo da expressão for `ICloneable`, ele chamará o método `bottom`.

Isso funciona, mas é muito feio. A imagem espelhada desta situação também ocorre com

parâmetros, onde se você tivesse uma interface ou método virtual com uma assinatura de, digamos, void Process(string x), pareceria lógico poder implementar ou substituir o método com uma assinatura menos exigente, como void Process( objeto x). Isso é chamado de *contravariância de tipo de parâmetro*; é tão incompatível quanto a covariância do tipo de retorno e você precisa usar a mesma solução alternativa para interfaces e sobrecarga normal para métodos virtuais. Não é um empecilho, mas é irritante.

É claro que os desenvolvedores C# 1 aguentaram todos esses problemas por muito tempo, e os desenvolvedores Java tiveram uma situação semelhante por muito mais tempo. Embora a segurança do tipo em tempo de compilação seja um ótimo recurso em geral, não me lembro de ter visto muitos bugs em que as pessoas colocassem o tipo errado de elemento em uma coleção. Posso conviver com a solução alternativa para a falta de covariância e contravariância. Mas existe elegância e fazer com que seu código expresse claramente o que você quer dizer, de preferência *sem* a necessidade de comentários explicativos. Mesmo que os bugs não ocorram, impor o contrato documentado de que uma coleção *deve* conter apenas strings (por exemplo) pode ser caro e frágil diante de coleções mutáveis. Este é o tipo de contrato que você realmente deseja que o próprio sistema de tipos aplique.

Você verá mais tarde que o C# 2 também não é perfeito, mas traz grandes melhorias. Há mais mudanças no C# 4, mas mesmo assim, faltam a covariância do tipo de retorno e a contravariância do parâmetro.<sup>11</sup>

### 2.2.3 Resumo das características do sistema de tipo

Nesta seção, você aprendeu algumas das diferenças entre sistemas de tipos e, em particular, quais características se aplicam ao C# 1:

- ÿ C# 1 é digitado estaticamente – o compilador sabe quais membros permitir que você use.
- ÿ C# 1 é explícito – você deve indicar o tipo de cada variável.
- ÿ C# 1 é seguro – você não pode tratar um tipo como se fosse outro, a menos que haja uma conversão genuína disponível.
- ÿ A tipagem estática não permite que uma única coleção seja uma lista de strings fortemente tipada ou uma lista de inteiros sem muita duplicação de código para diferentes tipos de elementos.
- ÿ A substituição de métodos e a implementação de interface não permitem covariância ou contravariância.

A próxima seção aborda um dos aspectos mais fundamentais do sistema de tipos do C#, além de suas características de alto nível: as diferenças entre estruturas e classes.

## 2.3

### Tipos de valor e tipos de referência Seria

difícil exagerar a importância do assunto desta seção. Tudo o que você faz no .NET lidará com um tipo de valor ou um tipo de referência e, ainda assim, é curiosamente possível desenvolver por um longo tempo com apenas uma vaga ideia de qual é a diferença. Pior ainda, existem muitos mitos para confundir ainda mais as coisas. O fato lamentável é que é fácil fazer uma declaração curta, mas incorreta, que esteja próxima o suficiente da verdade

<sup>11</sup> O C# 4 introduziu covariância e contravariância genéricas limitadas, mas isso não é exatamente a mesma coisa.

ser plausível, mas impreciso o suficiente para ser enganoso – mas é relativamente complicado chegar a uma descrição concisa, mas precisa.

Esta seção não é uma análise completa de como os tipos são tratados, empacotamento entre domínios de aplicativos, interoperabilidade com código nativo e assim por diante. Em vez disso, é uma breve olhada nos fundamentos absolutos do tópico (conforme aplicado ao C# 1) que são cruciais para lidar com versões posteriores do C#.

Começaremos vendo como as diferenças fundamentais entre tipos de valor e os tipos de referência aparecem naturalmente no mundo real, assim como no .NET.

### 2.3.1 Valores e referências no mundo real

Suponha que você esteja lendo algo fantástico e queira que um amigo leia também. Suponhamos ainda que se trata de um documento de domínio público, apenas para evitar quaisquer acusações de apoio à violação de direitos autorais. O que você precisa dar ao seu amigo para que ele também leia? Depende inteiramente do que você está lendo.

Primeiro, trataremos do caso em que você tem papel de verdade em mãos. Para dar uma cópia ao seu amigo, você precisa fotocopiar todas as páginas e depois entregá-la a ele. Nesse ponto, ele tem sua própria cópia completa do documento. Nesta situação, você está lidando com um comportamento *de tipo de valor*. Todas as informações estão diretamente em suas mãos – você não precisa ir a nenhum outro lugar para obtê-las. Sua cópia das informações também será independente da do seu amigo depois que você fizer a cópia. Você poderia adicionar algumas notas às suas páginas e as páginas dele não seriam alteradas.

Compare isso com a situação em que você está lendo uma página da web. Desta vez, tudo que você precisa dar ao seu amigo é o URL da página web. Este é um comportamento *de tipo de referência*, com o URL tomando o lugar da referência. Para ler o documento, você deve navegar pela referência colocando a URL no seu navegador e solicitando o carregamento da página. Se a página da web mudar por algum motivo (imagine que é uma página wiki e você adicionou suas notas à página), você e seu amigo verão essa mudança na próxima vez que cada um de vocês carregar a página.

Estas diferenças no mundo real ilustram o cerne da distinção entre tipos de valor e tipos de referência em C# e .NET. A maioria dos tipos no .NET são tipos de referência e é provável que você crie *muito* mais referências do que tipos de valor. Os casos mais comuns são classes (declaradas usando class), que são tipos de referência, e estruturas (declaradas usando struct), que são tipos de valor. As demais situações são as seguintes:

- ÿ Os tipos array são tipos de referência, mesmo que o tipo de elemento seja um tipo de valor (então int[] ainda é um tipo de referência, embora int seja um tipo de valor).
- ÿ Enumerações (declaradas usando enum) são tipos de valor. ÿ
- Os tipos delegados (declarados usando delegado) são tipos de referência. ÿ
- Tipos de interface (declarados usando interface) são tipos de referência, mas podem ser implementado por tipos de valor.

Agora que você tem uma ideia básica do que são os tipos de referência e os tipos de valor, veremos alguns dos detalhes mais importantes.

### 2.3.2 Fundamentos de valor e tipo de referência

O conceito-chave a ser entendido quando se trata de tipos de valor e tipos de referência é o que o valor de uma expressão específica é. Para manter as coisas concretas, usarei variáveis como exemplos mais comuns de expressões, mas o mesmo se aplica a propriedades, chamadas de método, indexadores e outras expressões.

Como discutimos na seção 2.2.1, a maioria das expressões possui um tipo estático associado a eles. O valor de uma expressão de tipo de valor é o valor puro e simples. Por exemplo, o valor da expressão "2 + 3" é 5. O valor de uma expressão de tipo de referência, entretanto, é uma referência – não é o objeto ao qual a referência se refere. O valor da expressão `String.Empty` não é uma string vazia – é uma referência a uma string vazia. Nas discussões diárias e mesmo na documentação, tendemos a confundir esta distinção. Para Por exemplo, eu poderia descrever `String.Concat` como retornando “uma string que é a concatenação de todos os parâmetros”. Usar uma terminologia precisa aqui seria demorado e perturbador, e não há problema, desde que todos os envolvidos entendam que apenas uma referência é retornada.

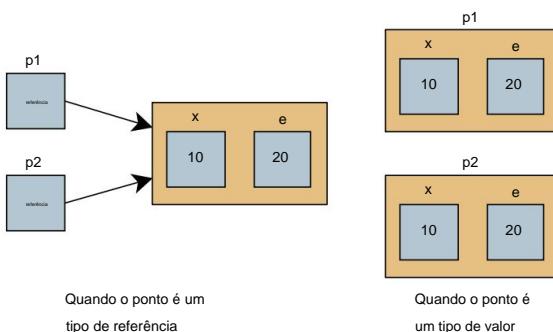
Para demonstrar isso ainda mais, considere um tipo Point que armazena dois inteiros, x e y. Poderia ter um construtor que assumisse os dois valores. Este tipo poderia ser implementado como uma estrutura ou uma classe. A Figura 2.3 mostra o resultado da execução das seguintes linhas do código:

```
Ponto p1 = novo Ponto(10, 20);
Ponto p2 = p1;
```

O lado esquerdo da figura 2.3 indica os valores envolvidos quando Point é uma classe (um tipo de referência), e o lado direito mostra a situação quando Point é uma struct (um tipo de valor).

Em ambos os casos, p1 e p2 possuem o mesmo valor após a atribuição. Mas no caso onde Point é um tipo de referência, esse valor é uma referência: tanto p1 quanto p2 referem-se ao mesmo objeto. Quando Point é um tipo de valor, o valor de p1 é o conjunto dos dados de um ponto - os valores x e y. Atribuir o valor de p1 a p2 copia todos esses dados.

Os valores das variáveis são armazenados onde quer que sejam declarados. Valores de variáveis locais são sempre armazenados na pilha,<sup>12</sup> e os valores das variáveis de instância são sempre armazenados



**Figura 2.3 Comparando comportamentos de tipo de valor e tipo de referência, particularmente no que diz respeito à atribuição**

<sup>12</sup> Isso só é totalmente verdade para C# 1. Você verá mais tarde que variáveis locais podem acabar no heap em determinadas situações em versões posteriores.

onde quer que a própria instância esteja armazenada. Instâncias de tipo de referência (objetos) são sempre armazenadas no heap, assim como variáveis estáticas.

Outra diferença entre os dois tipos de tipo é que os tipos de valor não podem ser derivados. Uma consequência disso é que o valor não precisa de nenhuma informação extra sobre o tipo que esse valor *realmente* é. Compare isso com os tipos de referência, onde cada objeto contém um bloco de dados no início identificando o tipo do objeto, juntamente com algumas outras informações. Você nunca pode alterar o tipo de um objeto – quando você executa uma conversão simples, o tempo de execução apenas pega uma referência, verifica se o objeto ao qual se refere é um objeto válido do tipo desejado e retorna a referência se for válida ou lança uma exceção caso contrário. A referência em si não conhece o tipo do objeto, portanto o mesmo valor de referência pode ser usado para múltiplas variáveis de tipos diferentes. Por exemplo, considere o seguinte código:

```
Fluxo de fluxo = new MemoryStream();
MemoryStream memoryStream = (MemoryStream) fluxo;
```

A primeira linha cria um novo objeto `MemoryStream` e define o valor da variável `stream` como uma referência a esse novo objeto. A segunda linha verifica se o valor de `stream` se refere a um objeto `MemoryStream` (ou tipo derivado) e define o valor de `memoryStream` como igual a `stream`.

Depois de compreender esses pontos básicos, você poderá aplicá-los ao pensar sobre algumas das falsidades que são frequentemente afirmadas sobre tipos de valor e tipos de referência.

### 2.3.3 Dissipando mitos

Vários mitos circulam regularmente. Tenho certeza de que a desinformação é quase sempre transmitida sem malícia e sem nenhuma ideia das imprecisões envolvidas, mas mesmo assim é inútil. Nesta seção, abordarei os mitos mais proeminentes, explicando a verdadeira situação à medida que avançamos.

#### MITO #1: ESTRUTURAS SÃO CLASSES LEVES

Esse mito vem em uma variedade de formas. Algumas pessoas acreditam que os tipos de valor não podem ou não devem ter métodos ou outro comportamento significativo — eles devem ser usados como tipos simples de transferência de dados, apenas com campos públicos ou propriedades simples. O tipo `DateTime` é um bom contraexemplo para isso: faz sentido que seja um tipo de valor, em termos de ser uma unidade fundamental como um número ou um caractere, e também faz sentido que seja capaz de realizar cálculos baseados em seu valor. Olhando as coisas de outra direção, os tipos de transferência de dados geralmente deveriam ser tipos de referência — a decisão deveria ser baseada no valor desejado ou na semântica do tipo de referência, não na simplicidade do tipo.

Outras pessoas acreditam que os tipos de valor são “mais leves” que os tipos de referência em termos de desempenho. A verdade é que, em *alguns* casos, os tipos de valor têm melhor desempenho — eles não exigem coleta de lixo, a menos que estejam em caixa, não têm sobrecarga de identificação de tipo e não exigem desreferenciação, por exemplo. Mas, de outras maneiras, os tipos de referência têm melhor desempenho - passagem de parâmetros, atribuição de valores a variáveis, retorno de valores e operações semelhantes requerem apenas 4 ou 8 bytes para serem

copiado (dependendo se você está executando o CLR de 32 ou 64 bits ) em vez de copiando *todos* os dados. Imagine se ArrayList fosse de alguma forma um tipo de valor "puro" e passar uma expressão ArrayList para um método envolvia copiar todos os seus dados! Em quase em todos os casos, o desempenho não é realmente determinado por esse tipo de decisão. Os gargalos quase nunca estão onde você imagina que estarão, e antes de tomar uma decisão de projeto com base no desempenho, você deve medir as diferentes opções.

É importante notar que a combinação das duas crenças também não funciona. Isto não importa quantos métodos um tipo possui (seja uma classe ou uma estrutura) - o a memória obtida por instância não é afetada. (Há um custo em termos de memória assumido para o código em si, mas isso ocorre uma vez e não para cada instância.)

#### **MITO #2: TIPOS DE REFERÊNCIA AO VIVO NO HEAP; TIPOS DE VALOR AO VIVO NA PILHA**

Este é muitas vezes causado pela preguiça por parte da pessoa que o repete. O primeiro parte está correta — uma instância de um tipo de referência é sempre criada no heap. É o segunda parte que causa problemas. Como já observei, o valor de uma variável reside onde quer que ela seja declarada, portanto, se você tiver uma classe com uma variável de instância do tipo int, o valor dessa variável para qualquer objeto estará sempre onde o resto dos dados para. o objeto está – na pilha. Somente variáveis locais (variáveis declaradas dentro de métodos) e métodos parâmetros residem na pilha. No C# 2 e posterior, mesmo algumas variáveis locais não vivem na pilha, como você verá quando examinarmos os métodos anônimos no capítulo 5.

**ESTES CONCEITOS SÃO RELEVANTES AGORA?** É discutível que, se você estiver escrevendo código gerenciado, deixe o tempo de execução se preocupar com a melhor forma de usar a memória. Na verdade, a especificação da linguagem não oferece garantias sobre o que vive onde; um tempo de execução futuro poderá criar alguns objetos na pilha se sabe que pode escapar impune, ou o compilador C# pode gerar código que quase não usa a pilha.

O próximo mito geralmente é apenas uma questão de terminologia.

#### **MITO #3: OBJETOS SÃO PASSADOS POR REFERÊNCIA EM C# POR PADRÃO**

Este é provavelmente o mito mais amplamente propagado. Novamente, as pessoas que fazem isso afirmam que muitas vezes (embora nem sempre) sabem como o C# realmente se comporta, mas não sabem o que "passar por referência" realmente significa. Infelizmente, isso é confuso para as pessoas que sei o que isso significa.

A definição formal de *passagem por referência* é relativamente complicada, envolvendo *valores /* e terminologia semelhante da ciência da computação, mas o importante é que, se você passar em um variável por referência, o método que você está chamando pode alterar o *valor da variável do chamador* alterando seu valor de parâmetro. Agora, lembre-se que o valor de uma referência variável de tipo é a *referência*, não o objeto em si. Você pode alterar o *conteúdo* do objeto ao qual um parâmetro se refere sem que o próprio parâmetro seja passado por referência. Por exemplo, o método a seguir altera o conteúdo do StringBuilder objeto em questão, mas a expressão do chamador ainda se referirá ao mesmo objeto que antes:

```
void AppendHello (construtor StringBuilder) {  
  
    construtor.Append("olá");  
}
```

Quando este método é chamado, o valor do parâmetro (uma referência a um `StringBuilder`) é passado por valor. Se você alterasse o valor da variável construtora dentro do método — por exemplo, com a instrução `construtor = null;` — essa alteração não seria vista pelo chamador, ao contrário do mito.

É interessante notar que não apenas a parte “por referência” do mito é imprecisa, mas também a parte “objetos são passados”. Os próprios objetos *nunca* são passados, seja por referência ou por valor. Quando um tipo de referência está envolvido, a variável é passada por referência ou o valor do argumento (a referência) é passado por valor.

Além de qualquer outra coisa, isso responde à questão do que acontece quando `null` é usado como argumento por valor — se objetos estivessem sendo passados, isso causaria problemas, pois não haveria um objeto para passar! Em vez disso, a referência nula é passada por valor da mesma forma que qualquer outra referência seria.

Se esta rápida explicação o deixou perplexo, você pode dar uma olhada em meu artigo, “Passagem de parâmetros em C#” (<http://mng.bz/otVt>), que fornece muito mais detalhes.

Esses mitos não são os únicos que existem. Boxing e unboxing entram em cena bastante mal-entendido, que tentarei esclarecer a seguir.

#### 2.3.4 Encaixotamento e desembalagem

Às vezes, você simplesmente não quer um valor de tipo de valor. Você quer uma referência. Existem vários motivos pelos quais isso pode acontecer e, felizmente, C# e .NET fornecem um mecanismo chamado *boxing* que permite criar um objeto a partir de um tipo de valor `value` e usar uma referência para esse novo objeto. Antes de passarmos a um exemplo, vamos começar revisando dois fatos importantes:

- ÿ O valor de uma variável de tipo de referência é sempre uma referência.
- ÿ O valor de uma variável de tipo de valor é sempre um valor desse tipo.

Dados esses dois fatos, as três linhas de código a seguir não parecem fazer muito sentido à primeira vista:

```
int eu = 5; objeto  
o = eu; intj = (int)o;
```

Você tem duas variáveis: `i` é uma variável do tipo valor e `o` é uma variável do tipo referência.

Como faz sentido atribuir o valor de `i` a `o`? O valor de `o` tem que ser uma referência, e o número 5 não é uma referência — é um valor inteiro. O que realmente está acontecendo é o boxe: o tempo de execução cria um objeto (no heap — é um objeto normal) que contém o valor (5). O valor de `o` é então uma referência a esse novo objeto. O valor no objeto é uma cópia do valor original — alterar o valor de `i` não alterará em nada o valor na caixa.

A terceira linha executa a operação inversa – unboxing. Você precisa informar ao compilador qual tipo desempacotar o objeto e, se usar o tipo errado (se for um uint ou longo em caixa, por exemplo, ou se não for um valor em caixa), uma InvalidCastException será lançada. Novamente, unboxing copia o valor que estava na caixa; após a atribuição, não há mais associação entre j e o objeto.

Em poucas palavras, isso é boxe e unboxing. O único problema restante é saber quando ocorre o boxe e o unboxing. O unboxing geralmente é óbvio, porque o elenco está presente no código. O boxe pode ser mais sutil. Você viu a versão simples, mas ela também pode ocorrer se você chamar os métodos ToString, Equals ou GetHashCode no valor de um tipo que não os substitui,<sup>13</sup> ou se você usar o valor como uma expressão de interface – atribuindo para uma variável cujo tipo é um tipo de interface ou passando-o como o valor de um parâmetro com um tipo de interface. Por exemplo, a instrução IComparable x = 5; colocaria o número 5 na caixa.

Vale a pena estar ciente do boxe e do unboxing por causa da potencial penalidade de desempenho envolvida. Uma única operação de box ou unbox é barata, mas se você executar centenas de milhares delas, você não apenas terá o custo das operações, mas também estará criando muitos *objetos*, o que dá mais trabalho ao coletor de lixo.

Esse impacto no desempenho geralmente não é um problema, mas vale a pena estar ciente para que você possa medir o efeito se estiver preocupado.

### 2.3.5 Resumo dos tipos de valor e tipos de referência

Nesta seção, examinamos as diferenças entre tipos de valor e tipos de referência e alguns dos mitos que os cercam. Aqui estão os pontos-chave:

- ÿ O valor de uma expressão de tipo de referência (uma variável, por exemplo) é uma referência essêncial, não um objeto.
- ÿ As referências são como URLs – são pequenos pedaços de dados que permitem acessar informações reais.
- ÿ O valor de uma expressão de tipo de valor são os dados reais. ÿ Há momentos em que os tipos de valor são mais eficientes que os tipos de referência, e vice-versa.
- ÿ Os objetos do tipo referência estão sempre no heap, mas os valores do tipo valor podem estar na pilha ou no heap, dependendo do contexto. ÿ Quando um tipo de referência é usado como parâmetro de método, por padrão o argumento é passado *por valor*, mas o valor em si é uma referência. ÿ Os valores do tipo de valor são colocados em caixa quando o comportamento do tipo de referência é necessário; desembalar é o processo inverso.

Agora que demos uma olhada em todas as partes do C# 1 com as quais você precisa se sentir confortável, é hora de dar uma rápida olhada no futuro e ver onde cada um dos recursos foi aprimorado pelas versões posteriores do C#.

---

<sup>13</sup> O boxe *sempre* ocorrerá quando você chamar GetType() em uma variável de tipo de valor, porque ela não pode ser substituída. Você já deve saber o tipo exato se estiver lidando com o formulário unboxed, então você pode simplesmente usar typeof .

## 2.4 Além do C# 1: novos recursos em uma base sólida

Os três tópicos abordados neste capítulo são vitais para todas as versões do C#. Quase todos os novos recursos estão relacionados a pelo menos um deles e alteram o equilíbrio de como a linguagem é usada. Antes de encerrarmos o capítulo, vamos explorar como os novos recursos se relacionam com os antigos. Não darei muitos detalhes (por algum motivo o editor não queria uma seção de 600 páginas), mas é útil ter uma ideia de para onde estamos indo antes de chegarmos ao âmago da questão. Iremos examiná-los na mesma ordem em que os abordamos anteriormente, começando pelos delegados.

### 2.4.1 Recursos relacionados aos delegados

Delegados de todos os tipos recebem um impulso no C# 2 e, em seguida, recebem tratamento ainda mais especial no C# 3. A maioria dos recursos não são novos no CLR, mas são truques inteligentes do compilador para fazer com que os delegados funcionem com mais facilidade dentro da linguagem. As alterações afetam não apenas a sintaxe que você pode usar, mas também a aparência e a sensação do código C# idiomático. Com o tempo, o C# está ganhando uma abordagem mais funcional.

C# 1 tem uma sintaxe bastante desajeitada quando se trata de criar uma instância delegada. Por um lado, mesmo que você precise realizar algo simples, você terá que escrever um método totalmente separado para criar uma instância delegada para ele. O C# 2 corrigiu isso com métodos anônimos e introduziu uma sintaxe mais simples para os casos em que você ainda deseja usar um método normal para fornecer a ação para o delegado. Você também pode criar instâncias delegadas usando métodos com assinaturas compatíveis — a assinatura do método não precisa mais ser exatamente igual à declaração do delegado.

A listagem a seguir demonstra todas essas melhorias.

#### Listagem 2.4 Melhorias na instanciação de delegação trazidas pelo C# 2

```
static void HandleDemoEvent(objeto remetente, EventArgs e) {
    Console.WriteLine ("Tratado por HandleDemoEvent");
}
...
Manipulador EventHandler;
manipulador = new EventHandler(HandleDemoEvent); manipulador
(nulo, EventArgs.Empty);

manipulador = HandleDemoEvent;
manipulador (nulo, EventArgs.Empty);

manipulador = delegado(remetente do objeto, EventArgs e) {
    Console.WriteLine ("Tratado anonimamente");
};
manipulador (nulo, EventArgs.Empty);

manipulador = delegado {
    Console.WriteLine ("Tratado anonimamente novamente");
};
manipulador (nulo, EventArgs.Empty);
```

- B Especifica o tipo e método do delegado
- C Converte implicitamente em instância delegada
- D Especifica ação com método anônimo
- E Usa atalho de método anônimo

```
MouseEventHandler mouseHandler = HandleDemoEvent;
mouseHandler(nulo, novo MouseEventArgs(MouseButtons.None,
                                         0, 0, 0));
```



A primeira parte do código principal **B** é apenas código C# 1, mantido para comparação. O todos os delegados restantes usam novos recursos do C# 2. Conversões de grupos de métodos **C** fazem o código de assinatura do evento é lido de maneira muito mais agradável - linhas como `saveButton.Click += SalvarDocumento;` são simples, sem penugem extra para distrair os olhos. O sintaxe do método anônimo **D** é um pouco complicada, mas permite que a ação ser claro no ponto de criação, em vez de ser outro método a ser observado antes você entende o que está acontecendo. Um atalho está disponível ao usar métodos anônimos **E**, mas este formulário só pode ser usado quando você não precisa dos parâmetros. Os métodos anônimos também possuem outros recursos poderosos, mas veremos isso mais tarde.

A instância delegada final criada **F** é uma instância de `MouseEventHandler` em vez do que apenas `EventHandler`, mas o método `HandleDemoEvent` ainda pode ser usado devido à *contravariância*, que especifica a compatibilidade dos parâmetros. *Covariância* especifica o tipo de retorno compatibilidade. Veremos ambos com mais detalhes no capítulo 5. Manipuladores de eventos são provavelmente os maiores beneficiários disso, porque de repente a diretriz da Microsoft para fazer com que todos os tipos de delegados usados em eventos sigam a mesma convenção faz muito mais sentido. No C# 1, não importava se dois manipuladores de eventos diferentes pareciam bastante semelhante - você precisava ter um método com uma assinatura *exatamente* correspondente para para criar uma instância delegada. Em C # 2, você pode usar o mesmo método para lidar com muitos tipos diferentes de eventos, especialmente se o propósito do O método é bastante independente de eventos, como o registro.

C# 3 fornece sintaxe especial para instanciar tipos delegados, usando *expressões lambda*. Para demonstrar isso, usaremos um novo tipo de delegado. Quando o CLR ganhou genéricos no .NET 2.0, tipos delegados genéricos tornaram-se disponíveis e foram usados em diversas chamadas de API em coleções genéricas. O .NET 3.5 dá um passo adiante, apresentando um grupo de tipos de delegados genéricos chamados `Func`, todos com parâmetros especificados tipos e retornar um valor de outro tipo especificado. A listagem a seguir mostra o uso de um tipo delegado `Func`, bem como expressões lambda.

#### Listagem 2.5 Expressões Lambda – como métodos anônimos aprimorados

```
Func<int,int,string> func = (x, y) => (x * y).ToString();
Console.WriteLine(func(5, 20));
```

`Func<int,int,string>` é um tipo delegado que recebe dois números inteiros e retorna uma string. A expressão lambda na listagem 2.5 especifica que a instância delegada (mantida em `func`) deve multiplicar os dois inteiros e chamar `ToString()`. A sintaxe é muito mais simples do que os métodos anônimos, e existem outros benefícios em termos da quantidade de inferência de tipo que o compilador está preparado para realizar para você. As expressões lambda são absolutamente cruciais para o LINQ e você deve se preparar para torná-los uma parte essencial do seu kit de ferramentas linguísticas. Eles não estão restritos a trabalhar

com LINQ, porém - qualquer uso de métodos anônimos do C# 2 pode usar lambda expressões em C# 3, e isso quase sempre levará a um código mais curto.

Para resumir, os novos recursos relacionados aos delegados são os seguintes:

- Genéricos (tipos delegados genéricos) —
- C# 2 — Delegar expressões de criação de instância
- C# 2 — Métodos anônimos —
- C# 2 — Delegar covariância/contravariância — C# 2
- Expressão lambda — C# 3

Além disso, o C# 4 permite covariância e contravariância genéricas para delegados, o que vai além do que você acabou de ver. Na verdade, os genéricos constituem um dos principais melhorias no sistema de tipos, que veremos a seguir.

#### 2.4.2 Recursos relacionados ao sistema de tipos

O principal novo recurso do C# 2 em relação ao sistema de tipos é a inclusão de genéricos. Ele aborda amplamente as questões que levantei na seção 2.2.2 sobre coleções fortemente tipadas, embora os tipos genéricos também sejam úteis em diversas outras situações. Como um característica, é elegante, resolve um problema real e, apesar de algumas rugas, geralmente funciona bem. Você já viu exemplos disso em alguns lugares, e é descrito completamente no próximo capítulo, portanto não entrarei em mais detalhes aqui. Genéricos formam provavelmente o recurso mais importante em C# 2 com relação ao sistema de tipos, e você verá tipos genéricos ao longo do restante do livro.

C# 2 não aborda os problemas de covariância de tipo de retorno e contravariância de parâmetro para substituir membros ou implementar interfaces. Mas melhora a situação para a criação de instâncias delegadas em determinadas situações, como você viu na seção 2.4.1.

O C# 3 introduziu uma série de novos conceitos no sistema de tipos, mais notavelmente tipos anônimos, variáveis locais digitadas implicitamente e métodos de extensão. Os próprios tipos anônimos estão presentes principalmente por causa do LINQ, onde é útil poder efetivamente crie um tipo de transferência de dados com um monte de propriedades somente leitura sem precisar realmente escrever o código para elas. Não há nada que os impeça de serem usados fora LINQ, porém, o que facilita a vida das demonstrações. A Listagem 2.6 mostra ambos os recursos em ação.

#### Listagem 2.6 Demonstração de tipos anônimos e digitação implícita

```
var jon = novo { Nome = "Jon", Idade = 31 };
var tom = new { Nome = "Tom", Idade = 4 };
Console.WriteLine ("{0} é {1}", jon.Name, jon.Age);
Console.WriteLine ("{0} é {1}", tom.Nome, tom.Idade);
```

As duas primeiras linhas mostram digitação implícita (o uso de var) e objeto anônimo inicializadores (o novo bit {...}), que criam instâncias de tipos anônimos.

Há duas coisas dignas de nota nesta fase, muito antes de entrarmos no detalhes – pontos que já fizeram com que as pessoas se preocupassem desnecessariamente antes. A primeira é que C# 3 ainda é digitado estaticamente. O compilador C# declarou que Jon e Tom são de um determinado

tipo, normalmente, e quando você usa as propriedades dos objetos, elas são propriedades normais – nenhuma pesquisa dinâmica está acontecendo. Acontece que você (como autor do código-fonte) não pode dizer ao compilador que tipo usar na declaração da variável porque o compilador estará gerando o próprio tipo. As propriedades também são digitadas estaticamente – aqui a propriedade Age é do tipo int e a propriedade Name é do tipo string.

O segundo ponto é que não criamos dois tipos anônimos diferentes aqui. As variáveis jon e tom têm o mesmo tipo porque o compilador usa os nomes das propriedades, os tipos e a ordem para descobrir que pode gerar apenas um tipo e usá-lo para ambas as instruções. Isso é feito por assembly e torna a vida muito mais simples em termos de poder atribuir o valor de uma variável a outra (por exemplo, jon = tom; seria permitido no código anterior) e operações semelhantes.

Os métodos de extensão também existem para o LINQ, mas podem ser úteis fora dele.

Pense em todas as vezes que você desejou que um tipo de framework tivesse um determinado método e teve que escrever um método utilitário estático para implementá-lo. Por exemplo, para criar uma nova string revertendo uma existente, você pode escrever um método estático StringUtil.Reverse. Bem, o recurso de método de extensão permite efetivamente chamar esse método estático como se ele existisse no próprio tipo de string, para que você pudesse escrever

```
string x = "dirow olleH".Reverse();
```

Os métodos de extensão também permitem que você adicione métodos com implementações às interfaces, e o LINQ depende muito disso, permitindo chamadas para todos os tipos de métodos em I Enumerable<T> que nunca existiram anteriormente.

C# 4 possui dois recursos relacionados ao sistema de tipos. Um recurso relativamente menor é a covariância e a contravariância para delegados e interfaces genéricos. Isso está presente no CLR desde o lançamento do .NET 2.0, mas somente com a introdução do C# 4 e atualizações nos tipos genéricos na *Base Class Library* (BCL) ele se tornou utilizável para desenvolvedores C#. Um recurso muito maior – embora muitos programadores talvez nunca precisem – é a digitação dinâmica em C#.

Lembra da introdução que dei à digitação estática, onde tentei usar a propriedade Length de um array e uma string através da mesma variável? Bem, em C# 4 funciona – quando você quiser. A listagem a seguir mostra o mesmo código, exceto pela declaração da variável, mas funcionando como código C# 4 válido.

#### Listagem 2.7 Tipagem dinâmica em C# 4

```
dinâmico o = "olá";
Console.WriteLine(o.Comprimento); o =
nova string[] {"oi", "lá"};
Console.WriteLine(o.Comprimento);
```

Ao declarar a variável o como tendo um tipo estático de dinâmico (sim, você leu certo), o compilador lida com quase tudo relacionado a o de maneira diferente, deixando todas as decisões de ligação (como o que Length significa) até o tempo de execução.

Obviamente, examinaremos a digitação dinâmica com mais profundidade, mas quero enfatizar agora que o C# 4 ainda é uma linguagem de tipagem estática em sua maior parte. A menos que você esteja

usando o tipo dinâmico (que atua como um tipo estático denotando um valor dinâmico), tudo funciona exatamente da mesma maneira que antes. A maioria dos desenvolvedores C# raramente precisará de digitação dinâmica e, no resto do tempo, eles podem ignorá-la. Quando a digitação dinâmica é útil, ela pode ser muito inteligente e permite que você jogue bem com código escrito em linguagens dinâmicas executadas no *Dynamic Language Runtime* (DLR). Eu apenas aconselho você a não começar a usar C# como uma linguagem principalmente dinâmica. Se é isso que você deseja, use Iron-Python ou algo semelhante; linguagens projetadas para oferecer suporte à digitação dinâmica desde o início provavelmente terão menos pegadinhas inesperadas.

Aqui está a lista de visualização rápida desses recursos, juntamente com a versão do C# em que eles foram introduzidos:

ÿ Genéricos—C# 2

ÿ Covariância/contravariância de delegação limitada — C# 2 ÿ

Tipos anônimos — C# 3 ÿ

Tipagem implícita — C# 3 ÿ

Métodos de extensão — C# 3

ÿ Covariância/contravariância genérica limitada — C# 4 ÿ

Tipagem dinâmica — C# 4

Depois desse conjunto bastante diversificado de recursos no sistema de tipos, vamos dar uma olhada nos recursos adicionados a uma parte específica da digitação no .NET: tipos de valor.

### 2.4.3 Recursos relacionados aos tipos de valor

Há apenas dois recursos para falar aqui, ambos introduzidos no C# 2. O primeiro remonta novamente aos genéricos e, em particular, às coleções. Uma reclamação comum sobre o uso de tipos de valor em coleções com .NET 1.1 era que, devido a todas as APIs de uso geral serem especificadas em termos de tipo de objeto, cada operação que adicionasse um valor struct a uma coleção envolveria encaixotá-la, e você teria que desempacotá-lo ao recuperá-lo. Embora o boxe seja bastante barato para uma chamada individual, ele pode causar um impacto significativo no desempenho se for usado sempre com coleções acessadas com frequência. Também ocupa mais memória do que o necessário, devido à sobrecarga por objeto. Os genéricos corrigem as deficiências de velocidade e memória usando o tipo *real*/envolvido em vez de um objeto de uso geral. Por exemplo, seria uma loucura ler um arquivo e armazenar cada byte como um elemento em um ArrayList no .NET 1.1, mas no .NET 2.0 não seria uma loucura fazer o mesmo com um List<byte>.

O segundo recurso aborda outra causa comum de reclamação, especialmente quando se trata de bancos de dados: o fato de que você não pode atribuir nulo a uma variável de tipo de valor.

Não existe um conceito de valor int nulo, por exemplo, mesmo que um campo inteiro *do banco de dados* possa ser anulável. Isso torna difícil modelar a tabela do banco de dados dentro de uma classe digitada estaticamente sem feiúra de uma forma ou de outra. Os tipos anuláveis fazem parte do .NET 2.0 e o C# 2 inclui sintaxe extra para torná-los fáceis de usar. A listagem a seguir dá um breve exemplo disso.

**Listagem 2.8 Demonstração de uma variedade de recursos de tipo anulável**

```

interno? x = nulo; x =
5; if (x! =
nulo) {

    int y = x.Valor;
    Console.WriteLine(y);

} int z = x ?? 10;

```

← Declara, define variável anulável

← Testes para presença de valor real

← Obtém valor real

← Usa operador de coalescência nula

A Listagem 2.8 mostra vários recursos dos tipos anuláveis e a abreviação que o C# fornece para trabalhar com eles. Veremos os detalhes de cada recurso no capítulo 4, mas o ponto importante aqui é como tudo isso é mais fácil e limpo do que qualquer solução alternativa usada no passado.

A lista de melhorias é menor desta vez, mas são recursos importantes no termos de desempenho e elegância de expressão:

- ÿ Genéricos—C# 2
- ÿ Tipos anuláveis — C# 2

## 2.5 Resumo

Este capítulo foi principalmente um exercício de revisão para C# 1. O objetivo não era cobrir nenhum tópico em sua totalidade, mas apenas colocar todos na mesma página para que eu pudesse descrever os recursos posteriores sem me preocupar com o terreno que eu estou construindo.

Todos os tópicos que abordamos são essenciais para C# e .NET, mas tenho visto muitos mal-entendidos em torno deles nas discussões da comunidade. Embora este capítulo não tenha se aprofundado muito em nenhum ponto específico, esperamos que ele tenha esclarecido qualquer confusão que tornaria o resto do livro mais difícil de entender.

Os três tópicos principais que abordamos brevemente neste capítulo foram significativamente aprimorados desde o C# 1, e alguns recursos abordam mais de um tópico. Em particular, a adição de genéricos tem impacto em quase todas as áreas que abordamos neste capítulo — é provavelmente o recurso mais utilizado e importante no C# 2. Agora que terminamos todos os nossos preparativos, podemos começar a olhar para genéricos corretamente no próximo capítulo.

## Parte 2

# C# 2: Resolvendo os problemas do C# 1

**Em** Na parte 1, demos uma rápida olhada em alguns dos recursos do C# 2. Agora é hora de fazer o trabalho corretamente. Você verá como o C# 2 corrige vários problemas que os desenvolvedores encontraram ao usar o C# 1 e como o C# 2 torna os recursos existentes mais úteis, simplificando-os. Isso não é tarefa fácil, e a vida com C# 2 é muito mais agradável do que com C# 1.

Os novos recursos do C# 2 têm uma certa independência. Isso não quer dizer que eles não tenham nenhum parentesco; muitos dos recursos são baseados - ou pelo menos interagem - na enorme contribuição que os genéricos dão à linguagem. Mas os diferentes tópicos que veremos nos próximos cinco capítulos não se combinam em um único super recurso.

Os primeiros quatro capítulos desta parte cobrem os maiores novos recursos. Veremos o seguinte:

- ÿ **Genéricos** — O novo recurso mais importante do C# 2 (e, de fato, do CLR para .NET 2.0), os genéricos permitem a parametrização de tipos e métodos em termos dos tipos com os quais interagem.
- ÿ **Tipos anuláveis** — Tipos de valor como int e DateTime não possuem nenhum conceito de “nenhum valor presente”; tipos anuláveis permitem representar a ausência de um valor significativo.
- ÿ **Delegados** — Embora os delegados não tenham mudado no nível CLR , o C# 2 torna muito mais fácil trabalhar com eles. Além de alguns atalhos simples, a introdução de métodos anônimos dá início ao movimento em direção a um estilo de programação mais funcional — uma tendência que continua no C# 3.

Iteradores — Embora o uso de iteradores sempre tenha sido simples em C# com a instrução `foreach` , é difícil implementá-los em C# 1. O compilador C# 2 tem o prazer de construir uma máquina de estado para você nos bastidores, escondendo grande parte da complexidade envolvida.

Depois de cobrirmos os principais e complexos novos recursos do C# 2 com um capítulo dedicado a cada um deles, o capítulo 7 completa a cobertura introduzindo vários recursos mais simples. Mais simples não significa necessariamente menos útil; tipos parciais, em particular, são cruciais para melhor suporte ao designer nas versões do Visual Studio de 2005 em diante. O mesmo recurso também é benéfico para outros códigos gerados. Da mesma forma, muitos desenvolvedores de C# consideram garantida a capacidade de escrever uma propriedade com um getter público e um setter privado atualmente, mas isso só foi introduzido no C# 2.

Quando a primeira edição deste livro foi publicada, muitos desenvolvedores ainda não usavam o C# 2. Minha impressão em 2013 é que é raro encontrar alguém que esteja usando C# atualmente, mas que não tenha pelo menos se envolvido com C# 2, provavelmente 3, e muitas vezes 4. Os tópicos abordados aqui são fundamentais para como as versões posteriores do C# funcionam; em particular, tentar aprender sobre LINQ sem entender genéricos e iteradores seria complicado. O capítulo sobre iteradores também está relacionado aos métodos assíncronos do C# 5; os dois recursos são aparentemente muito diferentes, mas ambos envolvem máquinas de estado construídas pelo compilador para alterar o fluxo convencional de execução.

Se você já usa C# 2 e versões posteriores há algum tempo, poderá descobrir que grande parte desta parte cobre terreno familiar, mas suspeito que você ainda se beneficiará de um conhecimento mais profundo dos detalhes apresentados.

# 3

## Tipagem parametrizada com genéricos

Este capítulo cobre

- ÿ Inferência de tipos para métodos genéricos
- ÿ Restrições de tipo
- ÿ Reflexão e genéricos
- ÿ Comportamento CLR
- ÿ Limitações dos genéricos
- ÿ Comparações com outras línguas

História verídica:<sup>1</sup> Outro dia, minha esposa e eu saímos para fazer nossa visita semanal às compras de supermercado. Pouco antes de sairmos, ela me perguntou se eu tinha a lista. Confirmei que tinha a lista e lá fomos nós. Foi só quando chegamos ao supermercado que nosso erro se tornou óbvio. Minha esposa estava perguntando sobre a lista de compras, enquanto eu trouxe a lista de recursos interessantes do C# 2. Quando perguntamos a um assistente se poderíamos comprar algum método anônimo, recebemos um olhar estranho.

<sup>1</sup> Com isso quero dizer “conveniente para fins de introdução do capítulo” – não necessariamente exato.

Se ao menos pudéssemos nos expressar com mais clareza! Se ao menos ela tivesse alguma forma de dizer que queria que eu trouxesse a lista de itens que queríamos comprar! Se ao menos tivéssemos genéricos...

Para a maioria dos desenvolvedores, os genéricos são o novo recurso mais importante do C# 2. Eles melhoraram o desempenho, tornam seu código mais expressivo e transferem muitas verificações de segurança do tempo de execução para o tempo de compilação. Essencialmente, eles permitem *parametrizar* tipos e métodos. Assim como as chamadas normais de métodos geralmente possuem parâmetros para informar quais *valores* usar, os tipos e métodos genéricos possuem parâmetros de tipo para informar quais *tipos* usar. No início, tudo parece confuso - e se você é completamente novo no uso de genéricos, pode esperar uma certa confusão - mas depois de ter a ideia básica, você passará a amá-los.

Neste capítulo, veremos como usar tipos e métodos genéricos fornecidos por outros (seja na estrutura ou como bibliotecas de terceiros) e como escrever os seus próprios. Ao longo do caminho, veremos como os genéricos funcionam com as chamadas de reflexão na API e alguns detalhes sobre como o CLR lida com os genéricos. Para concluir o capítulo, apresentarei algumas das limitações encontradas com mais frequência nos genéricos, juntamente com possíveis soluções alternativas, e compararei os genéricos em C# com recursos semelhantes em outras linguagens.

Porém, primeiro você precisa entender os problemas que levaram à criação dos genéricos.

### 3.1 Por que os genéricos são necessários

Se você ainda tiver algum código C# 1 disponível, observe-o e conte as conversões, principalmente em códigos que usam coleções extensivamente. Não esqueça que quase todo uso de foreach contém uma conversão implícita. Quando você usa tipos projetados para trabalhar com muitos tipos diferentes de dados, isso naturalmente leva à conversão, dizendo silenciosamente ao compilador para não se preocupar, que está tudo bem; apenas trate a expressão ali como se ela tivesse esse tipo específico. Usar quase qualquer API que tenha object como tipo de parâmetro ou tipo de retorno provavelmente envolverá conversões em algum momento. Ter uma hierarquia de classe única com objeto como raiz torna algumas coisas mais diretas, mas o tipo de objeto em si é extremamente enfadonho e, para fazer algo genuinamente útil com um objeto, você quase sempre precisa lançá-lo.

Os elencos são ruins, ok? Nada mal de uma forma *quase nunca feita* desse tipo (como estruturas mutáveis e campos não privados), mas ruim de uma forma *necessária e maléfica*. Eles são uma indicação de que você deve fornecer mais informações ao compilador de alguma forma e que você está optando por pedir ao compilador que confie em você em tempo de compilação e gerar uma verificação que será executada em tempo de execução para mantê-lo honesto.

Se você precisar informar as informações ao compilador de alguma forma, é provável que qualquer pessoa que esteja *lendo* seu código também precise das mesmas informações. Eles podem ver onde você está lançando, é claro, mas isso não é muito útil. O local ideal para guardar essas informações geralmente é no ponto em que você declara uma variável ou método. Isso é ainda mais importante se você fornecer um tipo ou método que outras pessoas chamarão

sem acesso ao seu código. Os genéricos permitem que os provedores de bibliotecas evitem que seus usuários compilem código que chame a biblioteca com argumentos incorretos.

No C# 1, você tinha que confiar em documentação escrita manualmente, que pode facilmente se tornar incompleta ou imprecisa, como muitas vezes acontece com informações duplicadas. Quando as informações extras podem ser declaradas no código como parte de uma declaração de método ou tipo, todos podem trabalhar de forma mais produtiva. O compilador pode fazer mais verificações; o IDE pode apresentar opções do IntelliSense com base nas informações extras (por exemplo, oferecendo os membros da string como a próxima etapa quando você acessa um elemento dentro de uma lista de strings); quem chama métodos pode ter mais certeza de que os argumentos passados e os valores retornados estão corretos; e qualquer pessoa que mantenha seu código pode entender melhor o que estava passando pela sua cabeça quando você o escreveu originalmente.

**Os genéricos reduzirão sua contagem de bugs?** Cada descrição de genéricos que li (incluindo a minha) enfatiza a importância da verificação de tipo em tempo de compilação em vez da verificação de tipo em tempo de execução. Vou lhe contar um segredo: não me lembro de alguma vez ter corrigido um bug no código lançado que fosse diretamente devido à falta de verificação de tipo. Em outras palavras, as conversões que colocamos no código C# 1 sempre funcionaram, na minha experiência. Essas conversões eram como sinais de alerta, forçando-nos a pensar explicitamente na segurança do tipo, em vez de fluir naturalmente no código que escrevemos. Mas embora os genéricos possam não reduzir radicalmente o número de bugs de segurança de tipo que você encontra, a maior legibilidade que eles oferecem pode reduzir o número de bugs em geral. Código simples de entender é simples de acertar. Da mesma forma, o código que precisa ser robusto diante de chamadores maliciosos é muito mais simples de ser escrito corretamente quando o sistema de tipos pode fornecer garantias apropriadas.

Tudo isso seria suficiente para fazer os genéricos valerem a pena, mas também há melhorias de desempenho. Primeiro, como o compilador pode executar mais imposições, isso deixa menos a ser verificado em tempo de execução. Em segundo lugar, o JIT pode tratar os tipos de valor de uma forma particularmente inteligente, conseguindo eliminar boxing e unboxing em muitas situações. Em alguns casos, isso pode fazer uma enorme diferença no desempenho em termos de velocidade e consumo de memória.

Muitos dos benefícios dos genéricos podem parecer semelhantes aos benefícios das linguagens de tipo estaticamente em relação às dinâmicas: melhor verificação em tempo de compilação, mais informações expressas diretamente no código, mais suporte IDE, melhor desempenho .

A razão para isso é bastante simples: quando você usa uma API geral (como Array-List) que não consegue diferenciar entre os diferentes tipos, você efetivamente está em uma situação dinâmica em termos de acesso a essa API. A propósito, o inverso geralmente não é verdade: os benefícios que as linguagens dinâmicas oferecem raramente se aplicam à escolha entre APIs genéricas e não genéricas. Quando você pode usar genéricos de maneira razoável, a decisão de fazê-lo geralmente é óbvia.

Então, essas são as vantagens que esperam por você no C# 2 – agora é hora de começar a usar genéricos.

## 3.2 Genéricos simples para uso diário

O tema dos genéricos tem muitos cantos obscuros se você quiser saber *tudo* sobre ele.

A especificação da linguagem C# é detalhada para garantir que o comportamento seja especificado em praticamente todos os casos concebíveis. Mas você não precisa entender a maioria desses casos para ser produtivo. (Na verdade, o mesmo acontece em outras áreas. Por exemplo, você não precisa conhecer todas as regras exatas sobre atribuição definida — basta corrigir o código adequadamente quando o compilador reclamar.)

Esta seção cobrirá a maior parte do que você precisará no uso diário de genéricos, tanto para consumir APIs genéricas que outras pessoas criaram quanto para criar as suas próprias. Se você ficar preso ao ler este capítulo, mas quiser continuar progredindo, sugiro que você se concentre no que precisa saber para usar tipos e métodos genéricos dentro do framework e de outras bibliotecas; escrever seus próprios tipos e métodos genéricos surge com muito menos frequência do que usar os de estrutura.

Começaremos examinando uma das classes de coleção introduzidas no .NET 2.0—`Dicionário< TKey, TValue >`.

### 3.2.1 Aprender pelo exemplo: um dicionário genérico

Usar tipos genéricos pode ser simples se você não atingir algumas das limitações e começar a se perguntar o que há de errado. Você nem precisa saber nada da terminologia para adivinhar o que um trecho de código fará e, com um pouco de tentativa e erro, você pode experimentar sua maneira de escrever seu próprio código funcional.

(Um dos benefícios dos genéricos é que mais verificações são feitas em tempo de compilação, então é mais provável que você tenha um código funcional quando tudo for compilado – isso torna a experimentação mais simples.) É claro que o objetivo deste capítulo é para desenvolver seu conhecimento de forma que você *não* use suposições – você saberá o que está acontecendo em cada estágio.

Por enquanto, vamos dar uma olhada em alguns códigos que são simples, mesmo que a sintaxe não seja familiar. A listagem a seguir usa um `Dictionary< TKey, TValue >` (aproximadamente o equivalente genérico da classe `Hashtable` não genérica) para contar as frequências de palavras em um determinado trecho de texto.

#### Listagem 3.1 Usando um `Dictionary< TKey, TValue >` para contar palavras no texto

```
Dicionário estático<string,int> CountWords(string texto)
{
    Dicionário<string,int> frequências; frequências = new
    Dicionário<string,int>();

    string[] palavras = Regex.Split(texto, @"\W+");
    foreach (string palavra em palavras) {

        if (frequências.ContainsKey(palavra)) {
            frequências[palavra]++;
        }
    }
}
```

**B** Cria um novo mapa da palavra à frequência

**C** Divide o texto em palavras

**D** Adiciona ou atualiza o mapa

```

outro
{
    frequências[palavra] = 1;
}

} frequências de retorno;
}
...
string text = @"Você gosta de ovos verdes e presunto?
Eu não gosto deles, Sam-eu-sou.
Não gosto de ovos verdes e presunto.";

Dicionário<string,int> frequências = CountWords(text); foreach (KeyValuePair<string,int>
entrada em frequências {

    string palavra = entrada.Key; frequência
    interna = entrada.Value; Console.WriteLine
    ("{0}: {1}", palavra, frequência);
}

```

**D** Adiciona ou atualiza o mapa

**E** Imprime cada par chave/valor do mapa

O método CountWords primeiro cria um mapa vazio da string para o int B. Isso contará efetivamente com que frequência cada palavra é usada no texto fornecido. Você então usa uma expressão regular **C** para dividir o texto em palavras. É grosseiro - você acaba com uma string vazia devido ao ponto final no final do texto, e *do* e *Do* são contados separadamente.

Esses problemas são facilmente solucionáveis, mas eu queria manter o código o mais simples possível neste exemplo.

Para cada palavra, você verifica se ela já está no mapa. Se for, você incrementa a contagem existente; caso contrário, você atribui à palavra uma contagem inicial de 1 D. Observe como o código de incremento não precisa fazer uma conversão para int para realizar a adição; o valor que você recupera é conhecido como int em tempo de compilação. A etapa que aumenta a contagem é, na verdade, executar um get no indexador do mapa, depois incrementar e, em seguida, executar um conjunto no indexador. Você pode achar mais fácil manter isso explícito, usando `frequências[palavra] = frequências[palavra] + 1;` em vez de.

A parte final da listagem é familiar: enumerar por meio de uma Hashtable fornece uma entrada de dicionário semelhante (não genérica) com propriedades chave e valor para cada entrada E. Mas em C# 1, você precisaria converter a palavra e a frequência, porque a chave e o valor teriam sido retornados apenas como objeto. Isso também significa que a frequência teria sido encaixotada. É certo que você não *precisa* colocar a palavra e a frequência em variáveis - você poderia ter feito uma única chamada para `Console.WriteLine` e passado `entry.Key` e `entry.Value` como argumentos. Incluí as variáveis aqui para deixar claro que nenhuma conversão é necessária.

Agora que você viu um exemplo, vamos ver o que significa falar sobre `Dictionary< TKey, TValue >` em primeiro lugar. O que são TKey e TValue e por que eles têm colchetes angulares ao redor deles?

### 3.2.2 Tipos genéricos e parâmetros de tipo

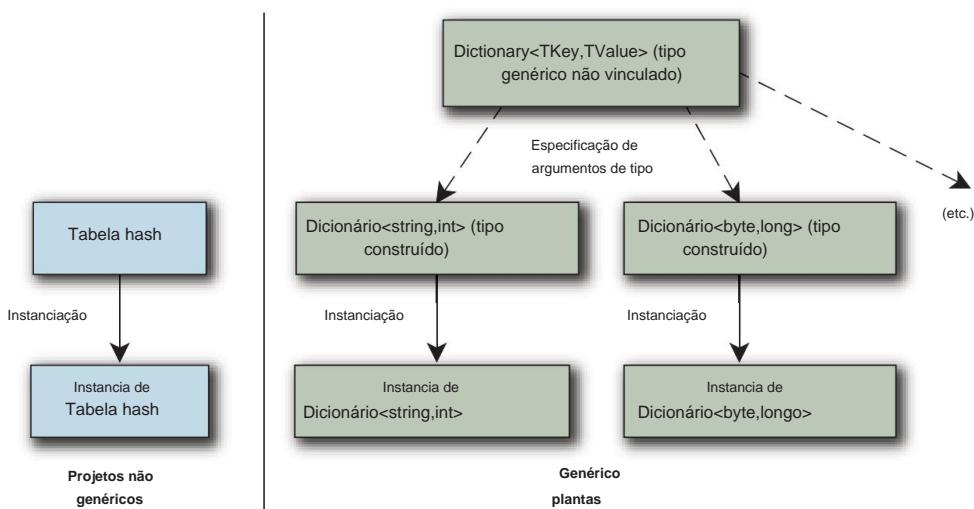
Existem duas formas de genéricos em C#: *tipos genéricos* (incluindo classes, interfaces, delegados e estruturas – não há enumerações genéricas) e *métodos genéricos*. Ambos são essencialmente maneiras de expressar uma API (seja para um único método genérico ou para um tipo genérico inteiro), de modo que em alguns lugares onde você esperaria ver um tipo normal, você verá um *parâmetro de tipo*.

Um parâmetro de tipo é um espaço reservado para um tipo real. Os parâmetros de tipo aparecem entre colchetes angulares em uma declaração genérica, usando vírgulas para separá-los. Portanto, em `Dictionary< TKey, TValue >`, os parâmetros de tipo são `TKey` e `TValue`. Ao usar um tipo ou método genérico, você especifica os tipos *reais* que deseja usar. Eles são chamados de *argumentos de tipo* — por exemplo, na listagem 3.1 os argumentos de tipo eram `string` (para `TKey`) e `int` (para `TValue`).

**ALERTA DE JARGO!** Muita terminologia detalhada está envolvida em genéricos. Incluí-o para referência – e porque ocasionalmente torna mais fácil falar sobre tópicos com precisão.

Também pode ser útil se você precisar consultar as especificações da linguagem, mas é improvável que precise dessa terminologia na vida cotidiana. Apenas sorria e aguente por enquanto. Grande parte dessa terminologia é definida na seção 4.4 da especificação C# 5 (“Tipos Construídos”) – procure lá para obter mais detalhes.

A forma de um tipo genérico em que nenhum dos parâmetros de tipo foi fornecido com argumentos de tipo é chamada de *tipo genérico não vinculado*. Quando argumentos de tipo são especificados, o tipo é considerado um *tipo construído*. Tipos genéricos não vinculados são efetivamente modelos para tipos construídos, da mesma forma que os tipos (genéricos ou não) podem ser considerados modelos para objetos. É uma espécie de camada extra de abstração. A Figura 3.1 mostra isso graficamente.



**Figura 3.1** Tipos genéricos não vinculados atuam como modelos para tipos construídos, que então atuam como modelos para objetos reais, assim como fazem os tipos não genéricos.

Como complicaçāo adicional, os tipos podem ser abertos ou fechados. Um *tipo aberto* é aquele que ainda envolve um parāmetro de tipo (como um dos argumentos de tipo ou como o tipo de elemento da matriz, por exemplo), enquanto um *tipo fechado* é aquele que nāo estā aberto; cada aspecto do tipo é conhecido com precisāo. Na verdade, todo cōdigo é executado no contexto de um tipo construído fechado. A nūica vez que vocē verá um tipo genérico nāo vinculado no cōdigo C# (exceto como um declaração) estā dentro do operador `typeof`, que vocē encontrará na seção 3.4.4.

A ideia de um parāmetro de tipo “receber” informações e um argumento de tipo “fornecer” as informações – as linhas tracejadas na figura 3.1 – é exatamente a mesma que com parāmetros e argumentos de método, embora os argumentos de tipo tenham que ser tipos em vez do que apenas valores arbitrários. O argumento de tipo deve ser conhecido em tempo de compilação, mas pode ser (ou pode envolver) um parāmetro de tipo do contexto relevante.

Vocē pode pensar em um tipo fechado como tendo a API do tipo aberto, mas com o tipo parāmetros sendo substituídos por seus argumentos de tipo correspondentes.<sup>2</sup> A Tabela 3.1 mostra alguns métodos públicos e declarações de propriedades do dicionário de tipo aberto `< TKey, TValue >` e o membro equivalente no tipo fechado que vocē construiu a partir dele— Dicionário`< string, int >`.

**Tabela 3.1 Exemplos de como assinaturas de métodos em tipos genéricos contêm espaços reservados, que sāo substituídos quando os argumentos de tipo sāo especificados**

Assinatura de método em tipo genérico	Assinatura do método apóis substituição do parāmetro de tipo
<code>void Add (chave TKey, valor TValue)</code>	<code>void Add(chave de string, valor int)</code>
<code>TValue this[ TKey key ] { get; definir; }</code>	<code>int esta[string chave] { obter; definir; }</code>
<code>bool ContémValue(valor TValue)</code>	<code>bool ContémValue(valor interno)</code>
<code>bool ContémKey (chave TKey)</code>	<code>bool ContainsKey(chave de string)</code>

Uma coisa importante a notar é que nenhum dos métodos da tabela 3.1 é realmente métodos genéricos. Eles sāo métodos normais dentro de um tipo genérico e acontecem use os parāmetros de tipo declarados como parte do tipo. Veremos métodos genéricos em a próxima seção.

Agora que vocē sabe o que significam `TKey` e `TValue` e o que sāo os colchetes angulares pois, vocē pode ver como seriam as declarações na tabela 3.1 dentro da declaração de classe. Esta é a aparência do código para `Dictionary< TKey, TValue >`, embora as implementações reais do método estejam faltando e haja mais membros na realidade:

```
namespace System.Collections.Generic
{
    dicionário de classe pública< TKey, TValue >
        : IEnumerable< KeyValuePair< TKey, TValue > >
```



<sup>2</sup> Nem sempre funciona *exatamente* assim - hā casos que falham quando vocē aplica essa regra simples - mas é uma maneira fácil de pensar sobre os genéricos que funciona na grande maioria das situações.

```

{
    dicionário público() { ... }

    Declarar método usando parâmetros de tipo → public void Add(chave TKey, valor TValue) { ... }

    public TValue isto[chave TKey] {
        Declarar construtor sem parâmetros ←
        usando parâmetros de tipo →
        prepare-se { ... }

    }
}

public bool ContainsValue (valor TValue) { ... }

public bool ContainsKey (chave TKey) { ... }

[... outros membros ...]
}
}

```

Observe como `Dictionary< TKey, TValue >` implementa a interface genérica `IEnumerable<KeyValuePair< TKey, TValue >>` (e muitas outras interfaces na vida real). Quaisquer argumentos de tipo especificados para a classe são aplicados à interface onde os mesmos parâmetros de tipo são usados, portanto, neste exemplo, `Dictionary< string, int >` implementa `IEnumerable< KeyValuePair< string, int >>`. Essa é uma interface duplamente genérica - é a interface `IEnumerable< T >` com a estrutura `KeyValuePair< string, int >` como argumento de tipo. É porque implementa essa interface que a listagem 3.1 foi capaz de enumerar as chaves e os valores da mesma forma.

Também vale ressaltar que o construtor não lista os parâmetros de tipo entre colchetes angulares. Os parâmetros de tipo pertencem ao *tipo* e não ao construtor específico, e é aí que são declarados. Os membros só declaram parâmetros de tipo quando introduzem novos — e somente métodos podem fazer isso.

**PRONUNCIANDO GENÉRICOS** Se você precisar descrever um tipo genérico para um colega, é convencional usar “of” para introduzir os parâmetros ou argumentos de tipo – então `List< T >` é pronunciado “lista de *T*”, por exemplo. Em VB, isso faz parte da linguagem: o próprio tipo seria escrito como `List(Of T)`. Quando existem vários parâmetros de tipo, acho que faz sentido separá-los com uma palavra apropriada ao significado do tipo geral, então eu falaria sobre um “dicionário de *string* para *int*” para enfatizar o aspecto *do mapeamento*, mas uma “tupla de *string* e *int*”.

Os tipos genéricos podem efetivamente ser sobrecarregados no número de parâmetros de tipo, portanto, você pode definir `MyType`, `MyType< T >`, `MyType< T, U >`, `MyType< T, U, V >` e assim por diante, todos dentro do mesmo namespace. Os nomes dos parâmetros de tipo não são usados quando se considera isso — apenas quantos existem. Esses tipos não estão relacionados, exceto no nome — não há conversão padrão de um para outro, por exemplo. O mesmo se aplica a métodos genéricos: dois métodos podem ser exatamente iguais em assinatura, exceto no número de parâmetros de tipo. Embora isso possa parecer uma receita para o desastre, pode ser útil se você quiser aproveitar as vantagens da *inferência genérica de tipos*, onde o compilador pode resolver alguns dos argumentos de tipo para você. Voltaremos a isso na seção 3.3.2.

**CONVENÇÕES DE NOMES PARA PARÂMETROS DE TIPO** Embora você pudesse ter um tipo com parâmetros de tipo T, U e V, isso não daria muita indicação do que eles realmente significam ou como deveriam ser usados. Compare isso com o Dicionário `< TKey, TValue >`, onde é óbvio que TKey representa o tipo das chaves e TValue representa o tipo dos valores. Onde você tem um único parâmetro de tipo e seu significado é claro, T é convencionalmente usado (`List<T>` é um bom exemplo disso). Vários parâmetros de tipo geralmente devem ser nomeados de acordo com o significado, usando o prefixo T para indicar um parâmetro de tipo. De vez em quando, você pode se deparar com um tipo com vários parâmetros de tipo de letra única (`SynchronizedKeyedCollection<K,T>`, por exemplo), mas você deve tentar evitar criar a mesma situação sozinho.

Agora que você tem uma ideia do que os tipos genéricos fazem, vamos dar uma olhada nos métodos genéricos.

### 3.2.3 Métodos genéricos e leitura de declarações genéricas

Mencionei métodos genéricos algumas vezes, mas ainda não conhecemos nenhum. Você pode achar a ideia geral dos métodos genéricos mais confusa do que os tipos genéricos – eles são, de certa forma, menos naturais para o cérebro – mas é o mesmo princípio básico. Você está acostumado com os parâmetros e o valor de retorno de um método que tem tipos firmemente especificados e viu como um tipo genérico pode usar seus parâmetros de tipo em declarações de método. Os métodos genéricos vão um passo além: mesmo que você saiba exatamente com qual tipo construído está lidando, um método individual também pode ter parâmetros de tipo.

Não se preocupe se você ainda não sabe disso – é provável que o conceito dê certo em algum momento, depois que você tiver visto exemplos suficientes.

`Dictionary< TKey, TValue >` não possui nenhum método genérico, mas seu vizinho próximo `List<T>` possui. Como você pode imaginar, `List<T>` é apenas uma lista de itens de qualquer tipo especificado – `List<string>` é uma lista de strings, por exemplo. Lembrando que T é o parâmetro de tipo para toda a classe, vamos dissecar uma declaração genérica de método.

A Figura 3.2 identifica as diferentes partes da declaração do método `ConvertAll`.

Quando você olha para uma declaração genérica - seja para um tipo genérico ou para um método genérico - tentar descobrir o que isso significa pode ser assustador, especialmente se você tiver que lidar com tipos genéricos de tipos genéricos, como fez quando viu a interface implementada pelo dicionário. O segredo é não entrar em pânico – apenas encarar as coisas com calma e escolher um exemplo de situação. Use um tipo diferente para cada parâmetro de tipo e aplique todos eles de forma consistente.

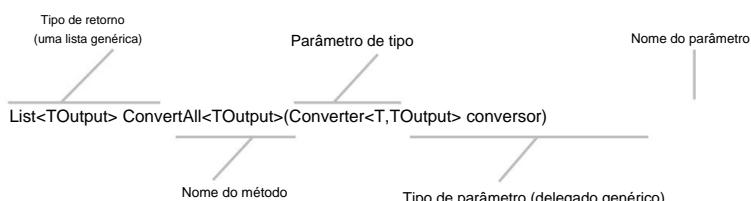


Figura 3.2 A anatomia de uma declaração de método genérico

Neste caso, vamos começar substituindo o parâmetro type do tipo que contém o método (a parte <T> de List<T>). Manteremos o conceito de lista de strings e substituiremos T por string em todos os lugares da declaração do método:

```
List<TOutput> ConvertAll<TOutput>(Conversor<string,TOutput> conversor)
```

Parece um pouco melhor, mas você ainda precisa lidar com o TOutput . Você pode dizer que é um parâmetro de tipo de método (desculpas pela terminologia confusa) porque está entre colchetes logo após o nome do método, então vamos tentar outro tipo familiar — Guid — como argumento de tipo para TOutput. Novamente você substitui o parâmetro type pelo argumento type em todos os lugares. Agora você pode pensar no método como se ele não fosse genérico, removendo a parte do parâmetro de tipo da declaração:

```
List<Guid> ConvertAll(Converter<string,Guid> conversor)
```

Agora tudo é expresso em termos de um tipo concreto, então é mais fácil pensar nisso.

Mesmo que o método real seja genérico, vamos tratá-lo como se não fosse, para melhor entendê-lo. Vamos examinar os elementos desta declaração da esquerda para a direita:

- ÿ O método retorna um List<Guid>.
- ÿ O nome do método é ConvertAll.
- ÿ O método possui um único parâmetro: a Converter<string,Guid> chamado conversor.

Agora você só precisa saber o que é Converter<string,Guid> e pronto. Não é de surpreender que Converter<string,Guid> seja um *tipo delegado genérico* construído (o tipo não associado é Converter<TInput,TOutput>), que é usado para converter uma string em um GUIA.

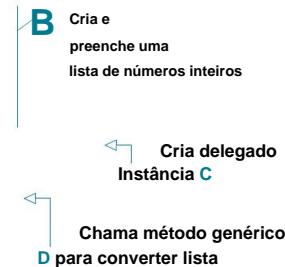
Então você tem um método que pode operar em uma lista de strings, usando um conversor para produzir uma lista de GUIDs. Agora que você entende a assinatura do método, fica mais fácil entender a documentação, que confirma que esse método faz o óbvio e cria um novo List<Guid>, converte cada elemento da lista original no tipo de destino, adicionando-o ao novo lista e, em seguida, retorna essa lista. Pensar na natureza da assinatura em termos concretos proporciona um modelo mental mais claro e torna mais simples pensar sobre o que o método pode fazer. Embora esta técnica possa parecer um tanto simplista, ainda hoje a considero útil para métodos complicados. Algumas das assinaturas do método LINQ com quatro parâmetros de tipo são feras temíveis, mas colocá-las em termos concretos as domesticaria significativamente.

Só para provar que não estou conduzindo você pelo caminho do jardim, vamos dar uma olhada no método ConvertAll em ação. A listagem a seguir mostra a conversão de uma lista de inteiros em uma lista de números de ponto flutuante, onde cada elemento da segunda lista é a raiz quadrada do elemento correspondente na primeira lista. Após a conversão, os resultados são impressos.

### Listagem 3.2 O método `List<T>.ConvertAll<TOoutput>` em ação

```
duplo estático TakeSquareRoot (int x)
{
    retornar Math.Sqrt(x);
}
...
Lista<int> inteiros = new Lista<int>(); inteiros.Add(1);
inteiros.Add(2);
inteiros.Add(3);
inteiros.Add(4);
Converter<int,duplo>
conversor = TakeSquareRoot; Lista<double> duplas = 
inteiros.ConvertAll<double>(conversor);
foreach (duplo d em duplas)

{
    Console.WriteLine(d);
}
```



A criação e o preenchimento da lista **B** são bastante simples – é apenas uma lista de inteiros fortemente tipada. A atribuição ao conversor **C** usa um recurso de delegados (conversões de grupos de métodos) que é novo no C# 2 e que discutiremos com mais detalhes na seção 5.2. Embora eu não goste de usar um recurso antes de descrevê-lo completamente, a linha seria muito longa para caber na página com a sintaxe de delegação C# 1.

No entanto, ele faz o que você espera. Em **D** você chama o método genérico, especificando o argumento de tipo para o método da mesma forma que viu para os tipos genéricos. Esta é uma situação em que você poderia ter usado a inferência de tipo para evitar especificar explicitamente o argumento de tipo, mas eu queria dar um passo de cada vez. Escrever a lista que foi retornada é simples e, ao executar o código, você verá que ele imprime 1, 1.414..., 1.732... e 2, conforme esperado.

Qual é o sentido de tudo isso? Poderíamos simplesmente ter usado um loop `foreach` para percorrer os números inteiros e imprimir a raiz quadrada imediatamente, é claro, mas não é incomum querer converter uma lista de um tipo em uma lista de outro executando alguma lógica nela. O código para fazer isso manualmente é simples, mas é mais fácil ler uma versão que faz isso em uma única chamada de método. Geralmente é assim que acontece com métodos genéricos – eles geralmente fazem coisas que anteriormente você teria feito “à mão” com prazer, mas que são mais simples com uma chamada de método. Antes dos genéricos, poderia ter havido uma operação semelhante a `ConvertAll` em `ArrayList` convertendo de objeto para objeto, mas teria sido muito menos satisfatório. Métodos anônimos (veja a seção 5.4) também ajudam aqui – se você não quisesse introduzir um método extra, você poderia ter especificado a conversão in-line. As expressões LINQ e lambda levam esse padrão muito além, como você verá na parte 3 do livro.

Observe que os métodos genéricos também podem fazer parte de tipos não genéricos. A listagem a seguir mostra um método genérico sendo declarado e usado dentro de um tipo normal não genérico.

**Listagem 3.3 Implementando um método genérico em um tipo não genérico**

```
static List<T> MakeList<T>(T primeiro, T segundo) {
    Lista<T> lista = new Lista<T>();
    lista.Adicionar(primeiro);
    lista.Adicionar(segundo);
    lista.de retorno;
}
...
List<string> list = MakeList<string>("Linha 1", "Linha 2"); foreach (string x na lista) {
    Console.WriteLine(x);
}
```

O método genérico `MakeList<T>` precisa apenas de um parâmetro de tipo (`T`). Tudo o que faz é construir uma lista contendo os dois parâmetros. É importante notar que você pode usar `T` como argumento de tipo ao criar `List<T>` no método. Assim como quando estávamos olhando para declarações genéricas, você pode pensar na implementação como (grosso modo) substituindo todas as menções de `T` por `string`. Ao chamar o método, você usa a mesma sintaxe que viu antes para especificar os argumentos de tipo.

Tudo bem até agora? Agora você deve dominar os genéricos simples. Receio que haja um pouco mais de complexidade por vir, mas se você está satisfeito com a ideia fundamental dos genéricos, você superou o maior obstáculo. Não se preocupe se ainda estiver um pouco confuso (especialmente quando se trata da terminologia aberta/fechada/não vinculada/construída), mas agora seria um bom momento para fazer algumas experiências para que você possa ver os genéricos em ação antes de partir. mais. Se você nunca usou as coleções genéricas antes, talvez queira dar uma olhada rápida no apêndice B, que descreve o que está disponível. Os tipos de coleção fornecem um ponto de partida simples para brincar com genéricos e são amplamente usados em quase todos os programas .NET não triviais.

Uma coisa que você pode descobrir ao experimentar é que é difícil percorrer apenas parte do caminho. Depois de tornar uma parte de uma API genérica, muitas vezes você descobrirá que precisa retrabalhar outro código, tornando-o genérico também ou inserindo as conversões exigidas pelas novas chamadas de método com tipagem mais forte. Uma alternativa seria ter uma implementação fortemente tipada, usando classes genéricas nos bastidores, mas deixando uma API fracamente tipada por enquanto. Com o passar do tempo, você ficará mais confiante sobre quando é apropriado usar genéricos.

### 3.3 Além do básico

Os usos relativamente simples de genéricos que vimos até agora podem ajudar você muito, mas existem mais alguns recursos que podem ajudá-lo a ir mais longe.

Começaremos examinando as *restrições de tipo*, que oferecem mais controle sobre quais argumentos de tipo podem ser especificados. Eles são úteis ao criar seus próprios tipos e métodos genéricos, e você precisará entendê-los para saber quais opções estão disponíveis ao usar a estrutura também.

Em seguida, examinaremos a *inferência de tipo* – um truque útil do compilador que permite não declarar explicitamente os argumentos de tipo ao usar métodos genéricos. Você não precisa usá-lo, mas pode tornar seu código muito mais fácil de ler quando usado adequadamente.

Você verá na parte 3 do livro que o compilador C# está gradualmente sendo autorizado a inferir muito mais informações do seu código, enquanto ainda mantém a linguagem segura e digitada estaticamente.<sup>3</sup> A última parte desta

seção trata da obtenção do valor padrão de um parâmetro de tipo e as comparações disponíveis quando você escreve código genérico. Encerraremos com um exemplo que demonstra a maioria dos recursos que abordamos e que é uma classe útil por si só.

Embora esta seção se aprofunde um pouco mais nos genéricos, não há nada *realmente* difícil nisso. Há muito o que lembrar, mas todos os recursos têm um propósito e você ficará grato por eles quando precisar deles. Vamos começar.

### 3.3.1 Restrições de tipo

Até agora, todos os parâmetros de tipo que vimos podem ser aplicados a qualquer tipo — eles são *irrestritos*. Você pode ter um `List<int>`, um `Dictionary<object, FileMode>`, qualquer coisa. Tudo bem quando você lida com coleções que não precisam interagir com o que armazenam, mas nem todos os usos de genéricos são assim. Freqüentemente, você deseja chamar métodos em instâncias do parâmetro de tipo, ou criar novas instâncias, ou certificar-se de aceitar apenas tipos de referência (ou aceitar apenas tipos de valor). Em outras palavras, você deseja especificar regras que digam quais argumentos de tipo são considerados válidos para seu tipo ou método genérico. No C# 2, você faz isso com *restrições*.

Quatro tipos de restrições estão disponíveis e a sintaxe geral é a mesma para todas elas. As restrições vêm no final da declaração de um método ou tipo genérico e são introduzidas pela palavra-chave contextual `where`. Eles podem ser combinados de maneiras sensatas, como você verá mais tarde. Porém, primeiro exploraremos cada tipo de restrição.

#### RESTRIÇÕES DE TIPO DE REFERÊNCIA

O primeiro tipo de restrição garante que o argumento de tipo usado seja um tipo de referência.

É expresso como `T : class` e deve ser a primeira restrição especificada para esse parâmetro de tipo. O argumento de tipo pode ser qualquer classe, interface, array, delegado ou outro parâmetro de tipo que já seja conhecido como um tipo de referência. Por exemplo, considere a seguinte declaração:

```
struct RefSample<T> onde T: classe
```

Os tipos fechados válidos que usam esta declaração incluem

```
    RefSample<IDisposable>
    RefSample<string>
    RefSample<int[]>
```

---

<sup>3</sup> Bem, exceto qualquer código C# 4 que use *explicitamente* digitação dinâmica.

Tipos fechados inválidos incluem

- ÿ RefSample<Guid> ÿ
- RefSample<int>

Eu deliberadamente fiz do RefSample uma estrutura (e, portanto, um tipo de valor) para enfatizar a diferença entre o parâmetro de tipo restrito e o próprio tipo. RefSample <string> ainda é um tipo de valor com semântica de valor em todos os lugares - apenas usa o tipo string sempre que T é especificado no código.

Quando um parâmetro de tipo é restrinido desta forma, você pode comparar referências (incluindo nulas) com == e !=, mas esteja ciente de que, a menos que haja outras restrições, apenas as referências serão comparadas, mesmo se o tipo em questão sobrepor esses operadores (como string faz, por exemplo). Com uma restrição de tipo de conversão (descrita em breve), você pode acabar com sobreporagens *garantidas pelo compilador* de == e !=, caso em que essas sobreporagens são usadas — mas isso é relativamente raro.

#### RESTRICOES DE TIPO DE VALOR

A restrição de tipo de valor, expressa como T : struct, garante que o argumento de tipo usado seja um tipo de valor, incluindo enums. No entanto, exclui tipos anuláveis (conforme descrito no capítulo 4). Vejamos um exemplo de declaração:

classe ValSample<T> onde T: estrutura

Os tipos fechados válidos incluem

- ÿ ValSample<int> ÿ
- ValSample< FileMode >

Tipos fechados inválidos incluem

- ÿ ValSample<objeto> ÿ
- ValSample<StringBuilder>

Desta vez, ValSample é um tipo de referência, apesar de T ser restrito a ser um tipo de valor. Observe que System.Enum e System.ValueType são tipos de referência em si, portanto, não são permitidos como argumentos de tipo válidos para ValSample. Quando um parâmetro de tipo é restrito a ser um tipo de valor, as comparações usando == e != são proibidas.

Raramente uso restrições de tipo de valor ou de referência, embora você verá no próximo capítulo que tipos de valor anuláveis dependem de restrições de tipo de valor. As duas restrições restantes provavelmente serão mais úteis quando você estiver escrevendo seus próprios tipos genéricos.

#### RESTRICOES DE TIPO DE CONSTRUTOR

A restrição do tipo construtor é expressa como T : new() e deve ser a *última* restrição para qualquer parâmetro de tipo específico. Ele simplesmente verifica se o argumento de tipo usado possui um construtor sem parâmetros que pode ser usado para criar uma instância. Este é o caso de qualquer tipo de valor; para qualquer classe não estática e não abstrata sem nenhum construtor explicitamente declarado; e para qualquer classe não abstrata com um construtor público explícito sem parâmetros.

**C# VERSUS CLI STANDARDS** Há uma discrepância entre C# e CLI padrões quando se trata de tipos de valor e construtores. A especificação C# afirma que todos os tipos de valor têm um construtor padrão sem parâmetros, e o linguagem usa a mesma sintaxe para chamar ambos os construtores declarados explicitamente e o sem parâmetros, contando com o compilador para fazer a coisa certa debaixo de. A especificação CLI não tem tal requisito, mas fornece um instrução especial para criar um valor padrão sem especificar nenhum parâmetro. Você pode ver essa discrepancia em ação quando usa a reflexão para encontrar os construtores de um tipo de valor – você não verá um sem parâmetros.

Novamente, vejamos um exemplo rápido, desta vez para um método. Só para mostrar como é útil, darei também a implementação do método:

```
public T CreateInstance<T>() onde T : new()
{
    retornar novo T();
}
```

Este método retorna uma nova instância de qualquer tipo que você especificar, desde que tenha um construtor sem parâmetros. Isso significa que chamadas para `CreateInstance<int>()` e `CreateInstance<object>()` estão corretas, mas `CreateInstance<string>()` não está, porque `string` não possui um construtor sem parâmetros.

Não há como restringir parâmetros de tipo para forçar assinaturas de outros construtores. Por exemplo, você não pode especificar que deve haver um construtor que receba um único parâmetro de `string`. Pode ser frustrante, mas infelizmente é assim que as coisas são. Bem examinaremos esse problema com mais detalhes quando considerarmos as várias restrições do .NET genéricos na seção 3.5.

As restrições de tipo de construtor podem ser úteis quando você precisa usar padrões de fábrica, onde um objeto criará outro como e quando necessário. Fábricas muitas vezes precisam produzir objetos que sejam compatíveis com uma determinada interface, é claro, e é aí que entra o nosso último tipo de restrição.

#### RESTRICOES DE TIPO DE CONVERSÃO

O tipo final (e mais complicado) de restrição permite especificar outro tipo que o argumento de tipo deve ser implicitamente conversível por meio de uma identidade, referência ou conversão box-ing. Você pode especificar que um argumento de tipo seja conversível para outro tipo argumento também — isso é chamado de restrição *de parâmetro de tipo*. Essas restrições fazem com que mais difícil entender a declaração, mas podem ser úteis de vez em quando. Tabela 3.2 mostra alguns exemplos de declarações de tipo genérico com restrições de tipo de conversão, junto com exemplos válidos e inválidos de tipos construídos correspondentes.

A terceira restrição na tabela 3.2, `T : IComparable<T>`, é apenas um exemplo do uso um tipo genérico como restrição. Outras variações, como `T : List<U>` (onde `U` é outro parâmetro de tipo) e `T : IList<string>`, também estão bem.

Tabela 3.2 Exemplos de restrições de tipo de conversão

Declaração	Exemplos de tipos construídos
class Sample<T> onde T: Fluxo	Válido: Sample<Stream> (conversão de identidade) Inválido: Amostra<string>
struct Sample<T> onde T: IDisposable	Válido: Sample<SqlConnection> (conversão de referência) Inválido: Amostra<StringBuilder>
class Sample<T> onde T : IComparable<T>	Válido: Sample<int> (conversão de boxe) Inválido: Amostra<FileInfo>
classe Amostra<T,U> onde T : U	Válido: Sample<Stream, IDisposable> (conversão de referência) Inválido: Amostra<string, IDisposable>

Você pode especificar diversas interfaces, mas apenas uma classe. Por exemplo, isso é bom (se for difícil de satisfazer):

```
class Sample<T> onde T: Fluxo,  
    IEnumerable<string>,  
    IComparable<int>
```

Mas isto não é:

```
class Sample<T> onde T: Fluxo,  
    ListaArray,  
    IComparable<int>
```

De qualquer forma, nenhum tipo pode derivar diretamente de mais de uma classe, portanto, tal restrição normalmente seria impossível (como a anterior) ou parte dela seria redundante (especificando que o tipo teria que derivar de Stream e Memory - Stream , por exemplo).

Há mais um conjunto de restrições: o tipo especificado não pode ser um tipo de valor, um classe selada (como string) ou qualquer um dos seguintes tipos “especiais”:

- ÿ Sistema.Object
- ÿ Sistema.Enum
- ÿ System.ValueType
- ÿ Sistema.Delegado

**TRABALHANDO COM A FALTA DE RESTRIÇÕES ENUM E DELEGADA** A incapacidade de especificar os tipos anteriores nas restrições de tipo de conversão parece ser devido a uma restrição CLR — mas não é. Isso pode ter sido verdade historicamente (em algum momento quando os genéricos ainda estavam sendo projetados), mas se você construir o código apropriado em IL, ele funcionará bem. A especificação CLI lista até restrições de enumeração e delegação como exemplos e explica o que seria válido e o que não seria. Isso é frustrante e há muitos métodos genéricos que seriam úteis quando restritos a delegados ou enums. eu tenho um

projeto de código aberto chamado Unconstrained Melody (<http://code.google.com/p/unconstrained-melody/>), que realiza alguns hackeamentos para construir uma biblioteca de classes que *tenha* essas restrições em vários métodos utilitários. Embora o compilador C# não permita que você declare tais restrições, ele fica feliz em aplicá-las quando você chama os métodos na biblioteca. Talvez a proibição seja suspensa em uma versão futura do C#.

As restrições de tipo de conversão são provavelmente o tipo mais útil, pois significam que você pode usar membros do tipo especificado em instâncias do parâmetro de tipo. Um exemplo particularmente útil disso é `T : IComparable<T>`, que permite comparar duas instâncias de `T` de maneira significativa e direta. Veremos um exemplo disso (e discutiremos outras formas de comparação) na seção 3.3.3.

#### **COMBINANDO RESTRIÇÕES**

Mencionei a possibilidade de haver múltiplas restrições e você as viu em ação para restrições de tipo de conversão, mas não mostrei os diferentes tipos sendo combinados. Obviamente nenhum tipo pode ser tanto um tipo de referência quanto um tipo de valor, portanto essa combinação é proibida. Da mesma forma, *todo* tipo de valor tem um construtor sem parâmetros, então você não pode especificar a restrição de construção quando já tiver uma restrição de tipo de valor (embora você ainda possa usar `new T()` dentro de métodos se `T` estiver restrito a ser um tipo de valor). Se você tiver diversas restrições de tipo de conversão e uma delas for uma classe, isso deverá vir antes das interfaces — e você não poderá especificar a mesma interface mais de uma vez. Parâmetros de tipo diferentes podem ter restrições diferentes e cada um deles é introduzido com um local separado.

Vejamos alguns exemplos válidos e inválidos:

*Válido:*

```
classe Sample<T> onde T: classe, IDisposable, new() classe Sample<T> onde
T: struct, classe IDisposable Sample<T,U> onde T: classe onde U:
struct, classe T Sample<T,U> onde T: Fluxo onde U: IDisposable
```

*Inválido:*

```
class Sample<T> onde T: classe, struct class Sample<T>
onde T: Stream, class class Sample<T> onde T: new(),
Stream class Sample<T> onde T: IDisposable, Stream class
Sample<T > onde T: XmlReader, IComparable, classe IComparable
Sample<T,U> onde T: struct onde U: classe, classe T Sample<T,U> onde T: Stream, U:
IDisposable
```

Incluí o último exemplo em cada lista porque é muito fácil tentar a versão inválida em vez da versão válida, e o erro do compilador não ajuda em nada. Apenas lembre-se de que cada lista de restrições de parâmetro de tipo precisa de seu próprio local introdutório. O terceiro exemplo válido é interessante: se `U` é um tipo de valor, como ele pode derivar de `T`, que é um tipo de referência? A resposta é que `T` poderia ser um objeto ou uma interface implementada por `U`. É uma restrição bastante desagradável, no entanto.

**TERMINOLOGIA DE ESPECIFICAÇÃO** A especificação categoriza restrições de forma ligeiramente diferente – em restrições *primárias*, restrições *secundárias* e restrições *de construtor*. Uma restrição primária é uma restrição de tipo de referência, um valor restrição de tipo ou uma restrição de tipo de conversão usando uma classe. Uma restrição secundária é uma restrição de tipo de conversão usando uma interface ou outro tipo parâmetro. Não considero essas categorias particularmente úteis, mas elas fazem com que mais fácil definir a gramática das restrições: a restrição primária é opcional mas você só pode ter um; você pode ter quantas restrições secundárias desejar como; a restrição do construtor é opcional (a menos que você tenha uma restrição de tipo de valor, caso em que é proibido).

Agora que você sabe tudo o que precisa para ler declarações de tipos genéricos, vamos dar uma olhada no digite inferência de argumento que mencionei anteriormente. Na listagem 3.2 você declarou explicitamente os argumentos de tipo para `List<T>.ConvertAll`, e você fez o mesmo na listagem 3.3 para o método `MakeList` — agora vamos pedir ao compilador para resolvê-los quando puder, tornando mais simples chamar métodos genéricos.

### 3.3.2 Inferência de tipo para argumentos de tipo de métodos genéricos

Especificar argumentos de tipo ao chamar um método genérico pode muitas vezes parecer bastante redundante. Geralmente é óbvio quais devem ser os argumentos de tipo, com base em os próprios argumentos do método. Para facilitar a vida, do C# 2 em diante, o compilador pode ser inteligente de maneiras bem definidas, para que você possa chamar o método sem declarando explicitamente os argumentos de tipo. Mas antes de prosseguirmos, devo salientar que isso só é verdade para *métodos genéricos*. Não se aplica a *tipos genéricos*.

Vejamos as linhas relevantes da listagem 3.3 e vejamos como as coisas podem ser simplificadas. Aqui estão as linhas que declaram e invocam o método:

```
Lista estática<T> MakeList<T>(T primeiro, T segundo)
...
List<string> list = MakeList<string>("Linha 1", "Linha 2");
```

Observe os argumentos – ambos são strings. Cada um dos parâmetros do método é declarado como sendo do tipo T. Mesmo que você não tivesse a parte `<string>` da expressão de invocação do método, seria bastante óbvio que você pretendia chamar o método usando string como argumento de tipo para T. O compilador permite que você o omita, deixando assim:

```
List<string> list = MakeList("Linha 1", "Linha 2");
```

Isso é um pouco mais legal, não é? Pelo menos é mais curto. Isso nem sempre significa que é mais legível, é claro. Em alguns casos, será mais difícil para o leitor descobrir que tipo argumentos que você está tentando usar, mesmo que o compilador possa fazer isso facilmente. Eu recomendo isso você julga cada caso de acordo com seus méritos. Minha preferência pessoal é deixar o compilador inferir os argumentos de tipo na *maioria* dos casos em que funciona.

Observe como o compilador definitivamente sabe que você está usando string como argumento de tipo, porque a atribuição para listar também funciona, e isso ainda especifica o tipo argumento (e tem que). A atribuição não tem influência no argumento de tipo processo de inferência, no entanto. Significa apenas que se o compilador descobrir que tipo

argumentos que ele acha que você deseja usar, mas errar, você ainda provavelmente receberá um erro em tempo de compilação.

Como o compilador poderia errar? Suponha que você realmente queira usar `object` como argumento de tipo. Os parâmetros do método ainda são válidos, mas o compilador pensa que você quis usar `string`, já que ambos são strings. Alterar um dos parâmetros para ser explicitamente convertido em objeto faz com que a inferência de tipo falhe, pois um dos argumentos do método sugeriria que `T` deveria ser `string`, e o outro sugere que `T` deveria ser objeto. O compilador poderia olhar para isso e dizer que definir `T` como objeto satisfaria tudo, mas definir `T` como `string` não, mas a especificação tem apenas um número limitado de etapas a seguir. Este assunto é bastante complicado em C# 2, e C# 3 vai ainda mais longe. Não tentarei cobrir todos os detalhes básicos das regras do C# 2 aqui, mas as etapas básicas são as seguintes:

- 1** Para cada argumento do método (os bits entre parênteses normais, não os colchetes angulares), tente inferir alguns dos argumentos de tipo do método genérico, usando algumas técnicas bastante simples.
- 2** Verifique se todos os resultados da primeira etapa são consistentes. Em outras palavras, se um argumento implicasse um argumento de tipo para um parâmetro de tipo específico e outro implicasse um argumento de tipo diferente para o mesmo parâmetro de tipo, então a inferência falhará para a chamada de método.
- 3** Verifique se todos os parâmetros de tipo necessários para o método genérico foram inferidos. Você não pode deixar o compilador inferir alguns enquanto especifica outros explicitamente – é tudo ou nada.

Para evitar aprender todas as regras (e eu não recomendaria isso, a menos que você esteja particularmente interessado nos detalhes), há uma coisa simples a fazer: experimente e veja o que acontece. Se você acha que o compilador *pode* inferir todos os argumentos de tipo, tente chamar o método sem especificar nenhum. Se falhar, insira os argumentos de tipo explicitamente. Você não perde nada mais do que o tempo que leva para compilar o código uma vez e não precisa ter todo o lixo extra de advogado de linguagem em sua cabeça.

Para facilitar o uso de tipos genéricos, a inferência de tipos pode ser combinada com a ideia de sobrecarregar nomes de tipos com base no número de parâmetros de tipo. Veremos um exemplo disso daqui a pouco, quando juntarmos tudo.

### 3.3.3 Implementando genéricos

É provável que você gaste mais tempo usando tipos e métodos genéricos do que escrevendo-os você mesmo. Mesmo quando você está fornecendo a implementação, geralmente você pode apenas fingir que `T` (ou qualquer que seja o nome do seu parâmetro de tipo) é o nome de um tipo e continuar escrevendo código como se não estivesse usando genéricos. Mas há algumas coisas extras que você deve saber.

#### EXPRESSÕES DE VALOR PADRÃO

Quando você sabe exatamente com que tipo está trabalhando, você sabe seu valor padrão – o valor que um campo de outra forma não inicializado teria, por exemplo. Quando você não

Se você sabe a que tipo você está se referindo, não é possível especificar esse valor padrão diretamente. Você não pode usar null porque pode não ser um tipo de referência. Você não pode usar 0 porque pode não ser um tipo numérico.

É bastante raro precisar do valor padrão, mas pode ser útil ocasionalmente. Dicionário < TKey, TValue > é um bom exemplo - ele possui um método TryGetValue que funciona um pouco como os métodos TryParse nos tipos numéricos: ele usa um parâmetro de saída para o valor que você está tentando buscar e um valor de retorno booleano para indicar se teve sucesso. Isso significa que o método *deve* ter algum valor do tipo TValue para preencher o parâmetro de saída. (Lembre-se de que os parâmetros de saída devem ser atribuídos antes que o método retorne normalmente.)

**O PADRÃO TRYXXX** Alguns padrões no .NET são facilmente identificáveis pelos nomes dos métodos envolvidos – BeginXXX e EndXXX sugerem uma operação assíncrona, por exemplo. O padrão TryXXX teve seu uso ampliado do .NET 1.1 para 2.0. Ele foi projetado para situações que normalmente poderiam ser consideradas erros (na medida em que o método não pode cumprir sua função principal), mas onde a falha poderia ocorrer sem indicar um problema sério e não deveria ser considerada excepcional. Por exemplo, os usuários muitas vezes não conseguem digitar os números corretamente, portanto, ser capaz de *tentar* analisar algum texto sem ter que capturar uma exceção e engoli-lo é útil. Isso não apenas melhora o desempenho em caso de falha, mas, mais importante ainda, salva exceções para casos de erro genuínos em que algo está errado no sistema (por mais amplamente que você queira interpretar isso). É um padrão útil para ter na manga como designer de biblioteca, quando aplicado de forma adequada.

C# 2 fornece a expressão de valor padrão para atender exatamente a essa necessidade. A especificação não se refere a ele como um operador, mas você pode considerá-lo semelhante ao operador typeof , apenas retornando um valor diferente. A listagem a seguir mostra isso em um método genérico e também fornece um exemplo de inferência de tipo e uma restrição de tipo de conversão em ação.

#### Listagem 3.4 Comparando um determinado valor com o padrão de forma genérica

```
static int CompareToDefault<T>(valor T) onde T :  
    IComparable<T>  
{  
    valor de retorno.CompareTo(default(T));  
}  
...  
Console.WriteLine(CompareToDefault("x"));  
Console.WriteLine(CompareToDefault(10));  
Console.WriteLine(CompareToDefault(0));  
Console.WriteLine(CompareToDefault(-10));  
Console.WriteLine(CompareToDefault(DateTime.MinValue));
```

A Listagem 3.4 mostra um método genérico sendo usado com três tipos diferentes: string, int e DateTime. O método CompareToDefault determina que ele só pode ser usado com tipos que implementam a interface IComparable<T> , que permite chamar

CompareTo(T) no valor passado. O outro valor que você usa para a comparação é o valor padrão para o tipo. Como string é um tipo de referência, o valor padrão é nulo, e a documentação do CompareTo afirma que, para tipos de referência, tudo deve ser maior que nulo, então o primeiro resultado é 1. As próximas três linhas mostram comparações com o valor padrão de int, demonstrando que o valor padrão é 0. A saída da última linha é 0, mostrando que DateTime .MinValue é o valor padrão para Data hora.

É claro que o método da listagem 3.4 falhará se você passar null como argumento—a linha que chama CompareTo lançará NullReferenceException da maneira normal.

Não se preocupe com isso no momento — há uma alternativa usando IComparer<T>, como você verá em breve.

#### COMPARAÇÕES DIRETAS

Embora a listagem 3.4 tenha mostrado como uma comparação é possível, você nem sempre desejará restringir seus tipos para implementar IComparable<T> ou sua interface irmã, IEquatable<T>, que fornece um método Equals(T) fortemente tipado para complementar o método Equals (objeto) método que todos os tipos possuem. Sem as informações extras, essas interfaces lhe dão acesso, há pouco que você possa fazer em termos de comparações, além de ligar Equals(objeto), o que resultará no boxe do valor com o qual você deseja comparar quando é um tipo de valor. (Existem alguns tipos para ajudá-lo em algumas situações - nós iremos para eles em um minuto.)

Quando um parâmetro de tipo é irrestrito (nenhuma restrição é aplicada a ele), você pode usar os operadores == e !=, mas apenas para comparar um valor desse tipo com nulo; você não é possível comparar dois valores do tipo T entre si. Quando o argumento de tipo for um tipo de referência, a comparação de referência normal será usada. No caso em que o tipo argumento fornecido para T é um tipo de valor não anulável, uma comparação com nulo sempre decida que eles são desiguais (para que a comparação possa ser removida pelo JIT compilador). Quando o argumento de tipo é um tipo de valor anulável, a comparação será comportar-se de forma natural, fazendo a comparação com o valor nulo do tipo.<sup>4</sup> (Não se preocupe se esta última parte ainda não fizer sentido – fará quando você ler a próxima capítulo. Alguns recursos estão muito interligados para me permitir descrever qualquer um deles completamente sem se referir ao outro, infelizmente.)

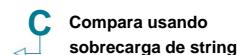
Quando um parâmetro de tipo é restrito a ser um tipo de valor, == e != não podem ser usados com isso. Quando é restrito a ser um tipo de referência, o tipo de comparação executada depende de como o parâmetro de tipo é restrito. Se a única restrição é que é um tipo de referência, comparações simples de referência são realizadas. Se for ainda mais restrito para derivar de um tipo específico que sobrecarrega os operadores == e !=, essas sobrecargas serão usadas. Porém, tenha cuidado - sobrecargas extras que podem ser disponibilizados pelo argumento de tipo especificado pelo chamador não são usados. Nas próximas

<sup>4</sup> No momento em que este livro foi escrito (testando com .NET 4.5 e versões anteriores), o código gerado pelo compilador JIT para comparar valores de parâmetros de tipo irrestritos com null era extremamente lento para tipos de valor anuláveis. Se você restringir um parâmetro de tipo T para não ser anulável e depois comparar um valor do tipo T? contra null, essa comparação é muito mais rápida. Isso mostra alguma margem para otimização futura do JIT.

listagem demonstra isso com uma restrição de tipo de referência simples e um argumento de tipo de corda.

#### Listagem 3.5 Comparações usando == e != realizando comparações de referência

```
static bool AreReferencesEqual<T>(T primeiro, T segundo)
    onde T: classe
{
    retornar primeiro == segundo;
}
...
string nome = "Jon";
string intro1 = "Meu nome é " string intro2 = "Meu  + nome;
nome é " + nome;
Console.WriteLine(introdução1 == introdução2);
Console.WriteLine(AreReferencesEqual(intro1, intro2));
```



Mesmo que a string sobrecarregue == (como demonstrado pela comparação na impressão C True), essa sobrecarga não é usada pela comparação em B. Basicamente, quando AreReferencesEqual<T> é compilado, o compilador não sabe quais sobrecargas estarão disponíveis - é como se os parâmetros passados fossem do tipo object.

Isso não é específico para operadores — ao encontrar um tipo genérico, o compilador resolve todas as sobrecargas de método ao compilar o tipo genérico não vinculado, em vez do que reconsiderar cada chamada de método possível para sobrecargas mais específicas na execução tempo. Por exemplo, uma instrução de Console.WriteLine(default(T)); sempre resolve uma chamada para Console.WriteLine(valor do objeto) - ele não chama Console.WriteLine(string value) quando T é string. Isto é semelhante à situação normal de sobrecargas sendo escolhidas em tempo de compilação em vez de tempo de execução, mas os leitores familiarizados com modelos em C++ podem se surpreender mesmo assim.<sup>5</sup>

Duas classes que são *extremamente* úteis quando se trata de comparar valores são EqualityComparer<T> e Comparer<T>, ambos no System.Collections.Generic espaço para nome. Eles implementam IEqualityComparer<T> e IComparer<T>, respectivamente, e a propriedade Default retorna uma implementação que geralmente faz a coisa certa coisa para o tipo apropriado.

**AS INTERFACES DE COMPARAÇÃO GENÉRICAS** Existem quatro interfaces genéricas principais para comparações. Dois deles —IComparer<T> e IComparable<T>— tratam de comparar valores para *ordenação* (é um valor menor, igual ou maior que o outro?) e os outros dois — IEqualityComparer<T> e IEquatable<T> — serve para comparar dois itens quanto à *igualdade* de acordo com alguns critérios e para encontrar o hash de um item (de maneira compatível com a mesma noção de igualdade).

Dividindo os quatro de outra forma, IComparer<T> e IEqualityComparer<T> são implementados por tipos que são capazes de comparar dois valores diferentes, enquanto uma instância de IComparable<T> ou IEquatable<T> é capaz de se comparar com outro valor.

<sup>5</sup> Você verá no Capítulo 14 que a digitação dinâmica oferece a capacidade de resolver sobrecargas em tempo de execução.

Consulte a documentação para obter mais detalhes e considere usar estes (e tipos semelhantes, como StringComparer) ao realizar comparações. Usaremos EqualityComparer<T> no próximo exemplo.

#### **EXEMPLO COMPLETO DE COMPARAÇÃO: REPRESENTANDO UM PAR DE VALORES**

Para finalizar nossa seção sobre implementação de genéricos, aqui está um exemplo completo. Ele implementa um tipo genérico útil — um Pair<T1,T2> que mantém dois valores juntos, como um par chave/valor, mas sem expectativas quanto ao relacionamento entre os dois valores.

**.NET 4 E TUPLES** O .NET 4 fornece muitas dessas funcionalidades prontas para uso — e também para muitos números diferentes de parâmetros de tipo. Procure Tuple<T1>, Tuple<T1,T2> e assim por diante no namespace System .

Além de fornecer propriedades para acessar os próprios valores, você substituirá Equals e GetHashCode para permitir que instâncias do seu tipo funcionem bem quando usadas como chaves em um dicionário. A listagem a seguir fornece o código completo.

#### **Listagem 3.6 Classe genérica representando um par de valores**

```
usando o sistema;
usando System.Collections.Generic;

classe selada pública Par<T1, T2> : IEquatable<Pair<T1, T2>> {

    privado estático somente leitura IEqualityComparer<T1> FirstComparer =
        EqualityComparer<T1>.Default;
    privado estático somente leitura IEqualityComparer<T2> SecondComparer =
        EqualityComparer<T2>.Default;

    privado somente leitura T1 primeiro;
    segundo T2 somente leitura privado;

    par público(T1 primeiro, T2 segundo) {

        isto.primeiro = primeiro;
        este.segundo = segundo;
    }

    public T1 Primeiro { obter { retornar primeiro; } }

    público T2 Segundo { get { return segundo; } }

    public bool Equals(Pair<T1, T2> outro) {

        retornar outro! = null &&
            FirstComparer.Equals(este.Primeiro, outro.Primeiro) &&
            SecondComparer.Equals(este.Segundo, outro.Segundo);
    }

    substituição pública bool Equals(objeto o) {

        return Equals(o as Pair<T1, T2>);
    }
}
```

```

substituição pública int GetHashCode() {

    retornar FirstComparer.GetHashCode(primeiro) * 37 +
        SecondComparer.GetHashCode(segundo);
}
}

```

A Listagem 3.6 é direta. Os valores constituintes são armazenados em variáveis de membro digitadas apropriadamente e o acesso é fornecido por propriedades simples somente leitura. Você implementa `IEquatable<Pair<T1,T2>>` para fornecer uma API fortemente tipada que evitará verificações desnecessárias no tempo de execução. Os cálculos de igualdade e código hash usam o comparador de igualdade padrão para os dois parâmetros de tipo – eles tratam nulos automaticamente, o que torna o código um pouco mais simples. As variáveis estáticas usadas para armazenar os comparadores de igualdade para T1 e T2 estão lá principalmente para formatar o código da página impressa, mas também serão úteis como ponto de referência na próxima seção.

**CALCULANDO CÓDIGOS HASH** A fórmula usada para calcular o código hash com base nos resultados das duas “partes” vem de *Effective Java*, 2<sup>a</sup> edição (Addison-Wesley, 2008), de Joshua Bloch. Certamente não garante uma boa distribuição de códigos hash, mas na minha opinião é melhor do que usar um OR exclusivo bit a bit. Consulte *Java eficaz* para obter mais detalhes e para muitas outras dicas úteis e insights de design.

Agora que você tem sua classe `Pair`, como construir uma instância dela? No momento, você precisaria usar algo assim:

```
Par<int,string> par = new Par<int,string>(10, "valor");
```

Isso não é muito legal. Seria bom usar inferência de tipos, mas isso só funciona para métodos genéricos e você não possui nenhum deles. Se você colocar um método genérico no tipo genérico, ainda precisará especificar os argumentos de tipo para o tipo antes de poder chamar um método nele, o que anularia o propósito. A solução é usar uma classe auxiliar não genérica com um método genérico, conforme mostrado na listagem a seguir.

#### Listagem 3.7 Usando um tipo não genérico com um método genérico para permitir a inferência de tipos

```

classe estática pública Par {

    public static Pair<T1,T2> Of<T1,T2>(T1 primeiro, T2 segundo) {

        retornar novo Par<T1,T2>(primeiro, segundo);
    }
}

```

Se esta é a primeira vez que você lê este livro, ignore o fato de que a classe é declarada como estática — falaremos disso no capítulo 7. O ponto importante é que você tem uma classe não genérica com um método genérico. Isso significa que você pode transformar o exemplo anterior nesta versão muito mais agradável:

```
Par<int,string> par = Pair.Of(10, "valor");
```

Em C# 3 você poderia até dispensar a digitação explícita da variável pair , mas vamos não nos adiantarmos. Este uso de classes auxiliares não genéricas (ou *parcialmente* genéricas classes auxiliares, se você tiver dois ou mais parâmetros de tipo e quiser inferir alguns deles mas deixe os outros explícitos) é um truque útil.

Terminamos de examinar os recursos intermediários agora. Eu percebo que tudo pode parecer complicado no começo, mas não desanime; os benefícios dos genéricos superam em muito os complexidade adicional. Com o tempo, eles se tornam uma segunda natureza. Agora que você tem o Classe par como exemplo, pode valer a pena examinar sua própria base de código para ver se existem alguns padrões que você continua reimplementando apenas para usar tipos diferentes.

Em qualquer tópico grande, sempre há mais para aprender. A próxima seção levará você através dos tópicos avançados mais importantes em genéricos. Se você estiver se sentindo sobrecarregado neste momento, você pode querer pular para o conforto relativo da seção 3.5, onde exploraremos algumas das limitações dos genéricos. Vale a pena entender o eventualmente, tópicos na próxima seção , mas se tudo até agora foi novo para você, não fará mal nenhum ignorá-lo por enquanto.

### 3.4 Genéricos avançados

Você pode esperar que eu afirme que no restante deste capítulo cobriremos todos os aspectos do genéricos que não vimos até agora. Mas há *tantos* recantos envolvendo genéricos que isso simplesmente não é possível – e eu certamente não gostaria de leia sobre todos os detalhes e muito menos escreva sobre eles. Felizmente, as pessoas legais da A Microsoft e a ECMA escreveram todos os detalhes na especificação da linguagem, então se você quiser verificar alguma situação obscura que não é abordada aqui, isso deve seja seu próximo porto de escala. Infelizmente não posso apontar uma área específica da especificação que abrange genéricos: eles aparecem em quase todos os lugares. Indiscutivelmente, se o seu o código termina em um caso tão complicado que você precisa consultar a especificação para descobrir o que ele deve fazer, você deve refatorá-lo em uma forma mais óbvia de qualquer forma; você não quer que cada engenheiro de manutenção de agora até a eternidade tenha que leia os detalhes sangrentos.

Meu objetivo com esta seção é cobrir tudo o que você *provavelmente* deseja saber sobre genéricos. Falarei mais sobre o CLR e o lado do framework do que sobre a sintaxe específica da linguagem C# 2, embora tudo seja relevante ao desenvolver em C#.

Começaremos considerando membros estáticos de tipos genéricos, incluindo inicialização de tipo. A partir daí, é um passo natural perguntar-se como tudo isso é implementado sob o capas, mas vamos mantê-lo bastante leve nos detalhes, concentrando-nos nos efeitos importantes de as decisões de implementação. Veremos o que acontece quando você enumera um coleção genérica usando foreach em C# 2 e finalize a seção vendo como a reflexão no .NET Framework é afetada por genéricos.

### 3.4.1 Campos estáticos e construtores estáticos

Assim como os campos de instância pertencem a uma instância, os campos estáticos pertencem ao tipo em que são declarados. Se você declarar um campo estático `x` na classe `SomeClass`, haverá exatamente um campo `Some-Class.x`, não importa quantas instâncias de `SomeClass` você create, e não importa quantos tipos derivam de `SomeClass`<sup>6</sup>. Esse é o cenário familiar do C# 1. Então, como ele é mapeado para os genéricos?

A resposta é que cada tipo *fechado* possui seu próprio conjunto de campos estáticos. Você viu isso na listagem 3.6 quando armazenou os comparadores de igualdade padrão para `T1` e `T2` em campos estáticos, mas vamos examinar isso com mais detalhes com outro exemplo. A listagem a seguir cria um tipo genérico incluindo um campo estático. Você define o valor do campo para diferentes tipos fechados e depois imprime os valores para mostrar que eles estão separados.

#### Listagem 3.8 Prova de que diferentes tipos fechados possuem diferentes campos estáticos

```
classe TypeWithField<T> {

    campo de string estática pública; public static
    void PrintField() {

        Console.WriteLine(campo + ":" + typeof(T).Nome);
    }
}

...
TypeWithField<int>.field = "Primeiro";
TypeWithField<string>.field = "Segundo";
TypeWithField<DateTime>.field = "Terceiro";

TypeWithField<int>.PrintField();
TypeWithField<string>.PrintField();
TypeWithField<DateTime>.PrintField();
```

Você define o valor de cada campo para um valor diferente e depois imprime cada campo junto com o nome do argumento de tipo usado para esse tipo fechado. Aqui está o resultado da listagem 3.8:

```
Primeiro: Int32
Segundo: corda
Terceiro: DataHora
```

A regra básica é um campo estático por tipo fechado. O mesmo se aplica a inicializadores estáticos e construtores estáticos. Mas é possível ter um tipo genérico aninhado em outro e tipos com vários parâmetros genéricos. Isso parece muito mais complicado, mas funciona como você provavelmente acha que deveria. A listagem a seguir mostra isso em ação, desta vez usando construtores estáticos para mostrar quantos tipos existem.

---

<sup>6</sup> Bem, há um por domínio de aplicativo. Para os propósitos desta seção, presumiremos que estamos lidando apenas com um domínio de aplicativo. Os conceitos para diferentes domínios de aplicação funcionam da mesma forma com tipos genéricos e não genéricos. Variáveis decoradas com [ThreadStatic] também violam esta regra.

**Listagem 3.9 Construtores estáticos com tipos genéricos aninhados**

```
classe pública Externa<T> {  
  
    classe pública Interna<U,V> {  
  
        interno estático() {  
  
            Console.WriteLine("Externo<{0}>.Inner<{1},{2}> ",  
                typeof(T).Nome,  
                typeof(U).Name,  
                typeof(V).Name);  
  
        } public static void DummyMethod() {}  
    }  
}  
...  
Outer<int>.Inner<string,DateTime>.DummyMethod();  
Outer<string>.Inner<int,int>.DummyMethod();  
Exterior<objeto>.Inner<string,objeto>.DummyMethod();  
Exterior<string>.Inner<string,objeto>.DummyMethod();  
Exterior<objeto>.Inner<objeto,string>.DummyMethod();  
Outer<string>.Inner<int,int>.DummyMethod();
```

A primeira chamada para `DummyMethod()` para qualquer tipo fará com que o tipo seja inicializado, momento em que o construtor estático imprime alguns diagnósticos. Cada lista diferente de argumentos de tipo conta como um tipo fechado diferente, então a saída da listagem 3.9 é semelhante a esta:

```
Outer<Int32>.Inner<String,DateTime>  
Outer<String>.Inner<Int32,Int32>  
Outer<Object>.Inner<String,Object>  
Outer<String>.Inner<String,Object> Outer<Object>.Inner  
<Objeto, String>
```

Assim como acontece com os tipos não genéricos, o construtor estático para qualquer tipo fechado só é executado uma vez, e é por isso que a última linha da listagem 3.9 não cria uma sexta linha de saída – o construtor estático para `Outer<string> . Inner<int,int>` executado anteriormente, produzindo a segunda linha de saída.

Para esclarecer qualquer dúvida, se você tivesse uma classe `PlainInner` não genérica dentro de `Outer`, ainda haveria um tipo `Outer<T>.PlainInner` possível por tipo `Outer` fechado , então `Outer<int>.PlainInner` seria separado de `Outer<long >.PlainInner`, com um conjunto separado de campos estáticos, como visto anteriormente.

Agora que você viu o que constitui um tipo diferente, devemos ver quais podem ser os efeitos disso em termos da quantidade de código nativo gerado. E não, não é tão ruim quanto você imagina...

### 3.4.2 Como o compilador JIT lida com genéricos

Dado que temos todos esses diferentes tipos fechados, o trabalho do JIT é converter o IL do tipo genérico em código nativo para que ele possa realmente ser executado. De certa forma, você

não deveria se importar exatamente como isso acontece - além de ficar de olho na memória e no tempo de CPU , você não veria muita diferença se o JIT adotasse a abordagem mais simples possível e gerasse código nativo para cada tipo fechado separadamente, como se cada um não tinha nada a ver com qualquer outro tipo. Mas os autores do JIT são inteligentes o suficiente para que valha a pena analisar o que fizeram.

Vamos começar primeiro com uma situação simples, com um único parâmetro de tipo — usaremos `List<T>` por uma questão de conveniência. O JIT cria um código diferente para cada tipo fechado com um argumento de tipo que é um tipo de valor – `int`, `long`, `Guid` e similares. Mas ele compartilha o código nativo gerado para todos os tipos fechados que usam um tipo de referência como argumento de tipo, como `string`, `Stream` e `StringBuilder`. Isso pode ser feito porque todas as referências têm o mesmo tamanho (o tamanho varia entre um CLR de 32 bits e um CLR de 64 bits , mas dentro de qualquer CLR todas as referências têm o mesmo tamanho). Uma matriz de referências sempre terá o mesmo tamanho, quaisquer que sejam as referências. O espaço necessário na pilha para uma referência será sempre o mesmo. O JIT pode usar as mesmas otimizações para armazenar referências em registros, independentemente do tipo – o `List<Reason>` continua.

Cada um dos tipos ainda possui seus próprios campos estáticos, conforme descrito na seção 3.4.1, mas o próprio código executável é reutilizado. É claro que o JIT faz tudo isso preguiçosamente — ele não gera o código para `List<int>` antes do necessário e armazena esse código em cache para todos os usos futuros de `List<int>`.

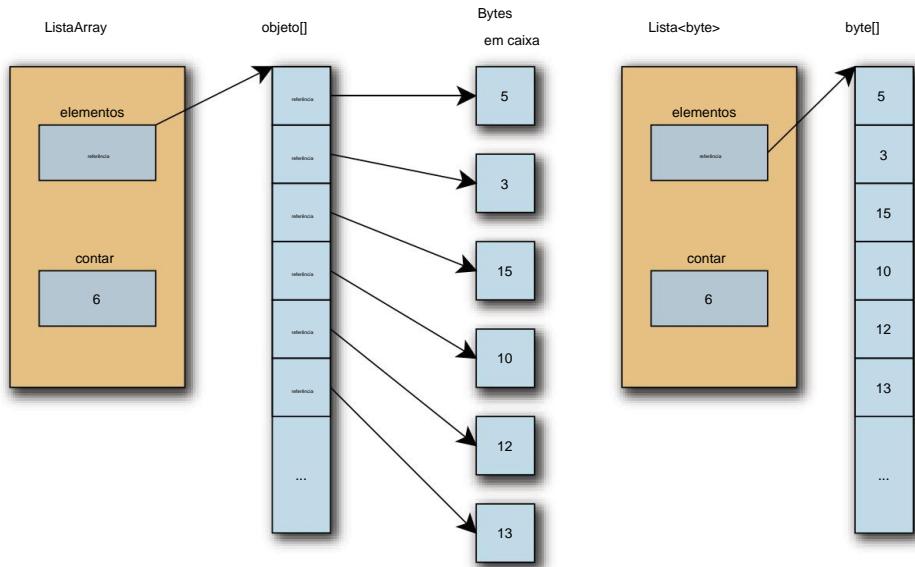
Em teoria, é possível compartilhar código para pelo menos *alguns* tipos de valores. O JIT teria que ser cuidadoso, não apenas devido ao tamanho, mas também por motivos de coleta de lixo – teria que ser capaz de identificar rapidamente áreas de um valor struct que são referências ativas. Mas os tipos de valor que têm o mesmo tamanho e ocupam o mesmo espaço na memória no que diz respeito ao coletor de lixo *podem* compartilhar código. No momento em que este livro foi escrito, isso era de prioridade suficientemente baixa e não foi implementado, e pode muito bem continuar assim.

Esse nível de detalhe é principalmente de interesse acadêmico, mas tem um leve impacto no desempenho em termos de mais código sendo compilado em JIT . Os benefícios de desempenho dos genéricos podem ser enormes e, novamente, isso se resume a ter a oportunidade de compilar códigos diferentes para tipos diferentes. Considere um `List<byte>`, por exemplo. No .NET 1.1, adicionar bytes individuais a um `ArrayList` significaria colocar cada um deles em caixa e armazenar uma referência para cada valor em caixa. Usar `List<byte>` não tem esse impacto – `List<T>` tem um membro do tipo `T[]` para substituir o `object[]` dentro de `ArrayList`, e esse array é do tipo apropriado, ocupando o espaço apropriado. `List<byte>` possui um `byte[]` direto usado para armazenar os elementos do array. (De muitas maneiras, isso faz com que `List<byte>` se comporte como um `MemoryStream`.)

A Figura 3.3 mostra um `ArrayList` e um `List<byte>`, cada um com os mesmos seis valores. As próprias matrizes possuem mais de seis elementos, para permitir o crescimento. Tanto `List<T>` quanto `ArrayList` possuem um buffer e criam um buffer maior quando necessário.

A diferença de eficiência aqui é incrível. Vejamos primeiro o `ArrayList` . Cada um dos bytes em caixa considerando um CLR de 32 bits .<sup>7</sup> ocupará 8 bytes de sobrecarga do objeto,

<sup>7</sup> Ao executar em um CLR de 64 bits, as despesas gerais são maiores.



**Figura 3.3 Demonstraçao visual de por que `List<T>` ocupa muito menos espaço que `ArrayList` ao armazenar tipos de valor**

mais 4 bytes (1 byte, arredondado para um limite de palavra) para os próprios dados. Além disso, você tem todas as referências, cada uma ocupando 4 bytes. Portanto, para cada byte de dados úteis, você paga pelo menos 16 bytes – e ainda há espaço extra não utilizado para referências no buffer.

Compare isso com `List<byte>`. Cada byte na lista ocupa um único byte na matriz de elementos. Ainda há espaço desperdiçado no buffer, esperando para ser usado por novos itens, mas pelo menos você está desperdiçando apenas um único byte por elemento não utilizado.

Você não apenas ganha espaço, mas também velocidade de execução. Você economiza o tempo necessário para alocar a caixa, para realizar a verificação de tipo envolvida na desembalagem dos bytes para chegar até eles e para coletar o lixo das caixas quando elas não são mais referenciadas.

No entanto, você não precisa descer ao nível CLR para descobrir que as coisas acontecem de forma transparente em seu nome. C# sempre facilitou a vida com atalhos sintáticos, e a próxima seção analisa um exemplo familiar, mas com um toque genérico: iterar com para cada.

### 3.4.3 Iteração genérica

Uma das operações mais comuns que você deseja realizar em uma coleção é iterar todos os seus elementos. Normalmente, a maneira mais simples de fazer isso é usar a instrução `foreach`. No C# 1, isso dependia da coleção implementar a interface `System.Collections.IEnumerable` ou ter um método `GetEnumerator()` semelhante que retornasse um tipo com um método `MoveNext()` adequado e uma propriedade `Current`. A propriedade `Current` não precisava ser do tipo `object`, e isso era tudo

ponto de ter estas regras adicionais, que à primeira vista parecem estranhas. Sim, mesmo em C# 1 você poderia evitar boxe e unboxing durante a iteração se tivesse um tipo de iteração personalizado.

O C# 2 torna isso um pouco mais fácil, pois as regras para a instrução foreach foram estendidas para usar também a interface System.Collections.Generic.IEnumerable<T> junto com seu parceiro, IEnumerator<T>. Estes são simplesmente os equivalentes genéricos das interfaces de iteração antigas e são usados preferencialmente às versões não genéricas. Isso significa que se você iterar através de uma coleção genérica de elementos de tipo de valor – List<int>, por exemplo – então nenhum boxing será executado. Se a interface antiga tivesse sido usada, você não teria incorrido no custo de boxe ao *armazenar* os elementos da lista, mas ainda assim acabaria encaixotando-os ao recuperá-los usando foreach.

Tudo isso é feito nos bastidores - tudo o que você precisa fazer é usar a instrução foreach da maneira normal, usando um tipo apropriado para a variável de iteração, e tudo ficará bem. Esse não é o fim da história, no entanto. Na situação relativamente rara em que você precisa implementar a iteração em um de seus próprios tipos, você descobrirá que IEnumerable<T> estende a antiga interface IEnumerable , o que significa que você precisa implementar dois métodos diferentes:

```
IEnumerator<T> GetEnumerator();
IEnumerator GetEnumerator();
```

Você pode ver o problema? Os métodos diferem apenas no tipo de retorno, e as regras de sobrecarga do C# impedem que você escreva dois desses métodos normalmente. Na seção 2.2.2, você viu uma situação semelhante e pode usar a mesma solução alternativa aqui. Se você implementar IEnumerable usando implementação de interface explícita, poderá implementar IEnumerable<T> com um método “normal”. Felizmente, como IEnumerator<T> estende IEnumerator, você pode usar o mesmo valor de retorno para ambos os métodos e implementar o método não genérico apenas chamando a versão genérica. Claro, agora você precisa implementar IEnumerator<T> e rapidamente se depara com problemas semelhantes, desta vez com a propriedade Current .

A listagem a seguir fornece um exemplo completo, implementando uma classe enumerável que sempre enumera os inteiros de 0 a 9.

#### Listagem 3.10 Um iterador genérico completo – dos números de 0 a 9

```
class CountingEnumerable: IEnumerable<int> {
    public IEnumerator<int> GetEnumerator() {
        return new CountingEnumerator();
    }
    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}
```

Implementos  
IEnumerator<T>  
**B implicitamente**

Implementos  
IEnumerator  
**C explicitamente**

```

class CountingEnumerator: IEnumerator<int> {

    corrente interna = -1;

    public bool MoveNext() {

        atual++; corrente
        de retorno < 10;
    }

    public int Atual { obter { retornar atual; } }

    objeto IEnumerator.Current { get { return Current; } }

    public void Redefinir() {

        corrente = -1;

    } public void Dispose() {}

}

...
Contador CountingEnumerable = new CountingEnumerable(); foreach (int x no contador) {

    Console.WriteLine(x);
}

```

**D** Implementa `IEnumerator<T>.Current` implicitamente

**E** Implementa `IEnumerator.Current` explicitamente

**F** Prova que o tipo enumerável funciona

É claro que esses resultados não são particularmente úteis, mas o código mostra os pequenos obstáculos que você precisa percorrer para implementar a iteração genérica de maneira adequada - pelo menos se você estiver fazendo tudo à mão. (E isso sem fazer um esforço para lançar exceções se o `Current` for acessado em um momento inapropriado.) Se você acha que a listagem 3.10 parece muito trabalhosa apenas para imprimir os números de 0 a 9, não posso ajudar, mas concordo com você, e haveria ainda mais código se você quisesse iterar sobre algo útil. Felizmente, você verá no capítulo 6 que o C# 2 tira grande parte do trabalho dos iteradores em muitos casos. Mostrei a versão completa aqui para que você possa apreciar as pequenas rugas que foram introduzidas pela decisão de design de `IEnumerable<T>` para estender `IEnumerable`. Não estou sugerindo que tenha sido uma decisão errada; permite passar qualquer `IEnumerable<T>` para um método escrito em C# 1 com um parâmetro `IEnumerable`. Isso não é tão importante agora como era em 2005, mas ainda é um caminho de transição útil.

Você só precisa do truque de usar a implementação explícita da interface duas vezes - uma vez para `IEnumerable.GetEnumerator` **C** e uma vez para `IEnumerator.Current` **E**. Ambos chamam seus equivalentes genéricos (**B** e **D**, respectivamente). Outra adição ao `IEnumerator<T>` é que ele estende `IDisposable`, então você precisa fornecer um método `Dispose`. A instrução `foreach` em C# 1 já chamou `Dispose` em um iterador se implementou `IDisposable`, mas em C# 2 nenhum teste de tempo de execução é necessário – se o compilador descobrir que você implementou `IEnumerable<T>`, ele cria uma chamada incondicional para `Dispose` no final do loop (em um bloco final). Muitos iteradores não precisarão realmente descartar nada, mas é bom saber que quando isso for necessário, a maneira mais comum de trabalhar com um iterador (a instrução `foreach` **F**) lida com o

lado chamador automaticamente. Isso é mais comumente usado para liberar recursos quando você termina a iteração. Por exemplo, você pode ter um iterador que lê linhas de um arquivo e precisa fechar o identificador do arquivo quando o código de chamada terminar o loop.

Passaremos agora da eficiência no tempo de compilação para a flexibilidade no tempo de execução: nosso último tópico avançado é a reflexão. Mesmo no .NET 1.0/1.1, a reflexão pode ser complicada, mas tipos e métodos genéricos introduzem um nível extra de complexidade. A estrutura fornece tudo o que você precisa (com um pouco de sintaxe útil do C# 2 como linguagem) e, embora as considerações adicionais possam ser assustadoras, não será tão ruim se você der um passo de cada vez.

### 3.4.4 Reflexão e genéricos

A reflexão é usada por pessoas diferentes para todos os tipos de coisas. Você pode usá-lo para introspecção de objetos em tempo de execução para executar uma forma simples de vinculação de dados. Você pode usá-lo para inspecionar um diretório cheio de assemblies para encontrar implementações de uma interface de plug-in. Você pode escrever um arquivo para uma estrutura de inversão de controle (consulte [www.martinfowler.com/articles/injection.html](http://www.martinfowler.com/articles/injection.html)) para carregar e configurar dinamicamente os componentes do seu aplicativo. Como os usos da reflexão são tão diversos, não vou me concentrar em nenhum deles em particular, mas, em vez disso, darei orientações mais gerais sobre a execução de tarefas comuns. Começaremos examinando as extensões do operador `typeof`.

#### USANDO TYPEOF COM TIPOS GENÉRICOS

A reflexão consiste em examinar objetos e seus tipos. Como tal, uma das coisas mais importantes que você precisa fazer é obter uma referência a um objeto `System.Type` específico, que permite acesso a todas as informações sobre esse tipo. C# usa o operador `typeof` para obter tal referência para tipos conhecidos em tempo de compilação, e isso foi estendido para abranger tipos genéricos.

Existem duas maneiras de usar `typeof` com tipos genéricos – uma recupera a *definição do tipo genérico* (em outras palavras, o tipo genérico não vinculado) e a outra recupera um tipo construído específico. Para obter a definição de tipo genérico – o tipo sem nenhum dos argumentos de tipo especificado – basta pegar o nome do tipo como ele teria sido declarado e remover os nomes dos parâmetros de tipo, mantendo todas as vírgulas. Para recuperar tipos construídos, você especifica os argumentos de tipo da mesma maneira que faria para declarar uma variável do tipo genérico. A próxima listagem dá um exemplo de ambos os usos.

Ele usa um método genérico para que possamos revisitar como `typeof` pode ser usado com um parâmetro de tipo, que vimos anteriormente em 3.8.

#### Listagem 3.11 Usando o operador `typeof` com parâmetros de tipo

```
static void DemonstrateTypeof<X>() {
    Console.WriteLine(typeof(X));
    Console.WriteLine(typeof(Lista<>));
    Console.WriteLine(typeof(Dicionário<,>));
    Console.WriteLine(typeof(Lista<X>));
```

 Exibe o parâmetro de tipo do método

 Exibe tipos genéricos

 B Exibe tipos fechados (apesar de usar o parâmetro type)

```

Console.WriteLine(typeof(Dicionário<string,X>));
Console.WriteLine(typeof(Lista<long>));
Console.WriteLine(typeof(Dictionary<long,Guid>));
}
...
DemonstrateTypeof<int>();

```

← Exibe tipos fechados

A maior parte da listagem 3.11 funciona como você poderia esperar, mas vale a pena ressaltar duas coisas. Primeiro, observe a sintaxe para obter a definição de tipo genérico de Dictionary<TKey,TValue>. A vírgula entre colchetes angulares é necessária para informar ao compilador para procurar o tipo com dois parâmetros de tipo; lembre-se que podem existir vários tipos genéricos com o mesmo nome, desde que variem de acordo com o número de parâmetros de tipo que possuem. Da mesma forma, você recuperaria a definição de tipo genérico para MyClass <T1,T2,T3,T4> usando typeof(MyClass<,,,>). O número de parâmetros de tipo é especificado em IL (e em nomes completos de tipos no que diz respeito à estrutura) colocando um sinal de crase após a primeira parte do nome do tipo e depois o número. Os parâmetros de tipo são então indicados entre colchetes em vez dos colchetes angulares aos quais estamos acostumados. Por exemplo, a segunda linha impressa termina com List`1[T], mostrando que existe um parâmetro de tipo, e a terceira linha inclui Dictionary`2[TKey,TValue].

Segundo, observe que sempre que o parâmetro de tipo do método (X) for usado, o valor real do argumento de tipo será usado em tempo de execução. Portanto, esta linha B imprime List`1[System.Int32] em vez de List`1[X], o que você poderia esperar.<sup>8</sup> Em outras palavras, um tipo que está aberto em tempo de compilação pode ser fechado em tempo de execução. *Isso é muito confuso. Você deve estar ciente disso caso não obtenha os resultados esperados, mas caso contrário, não se preocupe.* Para recuperar um tipo construído verdadeiramente aberto em tempo de execução, você precisa trabalhar um pouco mais. Consulte a documentação do MSDN para Type.IsGenericType para obter um exemplo adequadamente complicado (<http://mng.bz/9W6O>).

Para referência, aqui está o resultado da listagem 3.11:

```

System.Int32
System.Collections.Generic.List`1[T]
System.Collections.Generic.Dictionary`2[TKey,TValue]
System.Collections.Generic.List`1[System.Int32]
System.Collections.Generic.Dictionary`2[System.String,System.Int32]
System.Collections.Generic.List`1[System.Int64]
System.Collections.Generic.Dictionary`2[System.Int64,System.Guid]

```

Depois de recuperar um objeto que representa um tipo genérico, há muitas etapas seguintes que você pode seguir. Todos os disponíveis anteriormente (encontrar os membros do tipo, criar uma instância e assim por diante) ainda estão presentes - embora alguns não sejam aplicáveis para definições de tipo genérico - e há também novos que permitem perguntar sobre o genérico natureza do tipo.

---

<sup>8</sup> Eu deliberadamente contrariei a convenção de usar um parâmetro de tipo chamado T, precisamente para que pudéssemos saber a diferença entre o T na declaração List<T> e o X na declaração do nosso método.

### MÉTODOS E PROPRIEDADES DO TIPO DE SISTEMA

Existem muitos métodos e propriedades novos para examiná-los todos em detalhes, mas há dois particularmente importantes: `GetGenericTypeDefinition` e `MakeGenericType`. Eles são efetivamente opostos – o primeiro atua em um tipo construído, recuperando a definição genérica do tipo; o segundo atua em uma definição de tipo genérico e retorna um tipo construído. Provavelmente teria ficado mais claro se esse método tivesse sido chamado de `ConstructType`, `MakeConstructedType` ou algum outro nome com `construct` ou `construído` nele, mas estamos presos ao que temos.

Assim como os tipos normais, há apenas um objeto `Type` para qualquer tipo específico – portanto, chamar `MakeGenericType` duas vezes com os mesmos tipos dos argumentos retornará a mesma referência duas vezes. Da mesma forma, chamar `GetGenericTypeDefinition` em dois tipos construídos a partir da mesma definição de tipo genérico fornecerá o mesmo resultado para ambas as chamadas, mesmo que os tipos construídos sejam diferentes (como `List<int>` e `List<string>`).

Dois outros métodos que vale a pena explorar — desta vez métodos que já existiam no .NET 1.1 — são `Type.GetType(string)` e seu método `Assembly.GetType(string)` relacionado, ambos fornecendo um equivalente dinâmico para `typeof`. Você pode esperar poder alimentar cada linha da saída da listagem 3.11 para o método `GetType` chamado em um assembly apropriado, mas infelizmente a vida não é tão simples. É adequado para tipos construídos fechados — os argumentos de tipo ficam entre colchetes. Para definições de tipo genérico, porém, você precisa remover totalmente os colchetes - caso contrário, `GetType` pensará que você se refere a um tipo de array. A listagem a seguir mostra todos esses métodos em ação.

#### Listagem 3.12 Várias maneiras de recuperar objetos Type genéricos e construídos

```
string listTypeName = "System.Collections.Generic.List`1";
Digite defByName = Type.GetType(listTypeName);
Digite closedByName = Type.GetType(listTypeName + "[System.String]");
Digite closedByMethod = defByName.MakeGenericType(typeof(string));
Tipo fechadoByTypeof = typeof(List<string>);

Console.WriteLine(closedByMethod == closedByName);
Console.WriteLine(closedByName == closedByTypeof);

Digite defByTypeof = typeof(Lista<>); Digite
defByMethod = closedByName.GetGenericTypeDefinition();

Console.WriteLine(defByMethod == defByName);
Console.WriteLine(defByName == defByTypeof);
```

A saída da listagem 3.12 é True apenas quatro vezes, validando que, independentemente de como você obtém uma referência a um objeto de tipo específico, apenas um desses objetos está envolvido.

Como mencionei anteriormente, existem muitos métodos e propriedades novos em `Type`, como `GetGenericArguments`, `IsGenericTypeDefinition` e `IsGenericType`. Novamente, a documentação do `IsGenericType` é provavelmente o melhor ponto de partida para uma exploração mais aprofundada.

## REFLETINDO MÉTODOS GENÉRICOS

Os métodos genéricos possuem um conjunto semelhante (embora menor) de propriedades e métodos adicionais. A listagem a seguir dá uma breve demonstração disso, chamando um método genérico por reflexão.

### Listagem 3.13 Recuperando e invocando um método genérico com reflexão

```
public static void PrintTypeParameter<T>() {  
  
    Console.WriteLine(typeof(T));  
}  
...  
Tipo tipo = typeof(Snippet); Definição de  
MethodInfo = type.GetMethod("PrintTypeParameter"); MethodInfo construído =  
definição.MakeGenericMethod(typeof(string)); construído.Invoke(null, null);
```

Primeiro você recupera a definição do método genérico e, em seguida, cria um método genérico construído usando MakeGenericMethod. Assim como acontece com os tipos, você poderia seguir o outro caminho se quisesse, mas diferentemente de Type.GetType, não há como especificar um método construído na chamada GetMethod. A estrutura também tem um problema se os métodos forem sobrecarregados apenas pelo número de parâmetros de tipo - não há métodos em Type que permitam especificar o número de parâmetros de tipo, então, em vez disso, você teria que chamar Type.GetMethods e encontrar o certo examinando *todos* os métodos.

Após recuperar o método construído, você o invoca. Os argumentos neste exemplo são nulos, pois você está invocando um método estático que não possui nenhum parâmetro normal. A saída é System.String, como seria de esperar. Observe que os métodos recuperados de definições de tipos genéricos não podem ser invocados diretamente — em vez disso, você deve obter os métodos de um tipo construído. Isto se aplica a métodos genéricos e não genéricos.

**SALVO POR C# 4** Se tudo isso parece confuso para você, eu concordo. Felizmente, em muitos casos, a tipagem dinâmica do C# pode ajudar, eliminando muito do trabalho da reflexão genérica. Isso não ajuda em todas as situações, por isso vale a pena estar ciente do fluxo geral do código anterior, mas onde ele se *aplica* é ótimo.

Veremos a digitação dinâmica em detalhes no capítulo 14.

Novamente, mais métodos e propriedades estão disponíveis no MethodInfo, e IsGenericMethod é um bom ponto de partida no MSDN (<http://mng.bz/P36u>). Esperamos que as informações nesta seção tenham sido suficientes para você começar e apontar algumas das complexidades adicionais que você talvez não tenha previsto quando começou a acessar tipos e métodos genéricos com reflexão.

Isso é tudo que abordaremos em termos de recursos avançados. Apenas para reiterar, este capítulo não pretende ser um guia completo para genéricos, mas é improvável que a maioria dos desenvolvedores precise conhecer os detalhes mais obscuros. Espero, para seu bem, que você se enquadre nesse campo, pois as especificações tendem a ficar mais difíceis de ler quanto mais você se aprofunda nelas. Lembre-se de que, a menos que você esteja desenvolvendo sozinho e apenas para si mesmo, você estará

dificilmente será o único a trabalhar no seu código. Se você precisar de recursos mais complexos do que os demonstrados aqui, você deve assumir que qualquer pessoa que estiver lendo seu código precisará de ajuda para entendê-lo. Por outro lado, se você achar que seus colegas de trabalho não conhecem alguns dos tópicos que abordamos até agora, sinta-se à vontade para encaminhá-los para a livraria mais próxima...

Nossa seção principal final do capítulo analisa algumas das limitações dos genéricos em C# e considera recursos semelhantes em outras linguagens.

## 3.5 Limitações de genéricos em C# e outras linguagens

Não há dúvida de que os genéricos contribuem muito para o C# em termos de expressividade, segurança de tipo e desempenho. O recurso foi cuidadosamente projetado para lidar com a maioria das tarefas para as quais os programadores C++ normalmente usavam modelos, mas sem algumas das desvantagens que as acompanham. Mas isso não quer dizer que não existam limitações.

Existem alguns problemas que os modelos C++ resolvem com facilidade, mas que os genéricos do C# não podem ajudar. Da mesma forma, embora os genéricos em Java sejam geralmente menos poderosos do que em C#, existem alguns conceitos que podem ser expressos em Java, mas que não possuem um equivalente em C#. Esta seção mostrará alguns dos pontos fracos mais comumente encontrados e compararei brevemente a implementação de genéricos em C#/.NET com modelos C++ e genéricos Java.

É importante ressaltar que apontar esses obstáculos não significa que eles deveriam ter sido evitados em primeiro lugar. Em particular, não estou de forma alguma dizendo que poderia ter feito um trabalho melhor! Os projetistas de linguagem e plataforma tiveram que equilibrar o poder com a complexidade (e a pequena questão de conseguir tanto o design quanto a implementação dentro de uma escala de tempo razoável). Provavelmente, você não encontrará problemas e, se encontrar, poderá contorná-los com as orientações fornecidas aqui.

Começaremos com a resposta a uma pergunta que quase todo mundo levanta, mais cedo ou mais tarde:  
Por que não consigo converter um `List<string>` em um `List<object>?`

### 3.5.1 Falta de variação genérica

Na seção 2.2.2, examinamos a *covariância* de arrays – o fato de que um array de um tipo de referência pode ser visto como um array de seu tipo base ou como um array de qualquer uma das interfaces que ele implementa. Na verdade, existem duas formas dessa ideia, chamadas *covariância* e *contravariância*, ou coletivamente apenas *variância*. Os genéricos não suportam isso – eles são *invariáveis*.

Isso ocorre por uma questão de segurança de tipo, como você verá, mas pode ser irritante.

Uma coisa que gostaria de deixar claro para começar: C# 4 melhora um pouco a situação de variação genérica. Muitas das restrições listadas aqui *ainda* se aplicam, e esta seção serve como uma introdução útil à ideia de variação. Veremos como o C# 4 ajuda no capítulo 13, mas muitos dos exemplos mais claros de variação genérica dependem de outros novos recursos do C# 3, incluindo o LINQ. A variação também é um tópico bastante complicado por si só, então vale a pena esperar até que você se sinta confortável com o restante do C# 2 e 3 antes de abordá-lo. Para facilitar a leitura, não vou apontar todos os lugares desta seção que são ligeiramente diferentes no C# 4...tudo ficará claro no capítulo 13.

**POR QUE OS GENÉRICOS NÃO APOIAM A COVARIÂNCIA?**

Suponha que você tenha duas classes, Turtle e Cat, ambas derivadas de uma classe abstrata Animal . No código a seguir, o código da matriz (primeiro bloco) é válido em C# 2; o código genérico (segundo bloco) não é.

Válido (em tempo de compilação)	Inválido
Animal[] animais = novo Gato[5]; animais[0] = new Tartaruga();	List<Animal> animais = new List<Gato>(); animais.Add(new Tartaruga());

O compilador não tem problemas com a segunda linha em nenhum dos casos, mas a primeira linha em *Invalid* causa o seguinte erro:

erro CS0029: Não é possível converter implicitamente o tipo  
 'System.Collections.Generic.List<Cat>' para  
 'System.Collections.Generic.List<Animal>'

Esta foi uma escolha deliberada por parte dos designers da estrutura e da linguagem.

A pergunta óbvia a ser feita é *por que* isso é proibido, e a resposta está na segunda linha.

Não há nada na segunda linha que deva levantar qualquer suspeita. Afinal, List<Animal> efetivamente possui um método com a assinatura void Add(Animal value) — você deve ser capaz de colocar uma Tartaruga em qualquer lista de animais, por exemplo. Mas o objeto *real* referido por animais é um Cat[] (no código em *Válido*) ou um List<Cat> (em *Inválido*), ambos os quais exigem que apenas referências a instâncias de Cat (ou outras subclasses) sejam armazenados neles. Embora a versão do array seja compilada, ela falhará no tempo de execução. Isso foi considerado pelos projetistas de genéricos pior do que falhar em tempo de compilação, o que é razoável - o objetivo da digitação estática é descobrir erros antes que o código seja executado.

**POR QUE OS ARRAYS SÃO COVARIANTES?** Tendo respondido à pergunta sobre por que os genéricos são invariantes, o próximo passo óbvio é questionar por que os arrays são covariantes. De acordo com o *Common Language Infrastructure Annotated Standard* (Miller e Ragsdale, Addison-Wesley Professional, 2003), para a primeira versão do .NET os designers queriam atingir um público tão amplo quanto possível, o que incluía ser capaz de executar código compilado de Java fonte. Em outras palavras, o .NET possui matrizes covariantes porque Java possui matrizes covariantes - apesar de isso ser uma verruga conhecida em Java.

Então, é por isso que as coisas são como são – mas por que você deveria se importar e como você pode contornar a restrição?

**ONDE A COVARIÂNCIA SERIA ÚTIL**

O exemplo que dei com uma lista é claramente problemático. Você pode adicionar itens à lista, que é onde você perde a segurança de tipo neste caso, e uma operação add é um exemplo de um valor sendo usado como entrada na API: o chamador está fornecendo o valor.

O que aconteceria se você se limitasse a divulgar valores?

Os exemplos óbvios disso são `IEnumerable<T>` e (por associação) `IEnumerable<T>`. Na verdade, estes são quase os exemplos *canônicos* de covariância genérica. Juntos, eles descrevem uma sequência de valores – tudo o que você sabe sobre os valores que vê é que cada um deles será compatível com `T`, de modo que você sempre poderá escrever

```
T currentValue = iterador.Current;
```

Isso usa a ideia normal de compatibilidade – seria bom para um `IEnumerator <Animal>` produzir referências a instâncias de `Cat` ou `Turtle`, por exemplo. Não há como enviar valores inadequados para o tipo de sequência real, então você gostaria de poder tratar um `IEnumerator<Cat>` como um `IEnumerator<Animal>`. Vamos con-

Considere um exemplo de onde isso pode ser útil.

Suponha que você tome o exemplo de forma habitual para herança, mas usando uma interface (`IShape`). Agora considere outra interface, `IDrawing`, que representa um desenho feito de formas. Você terá dois tipos concretos de desenho – um `MondrianDrawing` (feito de retângulos) e um `SeuratDrawing` (feito de círculos).<sup>9</sup> A Figura 3.4 mostra as hierarquias de classes envolvidas.

Ambos os tipos de desenho precisam implementar a interface `IDrawing`, então eles precisam exponha uma propriedade com esta assinatura:

```
IEnumerable<IShape> Formas { get; }
```

Mas cada tipo de desenho provavelmente acharia mais fácil manter internamente uma lista com tipagem mais forte. Por exemplo, um desenho Seurat pode incluir um campo do tipo `List<Circle>`. É útil ter this em vez de `List<IShape>` para que, se precisar manipular os círculos de uma maneira específica do círculo, possa fazê-lo sem lançar. Se você tivesse um `List<IShape>`, poderia retorná-lo diretamente ou pelo menos envolvê-lo em um `ReadOnlyCollection<IShape>` para evitar que os chamadores mexam nele por meio de conversão – a implementação da propriedade seria barata e simples de qualquer maneira. Mas você não pode fazer isso quando seus tipos não correspondem. Você não pode converter de `IEnumerable<Circle>` para `IEnumerable<IShape>`. Então, o que você *pode* fazer?

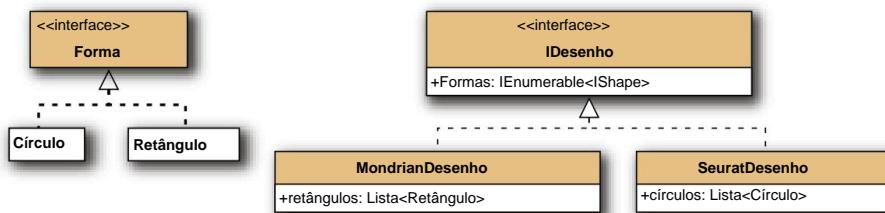


Figura 3.4 Interfaces para formas e desenhos e duas implementações de cada

<sup>9</sup> Se esses nomes não significam nada para você, verifique as entradas dos artistas na Wikipédia ([http://en.wikipedia.org/wiki/Piet\\_Mondrian](http://en.wikipedia.org/wiki/Piet_Mondrian) e [http://en.wikipedia.org/wiki/Georges-Pierre\\_Seurat](http://en.wikipedia.org/wiki/Georges-Pierre_Seurat)). Eles têm significados especiais para mim por diferentes razões: Mondrian também é o nome de uma ferramenta de revisão de código que usamos no Google, e Seurat é o homônimo George de *Sunday in the Park with George* – um musical maravilhoso de Stephen Sondheim.

Existem algumas opções aqui:

- ÿ Mude o tipo de campo para `List<IShape>` e apenas viva com as conversões. Isso não é agradável e praticamente anula o uso de genéricos.
  - ÿ Use os novos recursos fornecidos pelo C# 2 para implementar iteradores, como você verá no capítulo 6. Esta é uma solução razoável para este caso específico, mas apenas *neste* caso (onde você está lidando com `IEnumerable<T>`). ÿ
- Faça com que cada implementação da propriedade `Shapes` crie uma nova cópia da lista, possivelmente usando `List<T>.ConvertAll` para simplificar. Criar uma cópia independente de uma coleção geralmente é a coisa certa a se fazer em uma API, mas causa muitas cópias, o que pode ser desnecessariamente ineficiente em muitos casos.
- ÿ Torne o `IDrawing` genérico, indicando o tipo de formas no desenho. Assim, `MondrianDrawing` implementaria `IDrawing<Rectangle>` e `SeuratDrawing` implementaria `IDrawing<Circle>`. Isso só é viável quando você possui a interface.

- ÿ Crie uma classe auxiliar para adaptar um tipo de `IEnumerable<T>` em outro:

```
classe EnumerableWrapper<TOriginal, TWrapper> : IEnumerable<TWrapper>
    onde TOriginal: TWrapper
```

Novamente, como esta situação específica (`IEnumerable<T>`) é especial, você pode usar apenas um método utilitário. Na verdade, o .NET 3.5 vem com dois métodos úteis como este: `Enumerable.Cast<T>` e `Enumerable.OfType<T>`. Eles fazem parte do LINQ e os veremos no capítulo 11. Embora este seja um caso especial, é provavelmente a forma mais comum de covariância genérica que você encontrará.

Ao se deparar com problemas de covariância, talvez seja necessário considerar todas essas opções e qualquer outra coisa que possa imaginar. Depende muito da natureza exata da situação. Infelizmente, a covariância não é o único problema com o qual você precisa lidar. Há também a questão da *contravariância*, que é como a covariância ao contrário.

#### ONDE A CONTRAVARIÂNCIA SERIA ÚTIL

A contravariância parece um pouco menos intuitiva do que a covariância, mas faz sentido. Com covariância, você estava tentando converter de `SomeType<Circle>` para `SomeType<IShape>` (usando `IEnumerable<T>` para `SomeType` no exemplo anterior). A contravariância trata da conversão de outra maneira – de `SomeType<IShape>` para `SomeType <Circle>`. Como isso pode ser seguro? Bem, a covariância é segura quando `SomeType` descreve apenas operações que *retornam* o parâmetro de tipo - e a contravariância é segura quando `SomeType` descreve apenas operações que *aceitam* o parâmetro de tipo.<sup>10</sup>

O exemplo mais simples de um tipo que usa apenas seu parâmetro de tipo em uma posição de entrada é `IComparer<T>`, que é comumente usado para classificar coleções. Vamos expandir a interface `IShape` (que estava vazia até agora) para incluir uma propriedade `Area`. Agora é fácil escrever uma implementação de `IComparer<IShape>` que compare duas formas quaisquer por área. Você *gostaria* então de poder escrever o seguinte código:

---

<sup>10</sup> Você verá no capítulo 13 que há um pouco mais do que isso, mas esse é o princípio geral.

```
IComparer<IShape> areaComparer = new AreaComparer(); List<Círculo> círculos =
new List<Círculo>(); círculos.Adicionar(novo Círculo(Ponto.Vazio, 20));
círculos.Adicionar(novo Círculo(Ponto.Vazio, 10));
círculos.Sort(areaComparer);
```

INVÁLIDO →

Porém, isso não funcionará, porque o método Sort em List<Circle> efetivamente usa um IComparer<Circle>. O fato de AreaComparer poder comparar *qualquer* forma em vez de apenas círculos não impressiona em nada o compilador. Ele considera IComparer <Circle> e IComparer<IShape> tipos completamente diferentes. Enlouquecedor, não é? Seria bom se o método Sort tivesse esta assinatura:

```
void Sort<S>(IComparer<S> comparador) onde T : S
```

Infelizmente, isso *não* é apenas a assinatura de Sort, mas também *não pode ser* - a restrição é inválida, porque é uma restrição em T em vez de S. Você deseja uma restrição de tipo de conversão, mas na outra direção, restringindo o S esteja em algum lugar *acima* da árvore de herança de T , em vez de *abaixo*.

Dado que isso *não* é possível, o que você *pode* fazer? Existem menos opções desta vez. Primeiro, você poderia revisitar a ideia de criar uma classe auxiliar genérica, como segue.

#### Listagem 3.14 Contornando a falta de contravariância com um ajudante

```
class ComparisonHelper<TBase, TDerived> : IComparer<TDerived>
    onde TDerivado: TBase
{
    comparador privado somente leitura IComparer<TBase>;
    public ComparisonHelper(IComparer<TBase> comparador) {
        this.comparador = comparador;
    }

    public int Comparar(TDerived x, TDerived y) {
        retornar comparador.Compare(x, y);
    }
}
```

Este é um exemplo do padrão de adaptador em funcionamento, embora, em vez de adaptar uma interface para outra completamente diferente, você esteja apenas adaptando de IComparer<TBase> para IComparer<TDerived>. Você apenas se lembra do comparador original que fornece a lógica real para comparar itens do tipo base C e, em seguida, chama-o quando for solicitado a comparar itens do tipo derivado D. O fato de que nenhuma conversão está envolvida (nem mesmo as ocultas) deve dar-lhe alguma confiança: este auxiliar é totalmente seguro para digitação.

Você pode chamar o comparador base devido a uma conversão implícita disponível de TDerived para TBase, necessária com uma restrição de tipo B.

A segunda opção é tornar genérica a classe de comparação de áreas com uma restrição de tipo de conversão, para que ela possa comparar quaisquer dois valores do mesmo tipo, desde que esse tipo implemente IShape. Por uma questão de simplicidade na situação em que você realmente não

precisar dessa funcionalidade, você poderia manter a classe não genérica apenas fazendo-a derivar do genérico:

```
classe AreaComparer<T> : IComparer<T> onde T : IShape
```

```
classe AreaComparer: AreaComparer<IShape>
```

Claro, você só pode fazer isso quando puder alterar a classe de comparação. Esse pode ser uma solução eficaz, mas ainda parece pouco natural - por que você deveria construir o comparador de várias maneiras para diferentes tipos quando ele não vai se comportar? de forma diferente? Por que você deveria derivar da classe para simplificar as coisas quando você não está realmente especializando o comportamento?

Observe que as várias opções para covariância e contravariância usam mais genéricos e restrições para expressar a interface de uma maneira mais geral ou para fornecer classes auxiliares genéricas. Eu sei que adicionar uma restrição faz com que pareça *menos* geral, mas a generalidade é adicionada tornando-se primeiro o tipo ou método genérico. Quando você Se você se deparar com um problema como esse, adicionar um nível de genericidade em algum lugar com uma restrição apropriada deve ser a primeira opção a considerar. Métodos genéricos (em vez de tipos genéricos) são frequentemente úteis aqui, pois a inferência de tipo pode tornar a falta de variação invisível a olho nu. Isso é particularmente verdadeiro em C# 3, que possui tipos mais fortes capacidades de inferência do que C# 2.

Essa limitação é uma causa *muito* comum de perguntas em sites de discussão sobre C#. O as questões restantes são relativamente acadêmicas ou afetam apenas um subconjunto moderado da comunidade de desenvolvimento. O próximo afeta principalmente aqueles que fazem muitos cálculos (geralmente científicos ou financeiros) em seu trabalho.

### 3.5.2 Falta de restrições de operador ou restrição “numérica”

C# tem suas desvantagens quando se trata de código fortemente matemático. A necessidade usar explicitamente a classe Math para cada operação além da aritmética mais simples e A falta de typecasts no estilo C para permitir que a representação de dados usada em um programa seja facilmente alterada sempre foi levantada pela comunidade científica como barreiras à adoção do C#. Os genéricos provavelmente não resolveriam totalmente nenhum desses problemas, mas há um problema comum que impede os genéricos de ajudar tanto quanto poderiam ter.

Considere este método genérico (illegal):

```
ímpar T FindMean<T>(dados Inumerable<T>
{
    Soma T = padrão(T);
    contagem interna = 0;
    foreach (dado T em dados)
    {
        soma += dado;
        contar++;
    }
    retornar soma/contagem;
}
```



Obviamente, isso nunca funcionaria para *todos* os tipos de dados – o que significaria adicionar um Exceção para outra, por exemplo? É evidente que algum tipo de restrição é necessária... algo que pode expressar o que você precisa fazer: adicionar duas instâncias de T juntos e divida um T por um número inteiro. Se isso estivesse disponível, mesmo que limitado a tipos internos, você poderia escrever algoritmos genéricos que não se importariam se eles estavam trabalhando em um int, um long, um double, um decimal e assim por diante.

Limitá-lo aos tipos integrados teria sido decepcionante, mas melhor que nada. A solução ideal também teria que permitir que tipos definidos pelo usuário atuassem de forma numérica. capacidade, então você pode definir um tipo Complex para lidar com números complexos, por instância.<sup>11</sup> Esse número complexo poderia então armazenar cada um de seus componentes em um genérico também, então você poderia ter um Complex<float>, um Complex<double> e assim por diante.

Duas soluções relacionadas (mas hipotéticas) apresentam-se. Uma seria permitir restrições em operadores, então você poderia escrever um conjunto de restrições como estas (atualmente inválidos):

onde T: operador T+ (T, T), operador T/ (T, int)

Isso exigiria que T tivesse as operações necessárias no código anterior. O outro A solução seria definir alguns operadores e talvez conversões que devem ser suportadas para que um tipo atenda à restrição extra - você poderia torná-lo o “restrição numérica” escrita onde T: numérica.

Um problema com ambas as opções é que elas não podem ser expressas como normais interfaces, porque a sobrecarga do operador é realizada com membros *estáticos*, que não pode ser usado para implementar interfaces. Acho atraente a ideia de *interfaces estáticas*: interfaces que declaram apenas membros estáticos, incluindo métodos, operadores e construtores. Essas interfaces estáticas só seriam úteis dentro de restrições de tipo, mas eles apresentariam uma maneira genérica com segurança de tipo para acessar membros estáticos. Isto é apenas céu azul pensando, no entanto (veja minha postagem no blog sobre o assunto para mais detalhes: <http://mng.bz/3Rk3>). Não conheço nenhum plano para incluir isso em uma versão futura do C#.

As duas soluções alternativas mais legais para esse problema até o momento exigem versões posteriores do .NET: um projetado por Marc Gravell (<http://mng.bz/9m8i>) usa árvores de expressão (que você conhecerá no capítulo 9) para construir métodos dinâmicos; o outro usa o recursos dinâmicos do C# 4. Você verá um exemplo deste último no capítulo 14. Mas, ao Você pode perceber pelas descrições que ambos são dinâmicos — você precisa esperar até o momento da execução para ver se seu código funcionará com um tipo específico. Existem algumas soluções alternativas que ainda usam digitação estática, mas têm outras desvantagens (surpreendentemente, às vezes podem ser mais lentas que o código dinâmico).

As duas limitações que examinamos até agora foram bastante práticas – elas foram problemas que você pode encontrar durante o desenvolvimento real. Mas se você é curioso como eu, também pode estar se perguntando sobre outras limitações que não necessariamente retardam o desenvolvimento, mas são curiosidades intelectuais. Em particular, por que são genéricos limitados a tipos e métodos?

---

<sup>11</sup> Isso pressupõe que você não esteja usando o .NET 4 ou superior, é claro, porque então você poderia usar o System.Numerics.Complexo.

### 3.5.3 Falta de propriedades genéricas, indexadores e outros tipos de membros

Vimos tipos genéricos (classes, estruturas, delegados e interfaces) e genéricos métodos. Existem muitos outros membros que *poderiam* ser parametrizados, mas não há propriedades genéricas, indexadores, operadores, construtores, finalizadores ou eventos.

Primeiro, sejamos precisos sobre o que queremos dizer aqui: claramente um indexador pode ter um tipo de retorno que é um parâmetro de tipo – `List<T>` é um exemplo óbvio. `KeyValuePair< TKey, TValue >` fornece exemplos semelhantes para propriedades. O que você *não pode* ter é um indexador ou propriedade (ou qualquer outro membro dessa lista) com tipo extra parâmetros.

Deixando de lado a possível sintaxe de declaração por um minuto, vamos ver como esses membros podem ter que ser chamados:



```
SomeClass<string> instância = new SomeClass<string><Guid>("x");
int x = instância.SomeProperty<int>;
byte y = instance.SomeIndexer<byte>["chave"];
instância.Click<byte> += ByteHandler;
instância = instância +<int> instância;
```

Espero que você concorde que tudo isso parece um tanto bobo. Os finalizadores nem podem ser chamados explicitamente do código C#, e é por isso que não há uma linha para eles. O fato de você não posso fazer nada disso não vai causar problemas significativos em lugar nenhum, tanto quanto posso veja - vale a pena estar ciente disso como uma limitação acadêmica.

O membro onde esta restrição é mais irritante é provavelmente o construtor. A método genérico estático na classe é uma boa solução alternativa para isso, e o exemplo de sintaxe do construtor genérico mostrado anteriormente com duas listas de argumentos de tipo é horrível.

Essas não são de forma alguma as *únicas* limitações dos genéricos do C#, mas acredito que sejam as aqueles que você provavelmente encontrará, seja em seu trabalho diário, na comunidade conversas ou ao considerar ociosamente o recurso como um todo. Nas próximas duas seções, veremos como alguns aspectos destes não são problemas nas outras duas línguas. cujos recursos são mais comumente comparados com os genéricos do C#: C++ (com templates) e Java (com genéricos, a partir do Java 5). Abordaremos primeiro o C++.

### 3.5.4 Comparação com modelos C++

Os modelos C++ são um pouco como macros levadas a um nível extremo. Eles são incrivelmente poderosos, mas há custos associados a eles, tanto em termos de excesso de código quanto de facilidade de uso. entendimento.

Quando um modelo é usado em C++, o código é compilado para aquele conjunto específico de argumentos do modelo, como se os argumentos do modelo estivessem no código-fonte. Isso significa que não há tanta necessidade de restrições, porque o compilador verificará o que você tem permissão para fazer isso com o tipo enquanto compila o código para este conjunto específico de argumentos de modelo. O comitê de padrões C++ reconheceu que as restrições ainda são úteis, no entanto. As restrições foram incluídas e depois removidas do C++ 11 (o versão mais recente do C++), mas ainda podem ver a luz do dia, sob o nome de *conceitos*.

O compilador C++ é inteligente o suficiente para compilar o código apenas uma vez para qualquer conjunto de argumentos de modelo, mas não é capaz de compartilhar código da maneira que o CLR faz com tipos de referência. Essa falta de compartilhamento tem seus benefícios - ela permite otimizações específicas de tipo, como chamadas de método inlining para alguns parâmetros de tipo, mas não outros, do mesmo modelo. Isso também significa que a resolução de sobrecarga pode ser realizada separadamente para cada conjunto de parâmetros de tipo, em vez de apenas uma vez com base apenas no conhecimento limitado que o compilador C# possui devido a quaisquer restrições presentes.

Não se esqueça que com C++ normal há apenas uma compilação envolvida, em vez disso do que o modelo "compilar para IL" e depois "compilar JIT para código nativo" do .NET. A programa usando um modelo padrão de 10 maneiras diferentes incluirá o código 10 vezes em um programa C++. Um programa semelhante em C# usando um tipo genérico do framework de 10 maneiras diferentes não incluirá o código do tipo genérico - fará referência a ele, e o JIT compilará quantas versões diferentes forem necessárias (conforme descrito em seção 3.4.2) em tempo de execução.

Um recurso significativo que os modelos C++ têm em relação aos genéricos C# é que os argumentos do modelo não precisam ser nomes de tipos. Nomes de variáveis, nomes de funções e expressões constantes também podem ser usadas. Um exemplo comum disso é um tipo de buffer que tem o tamanho do buffer como um dos argumentos do modelo - um buffer <int,20> sempre será um buffer de 20 números inteiros, e um buffer<double,35> sempre será um buffer de 35 duplas. Essa habilidade é crucial para a metaprogramação de modelos (veja a Wikipedia artigo, [http://en.wikipedia.org/wiki/Template\\_metaprogramming](http://en.wikipedia.org/wiki/Template_metaprogramming)), que é uma técnica avançada de C++, cuja própria ideia me assusta, mas que pode ser poderosa em nas mãos de especialistas.

Os modelos C++ também são mais flexíveis de outras maneiras. Eles não sofrem com a falta das restrições do operador descritas na seção 3.5.2, e existem algumas outras restrições que não existem em C++: você pode derivar uma classe de um de seus parâmetros de tipo e pode especializar um modelo para um conjunto específico de argumentos de tipo. Esta última habilidade permite o autor do modelo para escrever código geral para ser usado quando não houver mais conhecimento código disponível e específico (geralmente altamente otimizado) para tipos específicos.

Os mesmos problemas de variação dos genéricos .NET também existem nos modelos C++. Um exemplo dado por Bjarne Stroustrup (o inventor do C++) é que não há conversões implícitas entre vector<shape\*> e vector<circle\*> com raciocínio semelhante - em neste caso, pode permitir que você coloque uma estaca quadrada em um buraco redondo.

Para obter mais detalhes sobre modelos C++, recomendo *The C++ Programming Language*, 3<sup>a</sup> edição de Stroustrup (Addison-Wesley Professional, 1997). Nem sempre é o livro mais fácil de seguir, mas o capítulo de modelos é bastante claro (depois de obter seu mente em torno da terminologia e sintaxe C++). Para obter mais comparações com os genéricos do .NET , consulte a postagem do blog da equipe do Visual C++ sobre este tópico (<http://mng.bz/En13>).

A outra linguagem óbvia para comparar com C# em termos genéricos é Java, que introduziu o recurso na linguagem principal para a versão 1.512 vários anos depois de outros projetos terem criado linguagens semelhantes a Java que suportavam genéricos.

---

<sup>12</sup> Ou 5.0, dependendo do sistema de numeração usado. Não me faça começar.

### 3.5.5 Comparação com genéricos Java

Enquanto o C++ inclui *mais* do modelo no código gerado do que o C#, o Java inclui *menos*. Na verdade, o tempo de execução Java não conhece nada de genéricos. O bytecode Java (aproximadamente equivalente a IL) para um tipo genérico inclui alguns metadados extras para dizer que é genérico, mas após a compilação o código de chamada não tem muito o que indicar que genéricos estavam envolvidos, e uma instância de um tipo genérico só conhece o lado não genérico de si mesmo. Por exemplo, uma instância de HashSet<E> não sabe se foi criado como HashSet<String> ou HashSet<Object>. O compilador adiciona efetivamente lançamentos quando necessário e realiza mais verificações de sanidade.

Aqui está um exemplo – primeiro o código Java genérico:

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("olá");
Entrada de string = strings.get(0);
strings.add(novo Objeto());
```

E aqui está o código não genérico equivalente:

```
ArrayList strings = new ArrayList();
strings.add("olá");
Entrada de string = (String) strings.get(0);
strings.add(novo Objeto());
```

Eles gerariam o mesmo bytecode Java, exceto a última linha, que é válida em o caso não genérico, mas será capturado pelo compilador como um erro no caso genérico versão. Você pode usar um tipo genérico como tipo bruto, que é semelhante a usar java.lang.Object para cada um dos argumentos de tipo. Essa reescrita – e perda de informação – é chamada de *apagamento de tipo*. Java não possui tipos de valores definidos pelo usuário, mas você não pode até mesmo usar os integrados como argumentos de tipo. Em vez disso, você deve usar as versões em caixa – ArrayList<Integer> para uma lista de números inteiros, por exemplo.

Você será perdoado por pensar que tudo isso é um pouco decepcionante em comparação com a situação geral. ics em C#, mas também existem alguns recursos interessantes dos genéricos Java:

ÿ A máquina virtual não sabe nada sobre genéricos, então você pode usar código compilado usando genéricos em uma versão mais antiga, desde que você não use nenhum classes ou métodos que não estão presentes na versão antiga. O versionamento em .NET é muito mais rigoroso em geral - para cada montagem referenciada, você pode especificar se o número da versão deve corresponder exatamente. Além disso, o código criado para ser executado no CLR 2.0 não será executado no .NET 1.1.

ÿ Você não precisa aprender um novo conjunto de classes para usar genéricos Java — onde um desenvolvedor não genérico usaria ArrayList, um desenvolvedor genérico apenas usaria Array-List<E>. As classes existentes podem ser atualizadas para versões genéricas com razoável facilidade.

ÿ O recurso anterior foi utilizado de forma bastante eficaz com o sistema de reflexão – java.lang.Class (o equivalente a System.Type) é genérico, o que permite que a segurança do tipo em tempo de compilação seja estendida para cobrir muitas situações que envolvem reflexão. Em algumas outras situações, é uma dor, no entanto.

Java tem suporte para variação genérica usando curingas. Por exemplo, `ArrayList <? extends Base>` pode ser lido como “este é um `ArrayList` de algum tipo que deriva de `Base`, mas não sabemos qual tipo exato”. Quando discutirmos o suporte do C# 4 para variância genérica no capítulo 13, revisitaremos isso com um pequeno exemplo.

Minha opinião pessoal é que os genéricos do .NET são superiores em quase todos os aspectos, embora, quando me deparo com questões de covariância/contravariância, muitas vezes deseje ter curingas. A variação genérica limitada do C# 4 melhora um pouco isso, mas ainda há momentos em que o modelo Java de variação funciona melhor. Java com genéricos ainda é muito melhor que Java sem genéricos, mas não há benefícios de desempenho e a segurança só se aplica em tempo de compilação.

## 3.6 Resumo

Ufa! É bom que os genéricos sejam mais simples de usar na realidade do que de descrever. Embora *possam* ser complicados, eles são amplamente considerados a adição mais importante ao C# 2 e são incrivelmente úteis. A pior coisa sobre escrever código usando genéricos é que, se você precisar voltar para o C# 1, sentirá muita falta deles.

(Felizmente isso está se tornando cada vez mais improvável, é claro.)

Neste capítulo, não tentei cobrir todos os detalhes do que é ou não permitido ao usar genéricos – esse é o trabalho da especificação da linguagem e torna a leitura árida. Em vez disso, busquei uma abordagem prática, fornecendo as informações necessárias no uso diário, com um pouco de teoria em prol do interesse acadêmico.

Vimos os três principais benefícios dos genéricos: segurança do tipo em tempo de compilação, desempenho e expressividade do código. Ser capaz de fazer com que o IDE e o compilador validem seu código antecipadamente é certamente uma coisa boa, mas é discutível que se pode ganhar mais com ferramentas que fornecem opções inteligentes baseadas nos tipos envolvidos do que no aspecto de segurança real.

O desempenho é melhorado de forma mais radical quando se trata de tipos de valor, que não precisam mais ser encaixotados e desempacotados quando são usados em APIs genéricas fortemente tipadas, particularmente os tipos de coleção genéricos fornecidos no .NET 2.0. O desempenho com tipos de referência geralmente é melhorado, mas apenas ligeiramente.

Seu código é capaz de expressar sua intenção de forma mais clara usando genéricos – em vez de um comentário ou um nome de variável longo ser necessário para descrever exatamente quais tipos estão envolvidos, os detalhes do tipo em si podem fazer o trabalho. Comentários e nomes de variáveis muitas vezes podem se tornar imprecisos com o tempo, pois podem ser esquecidos quando o código é alterado, mas as informações de tipo estão corretas por definição.

Os genéricos não são capazes de fazer *tudo* o que você às vezes gostaria que fizessem, e abordei algumas de suas limitações neste capítulo, mas se você realmente adotar o C# 2 e os tipos genéricos no .NET 2.0 Framework, você encontre bons usos para eles com uma frequência incrível em seu código.

Este tópico será abordado repetidamente em capítulos futuros, à medida que outros novos recursos se basearem neste tópico importante. Na verdade, o assunto do próximo capítulo seria muito diferente sem os genéricos — veremos os tipos anuláveis, conforme implementado por `Nullable<T>`.

# *Não dizendo nada com tipos anuláveis*

## **Este capítulo cobre**

- ÿ Razões para usar valores nulos
- ÿ Suporte de estrutura e tempo de execução para tipos anuláveis
- ÿ Suporte à linguagem C# 2 para tipos anuláveis
- ÿ Padrões usando tipos anuláveis

A nulidade é um conceito que tem provocado debate ao longo dos anos. Uma referência nula é um valor ou a ausência de um valor? “Nada” é “alguma coisa”? As linguagens deveriam apoiar o conceito de nulidade ou deveria ser representado em outros padrões?

Neste capítulo, tentarei ser mais prático do que filosófico. Primeiro, veremos por que há um problema — por que você não pode definir uma variável de tipo de valor como nula em C# 1 e quais têm sido as alternativas tradicionais. Depois disso, apresentarei nosso cavaleiro de armadura brilhante — `System.Nullable<T>` — e depois veremos como o C# 2 torna o trabalho com tipos anuláveis simples e compactos. Assim como os genéricos, os tipos com rótulo nulo às vezes têm usos além do que você poderia esperar, e veremos alguns exemplos deles no final do capítulo.

Então, quando um valor não é um valor? Vamos descobrir.

## 4.1 O que você faz quando simplesmente não tem valor?

Os designers de C# e .NET não adicionam recursos apenas por diversão. Tem que haver um verdadeiro, problema significativo que precisa ser corrigido antes que eles cheguem ao ponto de mudar o C# como linguagem ou .NET no nível da plataforma. Neste caso, o problema é melhor resumido em uma das perguntas mais frequentes em grupos de discussão sobre C# e .NET :

*Preciso definir minha variável DateTime como nula, mas o compilador não permite. O que devo fazer?<sup>1</sup>*

É uma questão que surge com bastante naturalidade – um exemplo pode ser num comércio eletrônico aplicativo onde os usuários visualizam o histórico de suas contas. Se um pedido tiver sido colocado, mas não entregue, pode haver uma data de compra, mas não uma data de envio, então como você representaria isso em um tipo destinado a fornecer os detalhes do pedido?

Antes do C# 2, a resposta à pergunta geralmente vinha em duas partes: uma explicação de por que você não poderia usar null em primeiro lugar, e uma lista de quais opções estavam disponíveis. Hoje em dia a resposta normalmente explicaria tipos anuláveis, mas vale a pena olhando as opções do C# 1 para entender de onde vem o problema.

### 4.1.1 Por que variáveis de tipo de valor não podem ser nulas

Como você viu no capítulo 2, o valor de uma variável de tipo de referência é uma referência, e o valor o valor de uma variável de tipo de valor são os próprios dados reais. Uma referência não nula é uma forma de chegar a um objeto, mas nulo atua como um valor especial, o que significa que *não me refiro a nenhum objeto*.

Se você quiser pensar nas referências como URLs, null é (*falando grosso modo*) a referência equivalente a about:blank. É representado como todos os zeros na memória (é por isso que é o valor padrão para todos os tipos de referência - limpando um bloco inteiro de a memória é barata, então é assim que os objetos são inicializados), mas ainda é basicamente armazenado da mesma forma que outras referências. Não há nada extra escondido em algum lugar para cada um variável de tipo de referência. Isso significa que você não pode usar o valor “todos os zeros” para uma referência real, mas tudo bem – sua memória vai acabar muito antes de você ter isso. muitos objetos vivos de qualquer maneira. Esta é a chave para explicar por que null não é um valor de tipo de valor válido .

Vamos considerar o tipo de byte como um tipo familiar e fácil de pensar. O valor que de uma variável do tipo byte é armazenado em um único byte - pode ser preenchido para alinhamento propósitos, mas o valor em si é conceitualmente composto apenas de um byte. Você *tem* que ser capaz de armazenar os valores de 0 a 255 nessa variável; caso contrário, é inútil ler dados binários arbitrários. Com os 256 valores normais e um valor nulo, você teria que lidar com um total de 257 valores, e não há como comprimir tantos valores em um único byte. Os designers poderiam ter decidido que cada tipo de valor teria um valor extra bit de sinalização em algum lugar determinando se um valor era nulo ou continha dados reais, mas as implicações do uso de memória são horríveis, sem mencionar o fato de que você teria para verificar o sinalizador toda vez que você quiser usar o valor. Resumindo, com tipos de valor muitas vezes você se preocupa em ter toda a gama de padrões de bits possíveis disponíveis como reais

---

<sup>1</sup> Quase sempre é DateTime em vez de qualquer outro tipo de valor. Não sei bem por que — é como se os desenvolvedores entendessem inherentemente por que um byte não deve ser nulo, mas sente que as datas são inherentemente anuláveis.

valores, enquanto com tipos de referência não há problema em perder um valor potencial para obter os benefícios de disponibilizar a referência nula .

Essa é a situação habitual. Agora, por que você *gostaria* de representar nulo para um tipo de valor, afinal? O motivo mais comum é simplesmente porque os bancos de dados normalmente suportam NULL como um valor para cada tipo (a menos que você torne especificamente o campo não anulável), então você pode ter dados de caracteres anuláveis, números inteiros anuláveis, Booleanos anuláveis – o todo funciona . Quando você busca dados de um banco de dados, geralmente não é uma boa ideia idéia de perder informações, então você quer ser capaz de representar a nulidade de qualquer coisa você lê, de alguma forma.

Isso apenas leva a questão um passo adiante. Por que os bancos de dados permitem valores nulos para datas, números inteiros e assim por diante? Valores nulos são normalmente usados para valores desconhecidos ou ausentes, como a data de envio no e-commerce anterior exemplo. A nulidade representa uma ausência de informação definida, o que pode ser importante em muitas situações. Na verdade, não é necessário haver um banco de dados envolvido para que os tipos de valores nulos sejam úteis; esse é apenas o cenário em que os desenvolvedores normalmente encontrar o problema primeiro.

Isso nos leva a opções para representar valores nulos em C# 1.

#### **4.1.2 Padrões para representação de valores nulos em C# 1**

Existem três padrões básicos comumente usados para contornar a falta de anulável tipos de valor em C# 1. Cada um tem seus prós e contras - principalmente contras - e todos eles são bastante insatisfatório. Mas vale a pena conhecê-los, em parte para apreciar melhor os benefícios da solução integrada em C# 2.

##### **PADRÃO 1: O VALOR MÁGICO**

O primeiro padrão é sacrificar um valor para representar um valor nulo. Isso tende a ser usado como solução para DateTime; poucas pessoas esperam que seus bancos de dados *realmente* contenham datas em AD 1, então DateTime.MinValue pode ser usado como um valor mágico conveniente sem perder nenhum dado útil. Por outras palavras, vai contra a linha de raciocínio que defendo dado anteriormente, que pressupõe que todos os valores possíveis precisam estar disponíveis para propósitos. O significado *semântico* de tal valor nulo variará de aplicação para aplicativo - pode significar que o usuário ainda não inseriu o valor em um formulário ou que não é necessário para esse registro, por exemplo.

A boa notícia é que usar um valor mágico não desperdiça memória nem requer qualquer novos tipos. Mas depende de você escolher um valor apropriado que você nunca desejará usar para dados reais. Além disso, é basicamente deselegante. Simplesmente não parece certo. Se você alguma vez precisar seguir esse caminho, você deve pelo menos usar uma constante (ou estática valor somente leitura para tipos que não podem ser expressos como constantes) para representar a mágica valor - comparações com DateTime.MinValue em todos os lugares, por exemplo, não expressam o significado do valor mágico. Além disso, é fácil usar acidentalmente a magia valor como se fosse um valor normal e significativo - nem o compilador nem o tempo de execução irá ajudá-lo a detectar o erro. Em contraste, a maioria das outras soluções apresentadas aqui

(incluindo aquele em C# 2) resultaria em um erro de compilação ou em uma exceção em tempo de execução, dependendo da situação exata.

O padrão de valor mágico está profundamente enraizado na computação na forma de IEEE-754 tipos binários de ponto flutuante, como float e double. Isso vai além da ideia de um único valor que representa *isso não é realmente um* número - existem *muitos* padrões de bits que são classificados como não-um-número (NaN), bem como valores para infinito positivo e negativo. Suspeito que poucos programadores (inclusive eu) são tão cautelosos com esses valores como deveríamos ser, o que é outra indicação das deficiências do padrão.

ADO.NET tem uma variação desse padrão onde o mesmo valor mágico—DBNull.Value — é usado para *todos* os valores nulos, independentemente do tipo. Neste caso, um adicional valor e, de fato, um tipo extra foi introduzido para indicar quando um banco de dados foi retornou nulo. Mas só é aplicável onde a segurança do tipo em tempo de compilação não é importante (em outras palavras, quando você quiser usar object e cast após testar a nulidade), e novamente não parece muito certo. Na verdade, é uma mistura do padrão de valor mágico e o padrão wrapper do tipo de referência, que veremos a seguir.

#### **PADRÃO 2: UM EMBALAGEM DE TIPO DE REFERÊNCIA**

A segunda solução pode assumir duas formas. A mais simples é usar object como tipo de variável, boxing e unboxing valores conforme necessário. Quanto mais complexo (e mais atraente) é ter um tipo de referência para cada tipo de valor necessário em um formato anulável formulário, contendo uma única variável de instância desse tipo de valor e com operadores de conversão implícitos de e para o tipo de valor. Com genéricos, você *poderia* fazer isso em um tipo genérico, mas se você estiver usando C# 2 de qualquer maneira, você também pode usar os tipos anuláveis descrito neste capítulo. Se você estiver preso no C# 1, precisará criar código-fonte para cada tipo que você deseja agrupar. Isso não é difícil de colocar na forma de um modelo para geração automática de código, mas ainda é um fardo que é melhor evitar se possível.

Ambas as formas têm o problema de que, embora permitam o uso de null diretamente, eles exigem que objetos sejam criados no heap, o que pode levar à pressão de coleta de lixo se você precisar usar essa abordagem com frequência e adicionar uso de memória devido à sobrecarga associada aos objetos. Para a solução mais complexa, você poderia tornar o tipo de referência mutável, o que pode reduzir o número de instâncias que você precisa criar, mas também pode gerar algum código não intuitivo.

#### **PADRÃO 3: UMA BANDEIRA BOOLEAN EXTRA**

O padrão final envolve um valor de tipo de valor normal e outro valor – um valor booleano. flag – indicando se o valor é “real” ou se deve ser desconsiderado. Novamente, existem duas maneiras de implementar esta solução. Ou você poderia manter duas variáveis separadas no código que usa o valor, ou você pode encapsular o value-plus-flag em outro tipo de valor.

Esta última solução é bastante semelhante à ideia mais complicada de tipo de referência descrito anteriormente, exceto que você evita o problema de coleta de lixo usando um valor digite e indique nulidade dentro do valor encapsulado em vez de uma referência nula. A desvantagem de ter que criar um novo desses tipos para cada tipo de valor

você deseja lidar é o mesmo, no entanto. Além disso, se o valor for colocado em caixa por algum motivo, ele será colocado em caixa da maneira normal, seja considerado nulo ou não.

O último padrão (na forma mais encapsulada) é efetivamente como os tipos anuláveis funcionam em C# 2, embora os novos recursos da estrutura, do CLR e da linguagem se combinem para fornecer uma solução que é significativamente mais organizada do que qualquer coisa que era possível em C# 1. A próxima seção trata do suporte fornecido pelo framework e pelo CLR no .NET 2: se o C# 2 suportasse apenas genéricos, a maior parte da seção 4.2 ainda seria relevante e o recurso ainda funcionaria e seria útil. Mas o C# 2 fornece açúcar sintático extra para torná-lo ainda melhor – esse é o assunto da seção 4.3.

## 4.2 System.Nullable<T> e System.Nullable

A estrutura central dos tipos anuláveis é a estrutura `System.Nullable<T>`. Além disso, a classe estática `System.Nullable` fornece métodos utilitários que ocasionalmente facilitam o trabalho com tipos anuláveis. (De agora em diante, deixarei de fora o namespace, para tornar a vida mais simples.) Veremos esses dois tipos separadamente e, nesta seção, evitarei quaisquer recursos extras fornecidos pela linguagem, para que você sermos capazes de entender o que está acontecendo no código IL quando observarmos a abreviação fornecida pelo C# 2.

### 4.2.1 Apresentando Nullable<T>

Como você pode perceber pelo nome, `Nullable<T>` é um tipo genérico. O parâmetro de tipo T possui uma restrição de tipo de valor, portanto você não pode usar `Nullable<Stream>`, por exemplo. Como mencionei na seção 3.3.1, isso também significa que você não pode usar outro tipo anulável como argumento, então `Nullable<Nullable<int>>` é proibido, mesmo que `Nullable<T>` seja um tipo de valor em todos os outros sentidos. O tipo de T para qualquer tipo anulável específico é chamado de *tipo subjacente* desse tipo anulável. Por exemplo, o tipo subjacente de `Nullable<int>` é `int`.

As partes mais importantes de `Nullable<T>` são suas propriedades `HasValue` e `Value`. Eles fazem o óbvio: `Value` representa o valor não anulável (*o real*, se preferir), quando existe um, e lança uma `InvalidOperationException` se (conceitualmente) não houver valor real. `.HasValue` é uma propriedade booleana que indica se existe um valor real ou se a instância deve ser considerada nula. Por enquanto, falarei sobre uma “instância com valor” ou uma “instância sem valor”, que significa uma instância onde a propriedade `HasValue` retorna verdadeiro ou falso, respectivamente.

Essas propriedades são apoiadas por campos simples da maneira óbvia. A Figura 4.1 mostra instâncias de `Nullable<int>` representando (da esquerda para a direita) nenhum valor, 0 e 5. Lembre-se de que `Nullable<T>` ainda é um tipo de valor, portanto, se você tiver uma variável do tipo `Nullable<int>`, o valor da variável conterá diretamente um `bool` e um `int` — não será uma referência a um objeto separado.

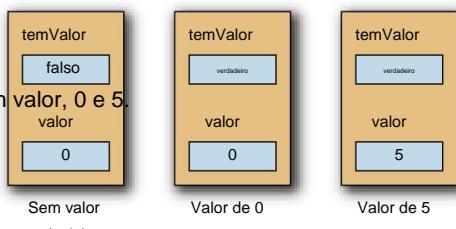


Figura 4.1 Exemplos de valores de `Nullable<int>`

Agora que você sabe o que as propriedades devem alcançar, vamos ver como você pode criar uma instância desse tipo. `Nullable<T>` tem dois construtores: o padrão (criando uma instância sem valor) e outro que toma uma instância de `T` como valor. Uma vez que uma instância foi construída, é imutável.

### Tipos de valor e mutabilidade

Diz-se que um tipo é imutável se for projetado de forma que uma instância não possa ser alterada depois de construído. Tipos imutáveis geralmente levam a um design mais limpo do que você obteria se tivesse que acompanhar o que pode estar alterando os valores compartilhados — especialmente entre diferentes threads.

A imutabilidade é particularmente importante para tipos de valor; eles deveriam quase sempre ser imutável. A maioria dos tipos de valor na estrutura são imutáveis, mas existem algumas exceções comumente usadas - em particular, as estruturas `Point` para Windows Formulários e `Windows Presentation Foundation` são mutáveis.

Se você precisar basear um valor em outro, siga o exemplo de `DateTime` e  `TimeSpan` — fornece métodos e operadores que retornam um novo valor em vez de modificar um valor existente. Isso evita todos os tipos de bugs sutis, incluindo situações em que você pode parecer estar mudando alguma coisa, mas na verdade está alterando uma cópia. Apenas diga não aos tipos de valores mutáveis.

`Nullable<T>` introduz um único novo método, `GetValueOrDefault`, que possui dois sobrecargas. Ambos retornam o valor da instância, se houver, ou um valor padrão, caso contrário. Uma sobrecarga não possui nenhum parâmetro (nesse caso, o valor padrão de o tipo subjacente é usado) e o outro permite que você especifique o valor padrão para retornar se necessário.

Todos os outros métodos implementados por `Nullable<T>` substituem os métodos existentes: `GetHashCode`, `ToString` e iguais. `GetHashCode` retorna 0 se a instância não tem um valor ou o resultado da chamada de `GetHashCode` no valor, se houver. `ToString` retorna uma string vazia se não houver um valor ou o resultado da chamada `ToString` no valor, se houver. `Equals` é um pouco mais complicado — nós iremos voltaremos a isso quando discutirmos o boxe.

Finalmente, duas conversões são fornecidas pela estrutura. Primeiro, há uma *implícita* conversão de `T` para `Nullable<T>`. Isso sempre resulta em uma instância onde `HasValue` retorna verdadeiro. Da mesma forma, há uma conversão *explícita* de `Nullable<T>` para `T`, que se comporta exatamente da mesma forma que a propriedade `Value`, incluindo o lançamento de uma exceção quando não há valor real para retornar.

**WRAPPING E UNWAPPING** A especificação C# nomeia o processo de conversão de uma instância de `T` em uma instância de *empacotamento* `Nullable<T>`, com o processo oposto óbvio sendo chamado de *desembrulhamento*. A especificação define estes termos com referência ao construtor que recebe um parâmetro e o valor propriedade, respectivamente. Na verdade, essas chamadas são geradas pelo código C# mesmo quando parecer que você está usando as conversões fornecidas pelo

estrutura. Os resultados são os mesmos de qualquer maneira. No restante deste capítulo, não farei distinção entre as duas implementações disponíveis.

Antes de prosseguirmos, vamos ver tudo isso em ação. A listagem a seguir mostra tudo o que você pode fazer diretamente com Nullable<T>, deixando Equals de lado por enquanto.

#### Listagem 4.1 Usando vários membros de Nullable<T>

```
Exibição de vazio estático (anulável<int> x)
{
    Console.WriteLine("HasValue: {0}", x.HasValue); se (x.HasValue)

    {
        Console.WriteLine("Valor: {0}", x.Valor); Console.WriteLine("Conversão
        explícita: {0}", (int)x);
    }
    Console.WriteLine("GetValueOrDefault(): {0}",
        x.GetValueOrDefault());
    Console.WriteLine("GetValueOrDefault(10): {0}",
        x.GetValueOrDefault(10));
    Console.WriteLine("ToString(): \"{0}\", x.ToString()); Console.WriteLine("GetHashCode(): {0}",
        x.GetHashCode()); Console.WriteLine();

    }
    ...
Anulável<int> x = 5; x = new
Anulável<int>(5); Console.WriteLine("Instância
com valor:"); Exibição(x);

x = new Nullable<int>();
Console.WriteLine("Instância sem valor."); Exibição(x);
```

Exibições  
Diagnóstico B

Valor
   
envolvente de 5

Instância de construções
   
sem valor

Na listagem 4.1, você primeiro usa duas maneiras diferentes (em termos de código-fonte C#) de agrupar um valor do tipo subjacente C e, em seguida, usa vários membros diferentes na instância B. Em seguida, você cria uma instância que *não* possui valor D e utiliza os mesmos membros na mesma ordem, apenas omitindo a propriedade Value e a conversão explícita para int, pois isso geraria exceções.

A saída da listagem 4.1 é a seguinte:

```
Instância com valor:
HasValue: Verdadeiro
Valor: 5
Conversão explícita: 5
GetValueOrDefault(): 5
GetValueOrDefault(10): 5
ToString(): "5"
ObterHashCode(): 5

Instância sem valor:
HasValue: Falso
GetValueOrDefault(): 0
GetValueOrDefault(10): 10
```

```
Para sequenciar(): ""
ObterHashCode(): 0
```

Até agora, você provavelmente poderia ter previsto todos os resultados observando os membros fornecidos por `Nullable<T>`. Porém, quando se trata de boxe e unboxing, há um comportamento especial para fazer com que os tipos anuláveis se comportem como você realmente *gostaria* que eles se comportassem, em vez de como eles se comportariam se você seguisse servilmente as regras normais do boxe.

#### 4.2.2 Encaixotamento `Nullable<T>` e unboxing

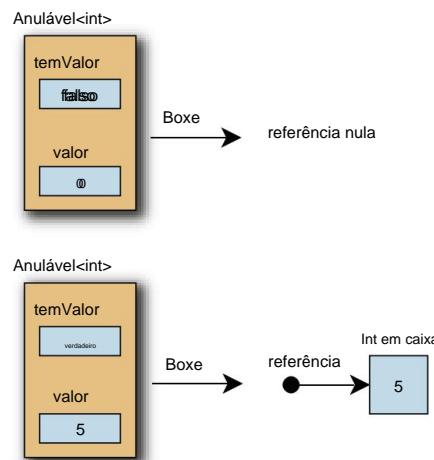
É importante lembrar que `Nullable<T>` é uma estrutura – um tipo de valor. Isso significa que se você quiser convertê-lo em um tipo de referência (objeto é o exemplo mais óbvio), você precisará encaixotá-lo. É apenas com relação ao boxing e unboxing que o CLR tem algum comportamento especial em relação aos tipos anuláveis – o resto são genéricos padrão, conversões, chamadas de método e assim por diante. Na verdade, o comportamento só foi alterado pouco antes do lançamento do .NET 2.0, como resultado de solicitações da comunidade. Nas versões de visualização, os tipos de valor anuláveis foram colocados em caixa como qualquer outro tipo de valor.

Uma instância de `Nullable<T>` está em caixa para uma referência nula (se não tiver um valor) ou um valor em caixa de `T` (se tiver), conforme mostrado na figura 4.2. Ele nunca se ajusta a um “int anulável em caixa” – há

não existe esse tipo.

Você pode desempacotar um valor em caixa para seu tipo normal ou para o tipo anulável correspondente. Desembalar uma referência nula lançará um `NullReferenceException`.

Exceção se você descompactar para o tipo normal, mas descompactará para uma instância sem valor se você descompactar para o tipo anulável apropriado. Esse comportamento é mostrado na listagem a seguir.

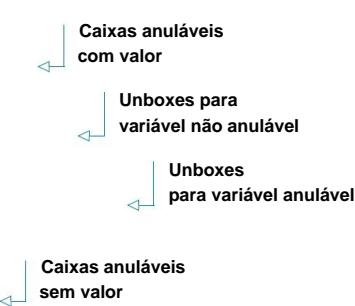


**Figura 4.2** Resultados do boxe de uma instância sem valor (parte superior) e com valor (parte inferior)

#### Listagem 4.2 Comportamento de boxing e unboxing de tipos anuláveis

```
Anulável<int> anulável = 5;
objeto em caixa = anulável;
Console.WriteLine(boxed.GetType());
int normal = (int)encaixotado;
Console.WriteLine(normal);
anulável = (anulável<int>)encaixotado;
Console.WriteLine(anulável);

anulável = new Nullable<int>(); em caixa = anulável;
```



```
Console.WriteLine(caixa == nulo);
anulável = (anulável<int>)encaixotado;
Console.WriteLine(nullable.HasValue);
```

← Unboxes para variável anulável

A saída da listagem 4.2 mostra o tipo do valor em caixa como System.Int32 (não System.Nullable<System.Int32>). Isso confirma que você pode recuperar o valor desempacotando para int ou para Nullable<int>. Por fim, a saída demonstra que você pode encaixotar de uma instância anulável sem um valor para uma referência nula e descompactar novamente com êxito para outra instância anulável sem valor. Se você tivesse tentado desembalar o último valor de boxed para um int não anulável , o programa teria explodido com uma NullReferenceException.

Agora que você entende o comportamento de boxing e unboxing, podemos começar a abordar o comportamento de Nullable<T>.Equals.

#### 4.2.3 Igualdade de instâncias Nullable<T>

Nullable<T> substitui object.Equals(object) mas não introduz nenhum operador de igualdade nem fornece um método Equals(Nullable<T>) . Como a estrutura forneceu os blocos de construção básicos, as linguagens podem adicionar funcionalidades extras, inclusive fazer com que os operadores existentes funcionem como você espera. Você verá os detalhes disso na seção 4.3.3, mas a igualdade básica, conforme definida pelo método vanilla Equals , segue estas regras para uma chamada para first.Equals(second):

- ÿ Se primeiro não tiver valor e segundo for nulo, eles são iguais. ÿ
- Se primeiro não tiver valor e segundo não for nulo, eles não são iguais. ÿ
- Se primeiro tiver um valor e segundo for nulo, eles não são iguais. ÿ
- Caso contrário, eles serão iguais se o valor do primeiro for igual ao segundo.

Observe que você não precisa considerar o caso em que second é outro Nullable<T> porque as regras do boxe proíbem essa situação. O tipo de segundo é objeto, portanto, para ser um Nullable<T>, ele teria que ser encaixotado e, como você acabou de ver, encaixotar uma instância anulável cria uma caixa do tipo não anulável ou retorna um nulo referência. Inicialmente, a primeira regra pode parecer quebrar o contrato do objeto .Equals(object), que insiste que x.Equals(null) retorne falso — mas isso somente quando x é uma referência não nula. Novamente, devido ao comportamento do boxing, a implementação de Nullable<T> nunca será chamada por meio de uma referência.

As regras são principalmente consistentes com as regras de igualdade em outras partes do .NET, portanto, você pode usar instâncias anuláveis como chaves para dicionários e quaisquer outras situações em que precise de igualdade. Só não espere que a igualdade diferencie entre uma instância não anulável e uma instância anulável com um valor — ela foi cuidadosamente configurada para que esses dois casos *sejam* tratados da mesma maneira.

Isso cobre a própria estrutura Nullable<T> , mas tem um parceiro obscuro: a classe Nullable .

#### 4.2.4 Suporte da classe Nullable não genérica

A estrutura System.Nullable<T> faz quase tudo o que você deseja, mas recebe ajuda da classe System.Nullable . Esta é uma classe estática – ela contém apenas métodos estáticos e você não pode criar uma instância dela.<sup>2</sup> Na verdade, tudo o que ela faz poderia ter sido feito igualmente bem por outros tipos, e se a Microsoft tivesse mostrado mais visão, o A classe anulável pode nem ter existido - o que teria evitado alguma confusão sobre a finalidade dos dois tipos. Mas este acidente da história tem três métodos em seu nome, e eles ainda são úteis.

Os dois primeiros são métodos de comparação:

```
public static int Comparar<T>(Nullable<T> n1, Nullable<T> n2) public static bool  
Equals<T>(Nullable<T> n1, Nullable<T> n2)
```

Compare usa Comparer<T>.Default para comparar os dois valores subjacentes (se existirem) e Equals usa EqualityComparer<T>.Default. Quando apresentados a instâncias sem valores, os resultados retornados de cada método obedecem às convenções do .NET de comparação de nulos iguais entre si e menores do que qualquer outra coisa.

Ambos os métodos poderiam facilmente fazer parte de Nullable<T> como métodos estáticos, mas não genéricos. A única pequena vantagem de tê-los como métodos genéricos em um tipo não genérico é que a inferência de tipo genérico pode ser aplicada, portanto raramente será necessário especificar explicitamente o parâmetro de tipo.

O método final de System.Nullable não é genérico – não poderia ser. Sua assinatura é do seguinte modo:

```
tipo estático público GetUnderlyingType (tipo nullableType)
```

Se o parâmetro for do tipo anulável, o método retornará seu tipo subjacente; caso contrário, ele retornará nulo. A razão pela qual este não poderia ser um método genérico é que se você conhecesse o tipo subjacente para começar, não precisaria chamá-lo.

Agora você viu o que a estrutura e o CLR fornecem para suportar valores anuláveis tipos - mas o C# 2 adiciona recursos de linguagem para tornar a vida muito mais agradável.

#### 4.3 O açúcar sintático do C# 2 para tipos anuláveis

Até agora você viu

tipos anuláveis fazendo seu trabalho, mas os exemplos não foram particularmente bonitos de se ver. É certo que está claro que você está usando tipos anuláveis quando precisa digitar Nullable<> em torno do nome do tipo no qual está interessado, mas isso torna a nulidade mais proeminente do que o tipo subjacente, o que geralmente não é uma boa ideia.

Além disso, o próprio nome *nullable* sugere que você deve ser capaz de atribuir null a uma variável de tipo anulável, e você não viu isso – você sempre usou o construtor padrão do tipo. Nesta seção, veremos como o C# 2 lida com esses e outros problemas.

---

<sup>2</sup> Você aprenderá mais sobre classes estáticas no capítulo 7.

Antes de entrarmos nos detalhes do que o C# 2 oferece como linguagem, há uma definição que posso finalmente apresentar. O *valor nulo* de um tipo de valor anulável é o valor onde HasValue retorna falso — ou uma “instância sem valor”, como me referi a ele na seção 4.2. Não usei o termo antes porque é específico para C#. A especificação CLI não menciona isso, e a documentação do Nullable<T> em si não menciona isso. Honrei essa diferença esperando até falarmos especificamente sobre C# 2 antes de introduzir o termo. O termo também se aplica a tipos de referência: o valor nulo de um tipo de referência é simplesmente a referência nula com a qual você está familiarizado no C# 1.

**TIPO NULO VERSUS TIPO DE VALOR NULO** Na especificação da linguagem C#, *tipo anulável* é usado para significar qualquer tipo com um valor nulo – portanto, qualquer tipo de referência ou qualquer Nullable<T>. Você deve ter notado que tenho usado esse termo como se fosse sinônimo de *tipo de valor anulável* (que obviamente não inclui tipos de referência). Embora eu normalmente seja um grande pedante quando se trata de terminologia, se eu tivesse usado “tipo de valor anulável” em todo este capítulo, teria sido horrível de ler. Você também deve esperar que o “tipo anulável” seja usado de forma ambígua no mundo real: provavelmente é mais comum usá-lo ao descrever Nullable<T> do que no sentido descrito na especificação.

Com isso resolvido, vamos ver quais recursos o C# 2 nos oferece, começando por reduzir a confusão em nosso código.

### 4.3.1 O ? modificador

Existem alguns elementos de sintaxe que podem não ser familiares a princípio, mas que transmitem uma sensação apropriada a eles. O operador condicional (a ? b : c) é um deles para mim – ele faz uma pergunta e depois tem duas respostas correspondentes. Da mesma forma, o ? modificador para tipos anuláveis parece certo.

O ? modificador é uma forma abreviada de especificar um tipo anulável, portanto, em vez de usar Nullable <byte>, você pode usar byte? em todo o seu código. Os dois são intercambiáveis e compilam exatamente na mesma IL, então você pode misturá-los e combiná-los se quiser – mas em nome de quem quer que leia seu código em seguida, recomendo que você escolha uma forma ou de outra e use-o de forma consistente. A listagem a seguir é exatamente equivalente à listagem 4.2, mas usa o ? modificador, conforme mostrado em negrito.

#### Listagem 4.3 O mesmo código da 4.2, mas usando o ? modificador

```
interno? anulável = 5;  
  
objeto em caixa = anulável;  
Console.WriteLine(boxed.GetType());  
  
int normal = (int) encaixotado;  
Console.WriteLine(normal);  
  
anulável = (int?) encaixotado;  
Console.WriteLine(anulável);  
  
anulável = novo int?();
```

```
em caixa = anulável;
Console.WriteLine(caixa == nulo);

anulável = (int?) encaixotado;
Console.WriteLine(nullable.HasValue);
```

Não vou explicar o que o código faz ou como faz, porque o resultado é exatamente o mesmo que na listagem 4.2. As duas listagens são compiladas no mesmo IL – elas simplesmente use sintaxe diferente, assim como int é intercambiável com System.Int32. Você pode usar o versão abreviada em todos os lugares, inclusive em assinaturas de métodos, expressões typeof , elencos e similares.

A razão pela qual sinto que o modificador foi bem escolhido é que ele adiciona um ar de incerteza ao a natureza da variável. A variável anulável na listagem 4.3 tem um número inteiro valor? Bem, em qualquer momento específico pode ser, ou pode ser o valor nulo.

De agora em diante, usarei o ? modificador em todos os exemplos – é mais simples e é sem dúvida a maneira idiomática de usar tipos anuláveis em C#. Mas você pode achar que é muito fácil errar ao ler o código; nesse caso, não há nada que o impeça de usar o sintaxe mais longa. Você pode querer comparar as listagens nesta seção e na seção anterior para ver o que você acha mais claro.

Dado que a especificação C# 2 define o valor nulo, seria estranho se não foi possível usar o literal nulo que já está na linguagem para representá-lo. Felizmente pudermos...

### 4.3.2 Atribuindo e comparando com nulo

Um autor conciso poderia cobrir toda esta seção em uma única frase: "O compilador C# permite o uso de null para representar o valor nulo de um tipo anulável em comparações e atribuições". Prefiro mostrar o que isso significa em código real e considere por que o idioma recebeu esse recurso.

Você pode ter se sentido desconfortável toda vez que usou o construtor padrão do Anulável<T>. Alcança o comportamento desejado , mas não expressa o motivo você quer fazer isso - isso não deixa a impressão certa no leitor. Idealmente, deveria dar o mesmo tipo de sensação que o uso de null dá com tipos de referência.

Se lhe parece estranho que eu tenha falado sobre sentimentos tanto nesta seção quanto na anterior, pense apenas em quem escreve o código e quem o lê. Claro, o compilador precisa entender o código e não está nem aí para as sutis nuances de estilo, mas poucos trechos de código usados em sistemas de produção são escritos e nunca lidos de novo. Qualquer coisa que você possa fazer para levar o leitor ao processo mental que você estava passando até quando você escreveu originalmente o código é bom, e usar o conhecido literal nulo ajuda a conseguir isso.

Com isso em mente, deixaremos de usar um exemplo que apenas mostra sintaxe e comportamento para aquele que dá uma impressão de como os tipos anuláveis podem ser usados. Bem modele uma classe Person onde você precisa saber o nome, data de nascimento e data de uma pessoa de morte. Acompanharemos apenas as pessoas que nasceram definitivamente, mas algumas delas essas pessoas ainda podem estar vivas - caso em que a data da morte será representada por

nulo. A listagem a seguir mostra alguns dos códigos possíveis. Uma classe real teria mais operações disponíveis – neste exemplo veremos apenas o cálculo da idade.

#### Listagem 4.4 Parte de uma classe Person incluindo cálculo de idade

```

classe Pessoa
{
    DataHora nascimento;
    Data hora? morte; nome
    da sequência;

    Idade pública do TimeSpan
    {
        pegar
        {
            if (morte == nulo)           ← B verifica HasValue
            {
                retornar DateTime.Now - nascimento;
            }
            outro
            {
                retornar morte.Valor - nascimento;   ← C Desembrulha para cálculo
            }
        }
    }

    Pessoa pública (nome da string, DateTime
                    nascimento, DateTime?
                    morte)
    {
        this.nascimento = nascimento;
        esta.morte = morte; este.nome
        = nome;
    }
}
...
Pessoa turing = new Pessoa("Alan Turing",
                           novo DateTime(1912, 6, 23), novo
                           DateTime(1954, 6, 7));   ← D Envolve DateTime
                                         como anulável

Pessoa knuth = new Pessoa("Donald Knuth",
                           novo DateTime(1938, 1, 10), null);   ← E Especifica data
                                         nula de falecimento

```

A Listagem 4.4 não produz nenhuma saída, mas o fato de ela ser compilada pode ter surpreendido você antes de ler este capítulo. Além do uso do ? modificador causando confusão, você pode ter achado estranho poder comparar um DateTime? com nulo ou passe nulo como argumento para um DateTime? parâmetro.

Esperamos que agora o significado seja intuitivo - quando você compara a variável death com null, você está perguntando se seu valor é nulo ou não. Da mesma forma quando você usa null como DateTime?. Por exemplo, você está realmente criando o valor nulo para o tipo chamando o construtor padrão. Na verdade, você pode ver no IL gerado que o código que o compilador cospe para a listagem 4.4 realmente apenas chama a propriedade death.HasValue **B** e cria uma nova instância de DateTime? **E** usando o construtor padrão (representado em IL

como a instrução `initobj`). A data da morte de Alan Turing é criada chamando o construtor `DateTime` normal e depois passando o resultado para o construtor `Nullable<Date-Time>` que recebe um parâmetro.

Mencionei olhar para o IL porque pode ser uma maneira útil de descobrir o que seu código está realmente fazendo, principalmente se algo for compilado quando você não espera. Você pode usar a ferramenta `ildasm` que vem com o .NET SDK ou um dos muitos descompiladores disponíveis agora, como .NET Reflector, ILSpy, dotPeek ou JustDecompile.

(Sempre que me refiro ao Reflector neste livro, é apenas porque essa é a ferramenta que uso por hábito. As outras também funcionam perfeitamente, tenho certeza.)

Você viu como o C# fornece uma sintaxe abreviada para o conceito de um valor nulo, tornando o código mais expressivo quando os tipos anuláveis são entendidos em primeiro lugar. Mas uma parte da listagem 4.4 deu um pouco mais de trabalho do que você esperava — a subtração em C. Por que você teve que desembrulhar o valor? Por que você não poderia simplesmente devolver a morte - o nascimento diretamente? O que você gostaria que essa expressão significasse se a morte fosse nula (excluída neste código pelo teste anterior)? Essas perguntas – e mais – serão respondidas na próxima seção.

### 4.3.3 Conversões e operadores anuláveis

Você viu que é possível comparar instâncias de tipos anuláveis com `null`, mas existem outras comparações que podem ser feitas e outros operadores que podem ser usados em alguns casos. Da mesma forma, você viu agrupar e desembrulhar, mas outras conversões podem ser usadas com alguns tipos. Esta seção explica o que está disponível. Receio que seja praticamente impossível tornar esse tipo de tópico genuinamente interessante, mas recursos cuidadosamente projetados como esses são o que tornam o C# uma linguagem agradável de se trabalhar no longo prazo.

Não se preocupe se tudo não for absorvido na primeira vez: apenas lembre-se de que os detalhes estão aqui se você precisar consultá-los no meio de uma sessão de codificação.

O resumo executivo é que se houver um operador ou conversão disponível em um tipo de valor não anulável, esse operador ou conversão envolver apenas outros tipos de valor não anuláveis, então o tipo de valor anulável também terá o mesmo operador ou conversão disponível, geralmente convertendo os tipos de valor não anuláveis em seus equivalentes anuláveis. Para dar um exemplo mais concreto, há uma conversão implícita de `int` para `long`, e isso significa que também há uma conversão implícita de `int?` muito tempo? que se comporta da maneira óbvia.

Infelizmente, embora essa descrição ampla dê a ideia geral correta, as regras exatas são um pouco mais complicadas. Cada regra é simples, mas existem algumas delas. Vale a pena conhecê-las porque, caso contrário, você pode acabar encarando um erro ou aviso do compilador por um tempo, perguntando-se por que ele acredita que você está tentando fazer uma conversão que nunca planejou em primeiro lugar. Começaremos com as conversões e depois veremos os operadores.

## CONVERSÕES ENVOLVENDO TIPOS NULOS

Para completar, vamos começar com as conversões que você já conhece:<sup>3</sup> Uma conversão implícita do literal nulo para T?<sup>3</sup> Uma conversão implícita de T para T?<sup>3</sup> Uma conversão explícita de T? para T

Agora considere as conversões predefinidas e definidas pelo usuário disponíveis nos tipos. Por exemplo, há uma conversão predefinida de int para long. Para qualquer conversão como esta, de um tipo de valor não anulável (S) para outro (T), as seguintes conversões também estão disponíveis:

- S? para T? (explícito ou implícito dependendo da conversão original)<sup>3</sup> S para T?
- (explícito ou implícito dependendo da conversão original)<sup>3</sup> S? para T (sempre explícito)

Para levar o exemplo adiante, isso significa que você pode converter implicitamente de int? muito tempo? e de int para long? bem como explicitamente de int? muito tempo. As conversões se comportam de forma natural, com valores nulos de S? convertendo em valores nulos de T? e valores não nulos usando a conversão original. Como antes, a conversão explícita de S? para T lançará uma InvalidOperationException ao converter de um valor nulo de S?. Para conversões definidas pelo usuário, essas conversões extras envolvendo tipos anuláveis são conhecidas como *conversões elevadas*.

Até agora, é relativamente simples. Agora vamos considerar os operadores, onde as coisas são um pouco mais complicadas.

## OPERADORES ENVOLVENDO TIPOS NULOS

C# permite que os seguintes operadores sejam sobrecarregados:<sup>3</sup>

- Unário: + ++ - - ! ~ truefalse<sup>3</sup> Binário: + - \* %
- & | ^<sup>3</sup> Igualdade: == !=<sup>3</sup> Relacional: << >>
- < > <= >=

Quando esses operadores estão sobrecarregados para um tipo de valor não anulável T, o tipo anulável T? tem os mesmos operadores, com operandos e tipos de resultados ligeiramente diferentes. Eles são chamados de *operadores elevados*, sejam operadores predefinidos, como adição em tipos numéricos, ou operadores definidos pelo usuário, como adição de TimeSpan a DateTime.

Existem algumas restrições quanto ao momento em que se aplicam:

- Os operadores verdadeiro e falso nunca são eliminados. Eles são incrivelmente raros em primeiro lugar, então não é uma grande perda.
- Somente operadores com tipos de valor não anuláveis para os operandos são levantados.

---

<sup>3</sup> Os operadores de igualdade e relacionais também são operadores binários, mas se comportam de maneira ligeiramente diferente dos demais; daí sua separação nesta lista.

- ↳ Para os operadores unários e binários (exceto igualdade e operações relacionais), o tipo de retorno deve ser um tipo de valor não anulável.
- ↳ Para os operadores de igualdade e relacionais, o tipo de retorno deve ser bool. ↳ O & e | operadores em bool? têm um comportamento definido separadamente, que você veja na seção 4.3.4.

Para todos os operadores, os tipos de operandos tornam-se seus equivalentes anuláveis. Para os operadores unários e binários, o tipo de retorno também se torna anulável e um valor nulo será retornado se algum dos operandos for um valor nulo. Os operadores de igualdade e relacionais mantêm seus tipos de retorno booleanos não anuláveis. Para igualdade, dois valores nulos são considerados iguais, e um valor nulo e qualquer valor não nulo são considerados diferentes, o que é consistente com o comportamento que você viu na seção 4.2.3. Os operadores relacionais sempre retornam falso se um dos operandos for um valor nulo. Quando nenhum dos operandos é um valor nulo, o operador do tipo não anulável é invocado da maneira óbvia.

Todas essas regras parecem mais complicadas do que realmente são – na maior parte, tudo funciona como você provavelmente espera. É mais fácil ver o que acontece com alguns exemplos, e como int tem tantos operadores predefinidos (e números inteiros podem ser expressos facilmente), é o tipo de demonstração natural. A Tabela 4.1 mostra uma série de expressões, a assinatura do operador levantada e o resultado. Supõe-se que existam variáveis quatro, cinco e nullInt, cada uma com o tipo int? e com os valores óbvios.

**Tabela 4.1 Exemplos de operadores levantados aplicados a inteiros anuláveis**

Expressão	Operador levantado	Resultado
-nullInt	interno? -(int?x)int? -(int?	nulo
-cinco	x)int? +(int? x, int? y) int?	-5
cinco + nullInt	+ (int? x, int? y) bool == (int? x, int? y)	nulo
cinco + cinco	bool == (int? x, int? y) bool == (int? x, int?	10
nullInt == nullInt	y) bool == (int? x, int? y) bool < (int? x, int?	verdadeiro
cinco == cinco	y) bool < (int? x, int? y) bool < (int? x, int?	verdadeiro
cinco == nullInt	y) bool < (int? x, int? y) bool <= (int? x, int?	falso
cinco == quatro	y)	falso
quatro < cinco		verdadeiro
nullInt < cinco		falso
cinco < nullInt		falso
nullInt < nullInt		falso
nullInt <= nullInt		falso

Possivelmente a linha mais surpreendente da tabela é a última – que um valor nulo não é considerado menor ou igual a outro valor nulo, mesmo que sejam considerados iguais entre si (conforme a quinta linha)! Muito estranho, mas improvável que cause problemas na vida real, na minha experiência.

Um aspecto dos operadores levantados e da conversão anulável que causou alguma confusão são as comparações não intencionais com null ao usar um tipo de valor não anulável. O código a seguir é legal, mas não é útil:

```

int eu = 5; if (i
== nulo) {

    Console.WriteLine ("Nunca vai acontecer");
}

```

O compilador C# gera avisos sobre esse código, mas você pode considerar surpreendente que ele seja permitido. O que está acontecendo é que o compilador vê a expressão int no lado esquerdo de ==, vê nulo no lado direito e sabe que há uma conversão implícita para int? de cada um deles. Porque uma comparação entre dois int? valores são perfeitamente válidos, o código não gera um erro – apenas o aviso. Como complicação adicional, isso *não* é permitido no caso em que, em vez de int, você está lidando com um parâmetro de tipo genérico que foi restringido para ser um tipo de valor - as regras sobre genéricos proíbem a comparação com null em essa situação.

De qualquer forma, haverá um erro ou um aviso, portanto, contanto que você observe atentamente os avisos, você não deverá acabar com um código deficiente devido a essa peculiaridade, e espero que o fato de eu apontar isso para você agora salve você fique com dor de cabeça tentando descobrir exatamente o que está acontecendo.

Agora você pode responder à pergunta no final da seção anterior – por que usamos morte.Valor - nascimento na listagem 4.4 em vez de apenas morte - nascimento. Aplicando as regras anteriores, você *poderia* ter usado a última expressão, mas o resultado teria sido TimeSpan? em vez de um TimeSpan. Isso deixaria você com as opções de converter o resultado para TimeSpan usando sua propriedade Value ou alterar a propriedade Age para retornar um TimeSpan?, o que apenas empurra o problema para o chamador. Ainda é um pouco feio, mas você verá uma implementação melhor da propriedade Age na seção 4.3.6.

Na lista de restrições quanto ao levantamento do operador, mencionei aquele bool? funciona de maneira um pouco diferente dos outros tipos. A próxima seção explica isso e puxa a lente para ver o panorama geral de por que todos esses operadores funcionam dessa maneira.

#### 4.3.4 Lógica anulável

Lembro-me vividamente das minhas primeiras aulas de eletrônica na escola. Eles sempre pareciam girar em torno do cálculo da tensão em diferentes partes de um circuito usando a fórmula  $V = IxR$  ou da aplicação de *tabelas verdade* – os gráficos de referência para explicar a diferença entre portas NAND e portas NOR e assim por diante. A ideia é simples: uma tabela verdade mapeia todas as combinações possíveis de entradas em qualquer parte da lógica em que você esteja interessado e informa a saída.

As tabelas verdade que desenhamos para portas lógicas simples de duas entradas sempre tiveram quatro linhas – cada entrada tinha dois valores possíveis, o que significa que havia quatro combinações possíveis. A lógica booleana é simples assim, mas o que acontece quando você tem um tipo lógico tristate? Bem, bool? é exatamente esse tipo - o valor pode ser verdadeiro, falso ou nulo. Isso significa que suas tabelas verdade agora precisam de nove linhas para operadores binários, pois há nove combinações. A especificação destaca apenas os operadores lógicos AND e OR inclusivo (& e |, respectivamente) porque os outros operadores - negação lógica unária (!) e OR exclusivo (^) - seguem as mesmas regras que outros operadores levantados. Há

nenhum operador lógico condicional (os operadores `&&` e `||` de curto-circuito ) definidos para `bool?`, o que torna a vida mais simples.

Para fins de completude, a tabela 4.2 fornece a tabela verdade para todos os quatro valores `bool?` Operadores lógicos.

**Tabela 4.2 Tabela verdade para os operadores lógicos AND, OR inclusivo, OR exclusivo e negação lógica, aplicados ao `bool?` tipo**

x	sim	x e y	x   sim	x ^ sim	!x
verdadeiro	verdadeiro	verdadeiro	verdadeiro	falso	falso
verdadeiro	falso	falso	verdadeiro	verdadeiro	falso
verdadeiro	nulo	nulo	verdadeiro	nulo	falso
falso	verdadeiro	falso	verdadeiro	verdadeiro	verdadeiro
falso	falso	falso	falso	falso	verdadeiro
falso	nulo	falso	nulo	nulo	verdadeiro
nulo	verdadeiro	nulo	verdadeiro	nulo	nulo
nulo	falso	falso	nulo	nulo	nulo
nulo	nulo	nulo	nulo	nulo	nulo

Se você achar que o raciocínio sobre regras é mais fácil de entender do que procurar valores em tabelas, a ideia é que um `null bool?` valor é, em alguns sentidos, um “talvez”. Se você imaginar que cada entrada nula no lado de entrada da tabela é uma variável, você sempre obterá um valor nulo no lado de saída da tabela se o resultado depender do valor dessa variável. Por exemplo, olhando para a terceira linha da tabela, a expressão `true & y` só será verdadeira se `y` for verdadeiro, mas a expressão `true | y` sempre será verdadeiro , seja qual for o valor de `y` , portanto, os resultados anuláveis são nulos e verdadeiros, respectivamente.

Ao considerar os operadores levantados e particularmente como funciona a lógica anulável, os designers da linguagem tinham dois conjuntos ligeiramente contraditórios de comportamento existente: referências nulas C# 1 e valores SQL NULL . Em muitos casos, eles não entram em conflito — o C# 1 não tinha nenhum conceito de aplicação de operadores lógicos a referências nulas, portanto não houve problema em usar os resultados semelhantes ao SQL fornecidos anteriormente. As definições que você viu podem surpreender alguns desenvolvedores de SQL , no entanto, quando se trata de comparações. No SQL padrão , o resultado da comparação de dois valores (em termos de igualdade ou maior que/menor que) é sempre desconhecido se um dos valores for NULL. O resultado em C# 2 nunca é nulo e, em particular, dois valores nulos são considerados iguais entre si.

**LEMBRETE: ISTO É C# ESPECÍFICO!** Vale lembrar que os operadores e conversões levantados, junto com o `bool?` lógica descrita nesta seção são todas fornecidas pelo compilador C# e não pelo CLR ou pela própria estrutura. Se você usar `ildasm` no código que avalia qualquer um desses operadores anuláveis, descobrirá que o compilador criou todos os IL apropriados para testar valores nulos e os tratou adequadamente. Isso significa que diferentes linguagens podem se comportar de maneira diferente nesses assuntos – definitivamente algo a ser observado se você precisar portar código entre diferentes linguagens baseadas em .NET. Por exemplo, o VB trata os operadores elevados muito mais como SQL, então o resultado de `x < y` é `Nothing` se `x` ou `y` for `Nothing`.

Outro operador familiar agora está disponível com tipos de valor anuláveis e se comporta exatamente como você esperaria se você considerasse seu conhecimento existente sobre referências nulas e apenas o ajustasse em termos de valores *nulos*.

#### 4.3.5 Usando o operador as com tipos anuláveis

Antes do C# 2, o operador `as` estava disponível apenas para tipos de referência. A partir do C# 2, agora ele também pode ser aplicado a tipos de valor anuláveis. O resultado é um valor desse tipo anulável – ou o valor nulo se a referência original for do tipo errado ou nula, ou um valor significativo caso contrário. Aqui está um pequeno exemplo:

```
static void PrintValueAsInt32 (objeto o) {
    internal? anulável = o como int?;
    Console.WriteLine(nullable.HasValue? nullable.Value.ToString():
        "nulo");
}
...
PrintValueAsInt32(5);                                ← Rende 5
PrintValueAsInt32("alguma string");                  ← Rende nulo
```

Isso permite converter com segurança de uma referência arbitrária para um valor em uma única etapa – embora normalmente você verifique. No C# 1, você teria que usar o operador `is` seguido por uma conversão, o que é deselegante: é essencialmente pedir ao CLR para realizar a mesma verificação de tipo duas vezes.

**ARMADILHA DE DESEMPENHO SURPREENDENTE** Sempre achei que fazer uma verificação seria mais rápido do que duas, mas parece que esse não é o caso — pelo menos com as versões do .NET com as quais testei (até o .NET 4.5, inclusive). Ao escrever um benchmark rápido que somava todos os números inteiros dentro de uma matriz do tipo `object[]`, onde apenas um terço dos valores eram na verdade números inteiros em caixa, usar `is` e então uma conversão acabou sendo *20 vezes mais rápida* do que usar o operador `as`. Os detalhes estão além do escopo deste livro e, como sempre, você deve testar o desempenho com seu código e dados reais antes de decidir o melhor curso de ação para sua situação específica, mas vale a pena estar ciente disso.

Agora você sabe o suficiente para usar tipos anuláveis e prever como eles se comportarão, mas o C# 2 tem uma espécie de “faixa bônus” quando se trata de aprimoramentos de sintaxe: o operador de coalescência nula.

#### 4.3.6 O operador coalescente nulo

Além do `?`  modificador, todos os demais truques do compilador C# relacionados aos tipos nulos até agora funcionaram com a sintaxe existente. Mas o C# 2 introduz um novo operador que pode ocasionalmente tornar o código mais curto e mais agradável. É chamado de *operador de coalescência nula* e aparece no código como `??` entre seus dois operandos. É como o operador condicional, mas especialmente ajustado para nulos.

É um operador binário que avalia primeiro `??` em segundo lugar , seguindo o seguinte etapas (grosso modo):

1 Avalie primeiro.

2 Se o resultado não for nulo, esse será o resultado de toda a expressão.

3 Caso contrário, avalie em segundo lugar; o resultado então se torna o resultado do todo expressão.

Digo “grosso modo” porque as regras formais na especificação têm que lidar com situações que envolvem conversões entre os tipos de primeiro e segundo. Como sempre, eles não são importantes na maioria dos usos do operador, e não pretendo abordá-los – consulte a seção 7.13 da especificação (“O Operador Coalescente Nulo”) se precisar de detalhes.

É importante ressaltar que se o tipo do segundo operando for o tipo subjacente do primeiro operando (e, portanto, não anulável), o resultado geral será esse tipo subjacente. Por exemplo, este código é perfeitamente válido:

```
interno? uma = 5;
interno b = 10; int
c = uma ?? b;
```

Observe que você está atribuindo diretamente a c mesmo que seu tipo seja do tipo int não anulável . Você só pode fazer isso porque b não é anulável, então você sabe que eventualmente obterá um resultado não anulável.

Obviamente, esse é um exemplo bastante simplista; vamos encontrar um uso mais prático para esse operador revisitando a propriedade Age da listagem 4.4. Como lembrete, veja como foi implementado naquela época, juntamente com as declarações de variáveis relevantes:

```
DataHora nascimento;
Data hora? morte;

public TimeSpan Idade {

    pegar
    {
        if (morte == nulo) {

            retornar DateTime.Now - nascimento;
        }
        outro
        {
            retornar morte.Valor - nascimento;
        }
    }
}
```

Observe como ambos os ramos da instrução if subtraem o valor do nascimento de alguns valor DateTime não nulo . O valor no qual você está interessado é a última hora em que a pessoa esteve viva – a hora da morte da pessoa, se ela já morreu, ou agora , caso contrário. Para progredir em pequenos passos, vamos tentar primeiro usar o operador condicional normal:

```
DateTime lastAlive = (morte == nulo? DateTime.Now: morte.Value); retornar lastAlive - nascimento;
```

Isso é uma espécie de progresso, mas sem dúvida o operador condicional tornou a leitura mais difícil do que fácil, embora o novo código seja mais curto. O operador condicional geralmente é assim: o quanto você o utiliza é uma questão de preferência pessoal, embora valha a pena consultar o restante da sua equipe antes de usá-lo extensivamente. Vamos ver como o operador de coalescência nula melhora as coisas. Você deseja usar o valor de death se não for nulo e DateTime.Now caso contrário. Você pode alterar a implementação para o seguinte:

```
DateTime lastAlive = morte ?? DateTime.Agora; retornar lastAlive -  
nascimento;
```

Observe como o tipo de resultado é DateTime em vez de DateTime? porque você usou DateTime.Now como segundo operando. Você *poderia* encurtar tudo para uma expressão:

```
retorno (morte ?? DateTime.Now) - nascimento;
```

Mas isso é mais obscuro — em particular, na versão de duas linhas, o nome da variável last-Alive ajuda o leitor a ver por que você está aplicando o operador de coalescência nula. Espero que você concorde que a versão de duas linhas é mais simples e mais legível do que a versão original usando a instrução if ou a versão usando o operador condicional normal do C# 1. É claro que depende do leitor entender o que é nulo. operador de coalescência faz. Na minha experiência, esse é um dos aspectos menos conhecidos do C# 2, mas é útil o suficiente para valer a pena tentar esclarecer seus colegas de trabalho em vez de evitá-lo.

Existem mais dois aspectos que aumentam a utilidade do operador. Primeiro, ela não se aplica apenas a tipos de valor anuláveis — ela também funciona com tipos de referência; você simplesmente não pode usar um tipo de valor não anulável para o primeiro operando, pois isso seria inútil. Além disso, é *associativo correto*, o que significa primeiro uma expressão na forma ?? segundo ?? o terceiro é avaliado como primeiro ?? (segundo ?? terceiro) - e assim continua para mais operandos. Você pode ter qualquer número de expressões e elas serão avaliadas em ordem, parando com o primeiro resultado não nulo. Se todas as expressões forem avaliadas como nulas, o resultado também será nulo.

Como um exemplo concreto disso, suponha que você tenha um sistema de pedidos online com os conceitos de endereço de cobrança, endereço de contato e endereço de entrega. As regras de negócio declaram que qualquer usuário *deve* ter um endereço de cobrança, mas o endereço de contato é opcional. O endereço de entrega para um pedido específico também é opcional, sendo o padrão o endereço de cobrança. Esses endereços opcionais são facilmente representados como referências nulas no código. Para determinar quem deve ser contatado no caso de um problema com uma remessa, o código em C# 1 pode ser semelhante a este:

```
Endereço contato = user.ContactAddress; if (contato == nulo) {
```

```
    contato = pedido.ShippingAddress; if (contato ==  
        nulo) {
```

```

    contato = usuário.BillingAddress;
}
}

```

Usar o operador condicional neste caso é ainda mais horrível. Mas usar o operador de coalescência nula torna o código muito simples:

```

Endereço contato = user.ContactAddress ??
                    pedido.ShippingAddress ?? usuário.BillingAddress;

```

Se as regras de negócios mudassem para usar o endereço de entrega por padrão em vez do endereço de contato do usuário, a mudança aqui seria extremamente óbvia. Não seria *particularmente* desgastante com a versão `if/else`, mas sei que teria que parar e pensar duas vezes e verificar o código mentalmente. Eu também confiaria em testes de unidade, então haveria poucas chances de realmente errar, mas prefiro não pensar em coisas como essa, a menos que seja absolutamente necessário.

**TUDO COM MODERAÇÃO** Caso você esteja pensando que meu código está repleto de usos do operador de coalescência nula, na verdade não é. Costumo considerar isso quando vejo mecanismos padrão envolvendo nulos e possivelmente o operador condicional, mas isso não aparece com frequência. Porém, quando seu uso é natural, pode ser uma ferramenta poderosa na batalha pela legibilidade.

Você viu como tipos anuláveis podem ser usados para propriedades comuns de objetos — casos em que você naturalmente pode não ter um valor para algum aspecto específico que ainda seja melhor expresso com um tipo de valor. Esses são os usos mais óbvios para tipos anuláveis e, na verdade, os mais comuns. Alguns outros padrões não são tão óbvios, mas ainda podem ser poderosos quando você está acostumado com eles. Exploraremos dois desses padrões na próxima seção. Isso é mais por uma questão de interesse do que como parte do aprendizado sobre o comportamento dos próprios tipos anuláveis — agora você tem todas as ferramentas necessárias para usá-los em seu próprio código. Se você estiver interessado em ideias peculiares e talvez tentar algo novo, continue lendo...

## 4.4 Novos usos de tipos anuláveis

Antes dos tipos anuláveis se tornarem realidade, vi muitas pessoas solicitando-os efetivamente, geralmente em relação ao acesso ao banco de dados. No entanto, esse não é o único uso que eles podem ter. Os padrões apresentados nesta seção não são convencionais, mas podem tornar o código mais simples. Se você sempre segue as expressões normais do C#, tudo bem — esta seção pode não ser para você, e tenho muita simpatia por esse ponto de vista. Geralmente prefiro código simples a código inteligente, mas se um *padrão* completo oferece benefícios, isso às vezes faz com que valha a pena aprender o padrão. Depende inteiramente de você usar essas técnicas — você pode descobrir que elas sugerem outras ideias para usar em outras partes do seu código.

Sem mais delongas, vamos começar com uma alternativa ao padrão TryXXX . mencionado na seção 3.3.3.

#### 4.4.1 Tentando uma operação sem usar parâmetros de saída

O padrão de usar um valor de retorno para dizer se uma operação funcionou e usar um parâmetro de saída para retornar o resultado real está se tornando cada vez mais comum no .NET Framework. Não tenho problemas com os objectivos – a ideia de que alguns métodos são *susceptíveis* de falhar no cumprimento do seu objectivo principal em circunstâncias não excepcionais é senso comum. Meu único problema é que não sou um grande fã de parâmetros de saída. Há algo um pouco desajeitado na sintaxe de declarar uma variável em uma linha e, em seguida, usá-la imediatamente como parâmetro de saída.

Os métodos que retornam tipos de referência geralmente usam um padrão de retorno nulo em caso de falha e não nulo em caso de sucesso, mas isso não funciona tão bem quando nulo é um valor de retorno válido no caso de sucesso. Hashtable é um exemplo de ambas as declarações, de uma forma ligeiramente ambivalente: null é um valor teoricamente válido em um Hashtable, mas na minha experiência a maioria dos usos de Hashtable nunca usa valores nulos, o que torna perfeitamente aceitável ter código que assume que um valor nulo significa uma chave ausente.

Um cenário comum é ter cada valor da Hashtable como uma lista: o primeiro sempre que um item é adicionado para uma chave específica, uma nova lista é criada e o item é adicionado a ela. Depois disso, adicionar outro item para a mesma chave envolve adicionar o item à lista existente. Aqui está o código em C# 1:

```
ListaArrayList = hash[chave]; if (lista ==  
nulo) {  
  
    lista = new ArrayList(); hash[chave]  
    = lista;  
  
} lista.Adicionar(novoItem);
```

Esperamos que você use nomes de variáveis mais específicos para sua situação, mas tenho certeza de que você entendeu e pode muito bem ter usado o padrão sozinho.<sup>4</sup> Com tipos anuláveis, esse padrão pode ser estendido para tipos de valor e é *mais seguro* com tipos de valor, porque se o tipo de resultado natural for um tipo de valor, um valor nulo só poderá ser retornado como resultado de falha. Tipos anuláveis adicionam aquela informação booleana extra de uma maneira geral agradável com suporte a linguagem, então por que não usá-los?

Para demonstrar esse padrão na prática e em um contexto diferente das pesquisas de dicionário, usei o exemplo clássico do padrão TryXXX – analisando um número inteiro. A implementação do método TryParse na listagem a seguir mostra a versão do padrão usando um parâmetro de saída, mas você vê a versão usando tipos anuláveis na parte principal na parte inferior.

---

<sup>4</sup> Não seria ótimo se Hashtable e Dictionary<TKey, TValue> pudesse chamar um delegado sempre que um novo valor fosse necessário devido à procura de uma chave ausente? Situações como essa seriam muito mais simples.

## Listagem 4.5 Uma implementação alternativa do padrão TryXXX

```

int estático? TryParse(string de texto)
{
    int ret; if
    (int.TryParse(texto, out ret)) {

        retornar ret;
    }
    outro
    {
        retornar nulo;
    }
}
...
interno? analisado = TryParse("Não válido"); if (analisado! =
Nulo)
{
    Console.WriteLine ("Analizado para {0}", analisado.Value);
}
outro
{
    Console.WriteLine ("Não foi possível analisar");
}

```

Você pode pensar que há pouco para distinguir as duas versões aqui – afinal, elas têm o mesmo número de linhas. Mas acredito que há uma diferença de ênfase. A versão anulável encapsula o valor de retorno natural e o sucesso ou falha em uma única variável. Também separa o *fazer do testar*, o que coloca a ênfase no lugar certo, na minha opinião. Normalmente, se eu chamar um método na parte de condição de uma instrução if , o objetivo principal desse método é retornar um valor booleano. Aqui, o valor de retorno é, de certa forma, menos importante que o parâmetro de saída. Quando você está lendo código, é fácil perder um parâmetro de saída em uma chamada de método e ficar se perguntando o que realmente está fazendo todo o trabalho e dando a resposta magicamente. Com a versão anulável, isso é mais explícito - o resultado do método contém todas as informações nas quais você está interessado. Usei essa técnica em vários lugares (geralmente com mais parâmetros de método, ponto em que os parâmetros de saída tornou-se ainda mais difícil de detectar) e acredito que melhorou a sensação geral do código. Claro, isso só funciona para tipos de valor.

Outra vantagem desse padrão é que ele pode ser usado em conjunto com o operador de coalescência nula – você pode tentar entender diversas entradas, parando na primeira válida. O padrão TryXXX normal permite isso usando operadores de curto-circuito, mas o significado não é tão claro quando você usa a mesma variável para dois parâmetros de saída diferentes na mesma instrução.

**ALTERNATIVAMENTE, USE UMA TUPLA...** Outra alternativa ao uso de um resultado anulável é usar um tipo de retorno com dois membros claramente separados, um dos quais é responsável por indicar sucesso ou falha e outro é responsável por indicar o valor em caso de sucesso . Nullable<T> é conveniente porque

fornecendo uma propriedade booleana e outra do tipo T, mas o *significado* do valor de retorno talvez pudesse ser mais explícito. O .NET 4 inclui a família de tipos Tuple : sem dúvida um Tuple<int, bool> pode ser mais limpo que int? aqui. Ainda mais limpo seria um tipo personalizado para representar o resultado de uma operação de análise: ParseResult<T>, por exemplo. Nesse caso, você pode passar o valor para outro código sem medo de que seu significado seja confuso e pode adicionar informações extras, como a *causa* de qualquer falha na análise.

O próximo padrão é uma resposta a um ponto problemático específico – a irritação e a perplexidade que podem estar presentes ao escrever comparações em vários níveis.

#### 4.4.2 Comparações simples com o operador coalescente nulo

Suspeito que você não goste de escrever o mesmo código repetidamente tanto quanto eu. A refatoração muitas vezes pode eliminar a duplicação, mas alguns casos resistem à refatoração de forma surpreendentemente eficaz. Code for Equals and Compare geralmente se enquadra nessa categoria.

Suponha que você esteja escrevendo um site de comércio eletrônico e tenha uma lista de produtos. Você pode querer classificá-los por popularidade (decrescente), depois por preço e depois por nome - para que os produtos classificados com cinco estrelas venham primeiro, mas os produtos cinco estrelas mais baratos venham antes dos mais caros. Se houver vários produtos com o mesmo preço, os produtos que começam com A serão listados antes dos produtos que começam com B. Este não é um problema específico de sites de comércio eletrônico – classificar dados por vários critérios é um requisito bastante comum na computação.

Supondo que você tenha um tipo de produto adequado , você poderia escrever a comparação com código como este em C# 1:

```
public int Compare(Produto primeiro, Produto segundo) {  
  
    // Comparação reversa de popularidade para classificação decrescente int ret =  
    second.Popularity.CompareTo(first.Popularity); se (ret! = 0) {  
  
        retornar ret;  
  
    } ret = primeiro.Price.CompareTo(second.Price); se (ret! = 0) {  
  
        retornar ret;  
  
    } return primeiro.Nome.CompareTo(segundo.Nome);  
}
```

Isso pressupõe que você não será solicitado a comparar referências nulas e que todas as propriedades também retornarão referências não nulas. Você poderia usar algumas comparações nulas iniciais e Comparer<T>.Default para lidar com esses casos, mas isso tornaria o código ainda mais longo e mais complexo. O código poderia ser mais curto (e evitar retornar do meio do método) se você o reorganizasse um pouco, mas o padrão fundamental “comparar, verificar, comparar, verificar” ainda estaria presente e não seria tão óbvio que uma vez você tem uma resposta diferente de zero, pronto.

Ah... essa última frase lembra outra coisa: o operador coalescente nulo. Como você viu na seção 4.3, se você tiver muitas expressões separadas por ??, então o operador será aplicado repetidamente até atingir uma expressão não nula. Agora tudo o que você precisa fazer é descobrir uma maneira de retornar nulo em vez de zero em uma comparação. Isso é fácil de fazer em um método separado que também pode encapsular o uso do comparador padrão. Você pode até ter uma sobrecarga para usar um comparador específico se quiser. Você também pode lidar com o caso em que qualquer uma das referências de produto que você passou é nula.

Primeiro, vamos dar uma olhada na classe que implementa os métodos auxiliares, conforme mostrado na listagem a seguir.

#### Listagem 4.6 Classe auxiliar para fornecer comparações parciais

```
classe estática pública PartialComparer {

    público estático int? Compare<T>(T primeiro, T segundo) {

        return Compare(Comparer<T>.Default, primeiro, segundo);
    }

    público estático int? Compare<T>(IComparer<T> comparador, T primeiro, T segundo)

    {
        int ret = comparador.Compare(primeiro, segundo); retornar ret == 0? novo int?() : ret;
    }

    público estático int? ReferenceCompare<T>(T primeiro, T segundo) onde T : classe

    {
        retornar primeiro == segundo? 0
        : primeiro == nulo? -1
        : segundo == nulo? 1
        : novo int?();
    }
}
```

Os métodos Compare na listagem 4.6 são quase pateticamente simples – quando um comparador não é especificado, o comparador padrão para o tipo é usado, e tudo o que acontece com o valor de retorno da comparação é que zero é traduzido para o valor nulo.

**VALORES NULOS E O OPERADOR CONDICIONAL** Você pode ter ficado surpreso ao me ver usar new int?() em vez de null para retornar o valor nulo no segundo método Compare . O operador condicional exige que seu segundo e terceiro operandos sejam do mesmo tipo, ou que haja uma conversão implícita de um para outro, e isso não seria o caso de null, porque o compilador não saberia que tipo o valor deveria ser. As regras de linguagem não levam em consideração o objetivo geral da instrução (retornando de um método com um tipo de retorno int?) ao examinar subexpressões. Outras opções incluem converter qualquer operando para int? explicitamente ou usando default(int?) para o valor nulo. Basicamente, o importante é ter certeza de que um dos operandos é um int? valor.

O método ReferenceCompare usa outro operador condicional – três deles, na verdade. Você pode achar isso menos legível do que o código equivalente (um pouco mais longo) usando blocos if/else – depende de quanto confortável você está com o operador condicional. Gosto porque deixa clara a ordem das comparações. Além disso, este poderia facilmente ter sido um método não genérico com dois parâmetros de objeto, mas este formulário evita que você use acidentalmente o método para comparar tipos de valor por meio de boxing. O método realmente só é útil com tipos de referência, o que é indicado pela restrição do parâmetro de tipo.

Embora esta classe seja simples, é extremamente útil. Agora você pode substituir a comparação de produtos anterior por uma implementação mais simples:

```
public int Compare(Produto primeiro, Produto segundo) {
    retornar PC.ReferenceCompare(primeiro, segundo) ??
        // Comparação reversa de popularidade para classificar em ordem decrescente
        PC.Compare(second.Popularity, first.Popularity) ??
        PC.Compare(primeiro.Price, segundo.Price) ??
        PC.Compare(primeiro.Nome, segundo.Nome) ?? 0;
}
```

Como você deve ter notado, usei PC em vez de PartialComparer – isso apenas para poder ajustar as linhas na página impressa. Em código real, eu usaria o nome completo do tipo e ainda teria uma comparação por linha. Claro, se você quisesse linhas curtas por algum motivo, você poderia especificar uma diretiva using para tornar PC um alias para Comparador Parcial - eu simplesmente não recomendaria isso.

O 0 final indica que, se todas as comparações anteriores foram aprovadas, as duas instâncias de Produto serão iguais. Você *poderia* simplesmente usar Comparer<string>.Default.Compare(first.Name, second.Name) como comparação final, mas isso prejudicaria a simetria do método.

Essa comparação funciona bem com nulos, é fácil de modificar, forma um padrão fácil de usar para outras comparações e compara apenas na medida do necessário – se os preços forem diferentes, os nomes não serão comparados.

Você pode estar se perguntando se a mesma técnica poderia ser aplicada a testes de igualdade, que geralmente apresentam padrões semelhantes. Há muito menos sentido no caso de igualdade, porque após os testes de nulidade e igualdade de referência, você pode simplesmente usar && para fornecer a funcionalidade de curto-circuito desejada para booleanos. Um método retornando um bool? pode ser usado para obter um resultado inicial *definitivamente igual*, *definitivamente diferente* ou *desconhecido* com base nas referências. O código completo do PartialComparer no site deste livro contém o método utilitário apropriado e exemplos de seu uso.

## 4.5 Resumo

Quando confrontados com um problema, os desenvolvedores tendem a escolher a solução mais fácil de curto prazo, mesmo que não seja particularmente elegante. Essa é muitas vezes a decisão certa – afinal, você não quer ser culpado de excesso de engenharia. Mas é sempre bom quando uma *boa* solução é também a solução *mais fácil*.

Os tipos anuláveis resolvem um problema específico que só tinha soluções um tanto feias antes do C# 2. Os recursos fornecidos são uma versão com melhor suporte de uma solução que era viável, mas demorada no C# 1. A combinação de genéricos (para evitar duplicação de código), CLR suporte (para fornecer comportamento adequado de boxing e unboxing) e suporte de linguagem (para fornecer sintaxe concisa junto com conversões e operadores convenientes) tornam a solução muito mais atraente do que era anteriormente.

Acontece que, ao fornecer tipos anuláveis, os projetistas do C# e da estrutura disponibilizaram alguns outros padrões que antes não valiam o esforço. Vimos alguns deles neste capítulo e não ficaria surpreso em ver mais deles aparecendo ao longo do tempo.

Até agora, os tipos genéricos e anuláveis abordaram áreas onde no C# 1 você ocasionalmente tinha que tapar o nariz devido a cheiros desagradáveis de código. Esse padrão continua no próximo capítulo, onde discutiremos as melhorias nos delegados. Eles constituem uma parte importante da mudança útil de direção da linguagem C# e do .NET Framework em direção a um ponto de vista um pouco mais funcional. Essa ênfase fica ainda mais clara no C# 3, portanto, embora *ainda* não estejamos analisando esses recursos, os aprimoramentos delegados no C# 2 atuam como uma ponte entre a familiaridade do C# 1 e o estilo idiomático do C# 3, que muitas vezes pode ser radicalmente diferente das ve-

# Delegados acelerados

## Este capítulo cobre

- ÿ Sintaxe C# 1 prolixo
- ÿ Construção simplificada de delegados
- ÿ Covariância e contravariância
- ÿ Métodos anônimos
- ÿ Variáveis capturadas

A jornada dos delegados em C# e .NET tem sido interessante, mostrando uma visão notável (ou muita sorte) por parte dos designers. As convenções sugeridas para manipuladores de eventos no .NET 1.0/1.1 não faziam muito sentido — até o C# 2 aparecer. Da mesma forma, o esforço colocado nos delegados para C# 2 parece, de certa forma, desproporcional ao quanto amplamente usados eles são — até que você veja quanto difundidos eles são no código idiomático C# 3. Em outras palavras, é como se os designers da linguagem e da plataforma tivessem uma visão pelo menos da direção aproximada que tomariam, anos antes de o destino em si se tornar claro.

É claro que o C# 3 não é um destino final em si — os delegados genéricos obtêm um pouco mais de flexibilidade no C# 4, o C# 5 facilita a gravação de delegados assíncronos e podemos ver ainda mais avanços no futuro — mas as diferenças entre C#1 e

C# 3 nesta área são os mais surpreendentes. (A principal mudança nos delegados de suporte do C# 3 está nas expressões lambda, que você conhecerá no capítulo 9.)

C# 2 é uma espécie de trampolim em termos de delegados. Seus novos recursos preparam o caminho para as mudanças drásticas do C# 3, mantendo os desenvolvedores *razoavelmente* confortáveis e ao mesmo tempo fornecendo benefícios úteis. Fui informado com segurança de que os designers de linguagem estavam cientes de que o conjunto de recursos combinados do C# 2 abriria novas maneiras de ver o código, mas eles não sabiam necessariamente aonde esses caminhos levariam. Até agora, os seus instintos revelaram-se extremamente benéficos na área dos delegados.

Os delegados desempenham um papel mais importante no .NET 2.0 do que nas versões anteriores, embora não sejam tão comuns como no .NET 3.5. No capítulo 3 você viu como eles podem ser usados para converter de um tipo de lista para outro, e no capítulo 1 você classificou uma lista de produtos usando o delegado Comparison em vez da interface IComparer. Embora a estrutura e o C# mantenham uma distância respeitosa um do outro sempre que possível, acredito que a linguagem e a plataforma se orientaram neste caso: a inclusão de mais chamadas de API baseadas em delegados suporta a sintaxe aprimorada disponível no C# 2 e vice-versa.

Neste capítulo, veremos como o C# 2 inclui duas pequenas alterações que facilitam a vida ao criar instâncias delegadas a partir de métodos normais e, em seguida, veremos a maior mudança: métodos anônimos, que permitem especificar uma instância delegada.ação inline no ponto de sua criação. A maior seção do capítulo é dedicada à parte mais complicada dos métodos anônimos — variáveis capturadas — que fornecem às instâncias delegadas um ambiente mais rico para atuar. Abordaremos o tópico com detalhes significativos devido à sua importância e complexidade. Depois que você dominar os métodos anônimos, as expressões lambda serão fáceis de entender.

Porém, primeiro vamos revisar os pontos problemáticos dos recursos de delegação do C# 1.

## 5.1 Dizendo adeus à estranha sintaxe de delegado

A sintaxe para delegados em C# 1 não parece tão ruim — a linguagem já tem açúcar sintático em torno de Delegate.Combine, Delegate.Remove e da invocação de instâncias delegadas. Faz sentido especificar o tipo de delegado ao criar uma instância delegada; afinal, é a mesma sintaxe usada para criar instâncias de outros tipos.

Tudo isso é verdade, mas por algum motivo também é uma droga. É difícil dizer exatamente por que as expressões de criação delegada do C# 1 causam polêmica, mas elas causam — pelo menos para mim. Ao conectar vários manipuladores de eventos, parece feio ter que escrever um novo manipulador de eventos (ou o que for necessário) em todos os lugares, quando o próprio evento especificou qual tipo de delegado será usado. A beleza está nos olhos de quem vê, é claro, e você poderia argumentar que há menos necessidade de suposições ao ler o código de fiação do manipulador de eventos no estilo C# 1, mas o texto extra apenas atrapalha e desvia a atenção da parte importante do código: o método com o qual você deseja tratar o evento.

A vida se torna mais preta e branca quando você considera a covariância e a contravariância aplicadas aos delegados. Suponha que você tenha um método de manipulação de eventos que salva o documento atual, ou registra que ele foi chamado, ou executa uma série de outras tarefas.

ações que podem não precisar saber detalhes do evento. O evento em si não deve se importar com o fato de seu método ser capaz de trabalhar apenas com as informações fornecidas pela assinatura `EventHandler`, mesmo que o evento seja declarado para passar detalhes do evento do mouse. Infelizmente, em C# 1 você precisa ter um método diferente para cada assinatura diferente do manipulador de eventos.

Da mesma forma, é inegavelmente feio escrever métodos que são tão simples que sua implementação é mais curta que sua assinatura, apenas porque os delegados precisam ter uma ação para executar na forma de um método. Ele adiciona uma camada extra de indireção entre o código que *cria* a instância delegada e o código que deve ser executado quando for invocado. Camadas extras de indireção geralmente são bem-vindas — essa opção não foi removida no C# 2 — mas, ao mesmo tempo, frequentemente torna o código mais difícil de ler e polui a classe com vários métodos que são usados apenas para delegados.

Não é novidade que todos esses problemas foram bastante melhorados no C# 2. A sintaxe ainda pode ser mais prolixo do que você gostaria (até obter expressões lambda no C# 3), mas a diferença é significativa. Para ilustrar o problema, começaremos com algum código em C# 1 e o melhoraremos nas próximas seções. A listagem a seguir cria um formulário (muito) simples com um botão e depois inscreve três eventos do botão.

#### Listagem 5.1 Inscrever-se em três eventos de um botão

```
static void LogPlainEvent(objeto remetente, EventArgs e) {  
  
    Console.WriteLine("LogPlain");  
}  
  
static void LogKeyEvent(objeto remetente, KeyPressEventArgs e) {  
  
    Console.WriteLine("LogKey");  
}  
  
static void LogMouseEvent(objeto remetente, MouseEventArgs e) {  
  
    Console.WriteLine("LogMouse");  
}  
...  
Botão botão = novo Botão(); button.Text =  
"Clique em mim"; botão.Click += new  
EventHandler(LogPlainEvent); button.KeyPress += new KeyPressEventHandler(LogKeyEvent);  
button.MouseClick += new MouseEventHandler(LogMouseEvent);  
  
Formulário formulário = new  
Formulário(); formulário.AutoSize  
= verdadeiro; formulário.Controls.Add(botão);  
Aplicativo.Run(formulário);
```

As linhas de saída nos três métodos de manipulação de eventos existem para provar que o código está funcionando: se você pressionar a barra de espaço com o botão destacado, verá que os eventos Click e KeyPress são gerados. Pressionar Enter apenas gera o evento Click ;

clicar no botão gera os eventos Click e MouseClick . Nas seções a seguir, melhoraremos esse código usando alguns dos recursos do C# 2.

Vamos começar pedindo ao compilador que faça uma dedução bastante óbvia – que exclui tipo de portão que você deseja usar ao se inscrever em um evento.

## 5.2 Conversões de grupos de métodos

Em C# 1, se você quiser criar uma instância delegada, será necessário especificar o tipo de delegado e a ação. O Capítulo 2 definiu a ação como o método a ser chamado e (por exemplo, métodos) o alvo como o objeto no qual ela é chamada.

Por exemplo, na listagem 5.1, esta expressão foi usada para criar um KeyPressEvent-Manipulador:

```
novo KeyPressEventHandler (LogKeyEvent)
```

Como expressão independente, não parece tão ruim. Mesmo usado em uma assinatura de evento simples, é tolerável. Porém, fica mais feio quando usado como parte de uma expressão mais longa. Um exemplo comum disso é iniciar um novo tópico:

```
Thread t = new Thread(new ThreadStart(MeuMetodo));
```

O que você quer fazer é iniciar um novo thread que executará MyMethod. Como sempre, você deseja se expressar da maneira mais simples possível, e o C# 2 permite fazer isso por meio de uma conversão implícita de um *grupo de métodos* para um tipo de delegado compatível. Um grupo de métodos é simplesmente o nome de um método, opcionalmente com um destino — exatamente o mesmo tipo de expressão que você usou em C# 1 para criar instâncias delegadas. (Na verdade, a expressão era chamada de grupo de métodos naquela época — só que a conversão não estava disponível.) Se o método for genérico, o grupo de métodos também pode especificar argumentos de tipo — embora isso raramente seja usado, na minha experiência. A nova conversão implícita permite que você transforme sua assinatura de evento em

```
button.KeyPress += LogKeyEvent;
```

Da mesma forma, o código de criação de thread torna-se simplesmente

```
Thread t = new Thread(MeuMetodo);
```

As diferenças de legibilidade entre as versões original e simplificada não são enormes para uma única linha, mas no contexto de uma quantidade significativa de código, podem reduzir consideravelmente a confusão. Para fazer com que pareça menos mágica, vamos dar uma olhada rápida no que essa conversão está fazendo.

Primeiro, vamos considerar as expressões LogKeyEvent e MyMethod conforme aparecem nos exemplos. A razão pela qual são classificados como *grupos de métodos* é porque mais de um método pode estar disponível, devido à sobrecarga. As conversões implícitas disponíveis converterão um grupo de métodos em qualquer tipo de delegado com uma assinatura compatível. Então, se você tivesse duas assinaturas de método como essas,

```
void MyMethod() void  
MyMethod(remetente do objeto, EventArgs e)
```

você poderia usar MyMethod como grupo de métodos em uma atribuição para um ThreadStart ou um EventHandler, como segue:

```
ThreadStart x = MeuMetodo;
EventHandler y = MeuMetodo;
```

Mas você *não poderia* usá-lo como parâmetro para um método que estivesse sobrecarregado para receber um ThreadStart ou um EventHandler - o compilador reclamaria que a chamada era ambígua. Da mesma forma, infelizmente você não pode usar uma conversão implícita de grupo de métodos para converter para o tipo System.Delegate simples , porque o compilador não sabe de qual tipo de delegado específico criar uma instância. Isso é chato, mas você ainda *pode* ser um pouco mais breve do que no C# 1, tornando a conversão explícita. Aqui está um exemplo:

```
Delegado inválido = SomeMethod;
Delegado válido = (ThreadStart)SomeMethod;
```

Para variáveis locais isso geralmente não é um problema, mas é um pouco mais irritante quando você usa uma API que possui um parâmetro do tipo Delegate, como Control.Invoke.

Existem algumas soluções aqui: usar um método auxiliar, lançar ou usar uma variável intermediária. Aqui está um exemplo usando o tipo delegado MethodInvoker , que não aceita parâmetros e não retorna nada:

```
static void SimpleInvoke(Controle de controle,
                         Invocador do MethodInvoker)
{
    control.Invoke(invocador);
}
...
SimpleInvoke(formulário, UpdateUI);
form.Invoke((MethodInvoker)UpdateUI); Invocador do
MethodInvoker = UpdateUI; form.Invoke(invocador);
```

Situações diferentes encorajarão soluções diferentes; nada disso é particularmente atraente, mas também não é horrível.<sup>1</sup>

Tal como acontece com os genéricos, as regras precisas de validade da conversão são um pouco complicadas e a abordagem apenas experimente funciona bem; se o compilador reclamar que não tem informações suficientes, basta dizer qual conversão usar e tudo ficará bem. Se não reclamar, você deve ficar bem. Para detalhes exatos, consulte a especificação do idioma, seção 6.6 (“Conversões de grupos de métodos”). Falando em possíveis conversões, pode haver mais do que você espera, como verá na próxima seção.

## 5.3 Covariância e contravariância

Já falamos muito sobre os conceitos de covariância e contravariância em diferentes contextos, geralmente lamentando sua ausência, mas a construção de delegados é a única área em que eles estão disponíveis em C# anteriores à versão 4. Se você quiser atualizar

<sup>1</sup> Os métodos de extensão (discutidos no capítulo 10) tornam a abordagem do método auxiliar um pouco mais atraente se você estiver usando C# 3.

você mesmo sobre o significado dos termos em um nível relativamente detalhado, consulte a seção 2.2.2, mas a essência do tópico com relação aos delegados é que se seria válido (no sentido de digitação estática) chamar um e usar seu valor de retorno em qualquer lugar onde você possa invocar uma instância de um tipo de delegado específico e usar *seu* valor de retorno, esse método poderá ser usado para criar uma instância desse tipo de delegado. Isso é prolixo – é muito mais simples com exemplos.

**DIFERENTES TIPOS DE VARIÂNCIA EM DIFERENTES VERSÕES** Você já deve estar ciente de que o C# 4 oferece covariância e contravariância genéricas para delegados e interfaces. Isso é totalmente diferente da variação que estamos analisando aqui – no momento, estamos apenas lidando com a criação *de novas* instâncias de delegados. A variação genérica em C# 4 usa *conversões de referência*, que não criam novos objetos — elas apenas visualizam o objeto existente como um tipo diferente.

Veremos primeiro a contravariância e depois a covariância.

### 5.3.1 Contravariância para parâmetros delegados

Vamos considerar os manipuladores de eventos no pequeno aplicativo Windows Forms da listagem 5.1. As assinaturas dos três tipos de delegados são as seguintes:<sup>2</sup>

```
void EventHandler(remetente do objeto, EventArgs e) void
KeyPressEventHandler(remetente do objeto, KeyPressEventArgs e) void MouseEventHandler(remetente
do objeto, MouseEventArgs e)
```

Considere que KeyPressEventArgs e MouseEventArgs derivam de EventArgs (assim como muitos outros tipos – o MSDN lista 403 tipos que derivam diretamente de EventArgs no .NET 4). Se você tiver um método com um parâmetro EventArgs , poderá sempre chamá-lo com um argumento KeyPressEventArgs . Portanto, faz sentido poder usar um método com a mesma assinatura de EventHandler para criar uma instância de Key-PressEventHandler, e é exatamente isso que o C# 2 faz. Este é um exemplo de contravariância de tipos de parâmetros.

Para ver isso em ação, lembre-se da listagem 5.1 e suponha que você não precisa saber qual evento foi disparado – você só deseja anotar o fato de que um evento aconteceu. Usando conversões de grupo de métodos e contravariância, o código se torna muito mais simples, conforme mostrado na listagem a seguir.

#### Listagem 5.2 Demonstração de conversões de grupos de métodos e contravariância de delegados

```
static void LogPlainEvent(objeto remetente, EventArgs e) {
    Console.WriteLine("Ocorreu um evento");
}
...
Botão botão = novo Botão(); button.Text =
"Clique em mim"; botão.Click +=
LogPlainEvent;
```

 Usa conversão  
de grupo de métodos
 Alças  
B todos os eventos

<sup>2</sup> Removi a parte *do delegado público* por razões de espaço.

```
button.KeyPress += LogPlainEvent;
button.MouseClick += LogPlainEvent;

Formulário formulário = new Formulário();
formulário.AutoSize = verdadeiro;
formulário.Controls.Add(botão);
Aplicativo.Run(formulário);
```



Os dois métodos manipuladores que lidaram especificamente com eventos de tecla e mouse foram removidos completamente e agora você está usando um método de manipulação de eventos para tudo B. Claro, isso não é muito útil se você quiser fazer coisas diferentes para diferentes tipos de eventos, mas às vezes tudo que você precisa saber é que um evento ocorreu e, potencialmente, a origem do evento. A assinatura apenas do evento Click **C** usa a conversão implícita que discutimos na seção anterior porque possui um parâmetro EventArgs simples, mas as outras assinaturas de eventos **D** envolvem a conversão e contravariância devido aos seus diferentes tipos de parâmetros.

Mencionei anteriormente que a convenção do manipulador de eventos .NET 1.0/1.1 não fazia sentido quando foi introduzido pela primeira vez. Este exemplo mostra exatamente por que as diretrizes são mais úteis com C# 2. A convenção determina que os manipuladores de eventos devem possuir uma assinatura com dois parâmetros, sendo que o primeiro é do tipo objeto e é a origem do evento, e o segundo carrega qualquer informação extra sobre o evento. event em um tipo derivado de EventArgs. Antes da contravariância se tornar disponível, esta não foi útil - não houve benefício em fazer com que o parâmetro informativo derivasse de EventArgs, e às vezes não havia muita utilidade para a origem do evento. Isto Muitas vezes era mais sensato passar as informações relevantes diretamente na forma de parâmetros normais com tipos apropriados, assim como qualquer outro método. Agora você pode usar um método com a assinatura EventHandler como a ação para *qualquer* tipo de delegado que honre a convenção.

Até agora vimos os valores inseridos em um método ou delegado - e quanto ao valor saindo?

### 5.3.2 Covariância dos tipos de retorno delegado

Demonstrar a covariância é mais difícil, pois relativamente poucos delegados disponíveis no .NET 2.0 são declarados com um tipo de retorno não nulo e aqueles que o são tendem a retornar valor tipos. Existem alguns disponíveis, mas é mais fácil declarar seu próprio tipo de delegado que usa Stream como tipo de retorno. Para simplificar, vamos torná-lo sem parâmetros:<sup>3</sup>

```
delegar Stream StreamFactory();
```

Agora você pode usar isso com um método declarado para retornar um tipo específico de fluxo, conforme mostrado na listagem a seguir. Você declara um método que sempre retorna um MemoryStream com alguns dados sequenciais (bytes 0, 1, 2 e assim por diante até 15) e então usa isso como a ação para uma instância delegada do StreamFactory .

---

<sup>3</sup> A covariância do tipo de retorno e a contravariância do tipo de parâmetro podem ser usadas ao mesmo tempo, mas é improvável encontrar situações em que isso seria útil.

## Listagem 5.3 Demonstração de covariância de tipos de retorno para delegados

```

delegar Stream StreamFactory();

MemoryStream estático GenerateSampleData() {
    byte[] buffer = novo byte[16]; for (int i = 0; i <
    buffer.Comprimento; i++) {
        buffer[i] = (byte)i;

    } retornar novo MemoryStream(buffer);
}

Fábrica StreamFactory = GenerateSampleData;

usando (Stream stream = fábrica()) {
    dados internos;
    enquanto ((dados = stream.ReadByte()) != -1) {

        Console.WriteLine(dados);
    }
}

```

**B Declara o tipo de delegado retornando Stream**

**C MemoryStream**

**D Converte grupo de métodos com covariância**

**E Invoca delegado para obter fluxo**

A geração e exibição dos dados na listagem 5.3 só estão presentes para dar ao código algo para fazer. Os pontos importantes são as linhas anotadas. Você declara que o tipo delegado tem um tipo de retorno Stream B, mas o método GenerateSampleData tem um tipo de retorno MemoryStream C. A linha que cria a instância delegada D executa a conversão que você viu anteriormente e usa covariância de tipos de retorno para permitir GenerateSampleData para ser usado como a ação para StreamFactory. No momento em que você invoca a instância delegada E, o compilador não sabe mais que um MemoryStream será retornado – se você alterar o tipo da variável de fluxo para MemoryStream, receberá um erro de compilação.

Covariância e contravariância também podem ser usadas para construir uma instância delegada a partir de outra. Por exemplo, considere estas duas linhas de código (que assumem um método HandleEvent apropriado):

```

EventHandler geral = novo EventHandler(HandleEvent);
Chave KeyPressEventHandler = new KeyPressEventHandler (geral);

```

A primeira linha é válida em C# 1, mas a segunda não – para construir um delegado a partir de outro em C# 1, as assinaturas dos dois tipos de delegado envolvidos devem corresponder. Por exemplo, você poderia criar um MethodInvoker a partir de um ThreadStart, mas não poderia criar um KeyPressEventHandler a partir de um EventHandler, conforme mostrado na segunda linha. Você está usando a contravariância para criar uma nova instância de delegado a partir de uma existente com uma assinatura de tipo de delegado *compatível*, onde a compatibilidade é definida de maneira menos restritiva no C# 2 do que no C# 1.

Tudo isto é positivo, exceto por uma pequena mosca na sopa.

### 5.3.3 Um pequeno risco de incompatibilidade

Essa nova flexibilidade no C# 2 cria um dos poucos casos em que o código C# 1 válido existente pode produzir resultados diferentes quando compilado em C# 2. Suponha que uma classe derivada sobrecarregue um método declarado em sua classe base e você tente criar uma instância de um delegar usando uma conversão de grupo de métodos. Uma conversão que anteriormente correspondia apenas ao método da classe base poderia corresponder ao método da classe derivada devido à covariância ou contravariância em C# 2, caso em que esse método de classe derivada seria escolhido pelo compilador. A listagem a seguir dá um exemplo disso.

#### Listagem 5.4 Demonstração de alterações significativas entre C# 1 e C# 2

```
delegado void SampleDelegate(string x);
public void CandidateAction(string x) {
    Console.WriteLine("Snippet.CandidateAction");
}
classe pública Derivado: Snippet {
    public void CandidateAction(objeto o) {
        Console.WriteLine("Derivado.CandidateAction");
    }
}
...
Derivado x = novo Derivado(); Fábrica
SampleDelegate = new SampleDelegate(x.CandidateAction); Teste de fábrica";
```

Lembre-se de que Snippy<sup>4</sup> gerará todo esse código dentro de uma classe chamada Snippet, da qual deriva o tipo aninhado. Em C# 1, a listagem 5.4 imprimiria Snippet.CandidateAction porque o método que utiliza um parâmetro de objeto não era compatível com SampleDelegate. Em C# 2, o método é compatível, e é o método escolhido por ser declarado em um tipo mais derivado, então o resultado é que Derived.CandidateAction é impresso.

Felizmente, o compilador C# 2 sabe que esta é uma alteração significativa e emite um aviso apropriado. Inclui esta seção porque você deve estar ciente da possibilidade de tal problema, mas tenho certeza de que raramente é encontrado na vida real.

Chega de desgraça e tristeza sobre uma possível ruptura. Ainda precisamos ver o novo recurso mais importante em relação aos delegados: métodos anônimos. Eles são um pouco mais complicados do que os tópicos que abordamos até agora, mas também são *muito* poderosos — e um grande passo em direção ao C# 3.

---

<sup>4</sup> Caso você tenha pulado o primeiro capítulo, Snippy é uma ferramenta que desenvolvi para criar exemplos de código curtos, mas completos. Consulte a seção 1.8.1 para obter mais detalhes.

## 5.4 Ações de delegação inline com métodos anônimos

No C# 1, era comum implementar um delegado com uma assinatura específica, mesmo que você já tivesse um método com exatamente o comportamento correto, mas com um conjunto de parâmetros ligeiramente diferente. Da mesma forma, muitas vezes você gostaria que um delegado fizesse apenas uma coisa minúscula, mas isso significava que você precisava de um método totalmente extra. O novo método representaria um comportamento que só era relevante dentro do método original, mas agora estava exposto para toda a classe, criando ruído no IntelliSense e geralmente atrapalhando.

Tudo isso foi intensamente frustrante. Os recursos de covariância e contravariância de que acabamos de falar podem às vezes ajudar no primeiro problema, mas muitas vezes não o fazem. *Métodos anônimos*, que também são novos no C# 2, sempre podem ajudar com esses problemas.

Informalmente, os métodos anônimos permitem especificar a ação para uma instância delegada inline como parte da expressão de criação da instância delegada. Eles também fornecem comportamentos muito mais poderosos na forma de *encerramentos*, mas falaremos deles na seção 5.5. Por enquanto, vamos nos ater a coisas relativamente simples.

Primeiro veremos exemplos de métodos anônimos que aceitam parâmetros, mas não retornam nenhum valor; em seguida, exploraremos a sintaxe envolvida no fornecimento de valores de retorno e um atalho disponível quando você não precisar usar os valores de parâmetro passados para você.

### 5.4.1 Começando de forma simples: agindo sobre um parâmetro

O .NET 2.0 introduziu um tipo de delegado genérico chamado `Action<T>`, que usaremos em nossos exemplos. Sua assinatura é simples (além de ser genérica):

```
delegado público void Action<T>(T obj)
```

Em outras palavras, um `Action<T>` faz algo com um valor do tipo `T`; por exemplo, um `Action<string>` poderia reverter a string e imprimi-la, um `Action<int>` poderia imprimir a raiz quadrada do número passado para ele e um `Action<IList<double>>` poderia encontrar a média de todos os números dados a ele e imprima-os. Por total coincidência, todos esses exemplos são implementados usando métodos anônimos na listagem a seguir.

#### Listagem 5.5 Métodos anônimos usados com o tipo delegado `Action<T>`

```
Action<string> printReverse = delegar(string texto)
{
    char[] chars = text.ToCharArray(); Array.Reverse(characters);
    Console.WriteLine(nova
        string(characters));
};

Action<int> printRoot = delegado(int número)
{
    Console.WriteLine(Math.Sqrt(número));
};
```

**B** Usa método anônimo para criar `Ação<string>`

```
Action<IList<double>> printMean = delegado(IList<double> números) {
    total duplo = 0; foreach
    (valor duplo em números) {
        total += valor;
    }
    Console.WriteLine(total/números.Count);
};

printReverse("Olá mundo"); imprimirRoot(2);
printMean(new
duplo[] { 1,5, 2,5, 3, 4,5 });


```



Usa loop em  
método  
anônimo

D Invoca delegados normalmente

A Listagem 5.5 mostra alguns dos diferentes recursos dos métodos anônimos. Primeiro, há a sintaxe dos métodos anônimos: use a palavra-chave delegada , seguida dos parâmetros (se houver), seguido do código para a ação da instância delegada, em um bloco. O código de reversão de string **B** mostra que o bloco pode conter declarações de variáveis locais, e o código de média de lista **C** demonstra o loop dentro do bloco.

Basicamente, você pode fazer (quase) qualquer coisa em um método anônimo que você pode fazer em um corpo de método normal. Da mesma forma, o resultado de um método anônimo é uma instância delegada que pode ser usada como qualquer outra **D**. Mas esteja avisado que a contravariância não se aplica a métodos anônimos; você deve especificar os tipos de parâmetros que correspondem exatamente ao tipo de delegado.

**ALGUMAS RESTRIÇÕES...** Uma pequena estranheza é que se você estiver escrevendo um método anônimo em um tipo de valor, você não poderá referenciar isso de dentro dele. Não existe tal restrição dentro de um tipo de referência. Além disso, nas implementações do compilador Microsoft C# 2 e 3, o acesso a um membro base dentro de um método anônimo por meio da palavra-chave base resultou em um aviso de que o código resultante não era verificável. Isso foi corrigido no compilador C# 4.

Em termos de implementação, você ainda está criando um método em IL para cada método anônimo no código-fonte. O compilador gerará um método dentro da classe existente e o utilizará como ação ao criar a instância delegada, como se fosse um método normal.<sup>5</sup> O CLR não sabe nem se importa com o uso de um método anônimo. Você pode ver os métodos extras no código compilado usando ildasm ou Reflector. (O Reflector sabe como interpretar o IL para exibir métodos anônimos no método que os utiliza, mas os métodos extras ainda são visíveis.) Esses métodos têm nomes *indizíveis* — aqueles que são válidos em IL, mas inválidos em C#. Isso impede que você tente referenciá-los diretamente em seu código C# e evita a possibilidade de colisões de nomes. Muitos dos recursos do C# 2 e versões posteriores são implementados de maneira semelhante; uma maneira fácil de identificá-los é que eles geralmente contêm colchetes angulares. Por exemplo, um método anônimo em um método Main pode fazer com que um método chamado <Main>b\_\_0 seja criado. No entanto, é totalmente específico da implementação. Microsoft

---

<sup>5</sup> Você verá na seção 5.5.4 que embora sempre haja um novo método, ele nem sempre é criado onde você espera.

poderia alterar suas convenções privadas em uma versão futura, por exemplo. Isso não deveria quebrar nada, pois nada deveria depender desses nomes.

Vale ressaltar neste estágio que a listagem 5.5 é explodida em comparação com a aparência normal dos métodos anônimos em código real. Freqüentemente, você os verá usados como argumentos para outro método (em vez de atribuídos a uma variável do tipo delegado) e com poucas quebras de linha - afinal, a compactação é parte da razão para usá-los. Para demonstrar isso, usaremos o método `List<T>.ForEach` que recebe um `Action<T>` como parâmetro e executa essa ação em cada elemento. A listagem a seguir mostra um exemplo extremo, aplicando a mesma ação de raiz quadrada usada na listagem 5.5, mas de forma compacta.

#### Listagem 5.6 Exemplo extremo de compactação de código. Aviso: código ilegível à frente!

```
Lista<int> x = new Lista<int>(); x.Adicionar(5);
x.Adicionar(10);
x.Adicionar(15);
x.Adicionar(20);
x.Adicionar(25);

x.ForEach(delegate(int n){Console.WriteLine(Math.Sqrt(n));});
```

Isso é horrível - especialmente quando os últimos seis caracteres parecem estar ordenados quase aleatoriamente. Existe um meio-termo, é claro. Tenho tendência a quebrar minha regra usual de "colchetes em uma linha por conta própria" para métodos anônimos (como faço para propriedades triviais), mas ainda permito uma quantidade razoável de espaços em branco. Eu poderia muito bem escrever a última linha da listagem 5.6 em uma destas duas formas:

```
x.ForEach(delegate(int n)
    {Console.WriteLine(Math.Sqrt(n)); })
;

x.ForEach(delegate(int n)
    { Console.WriteLine(Math.Sqrt(n)); });
```

Apenas adicionar espaços à listagem 5.6 já teria ajudado. Em cada um desses formatos, os parênteses e colchetes agora são menos confusos e a parte do que faz se destaca adequadamente. É claro que o modo como você espaça seu código é da sua conta, mas recomendo que você pense ativamente sobre onde deseja encontrar o equilíbrio e converse sobre isso com seus colegas de equipe para tentar alcançar alguma consistência. Porém, a consistência nem *sempre* leva ao código mais legível - às vezes, manter tudo em uma linha é o formato mais direto.

Até agora a única interação que você teve com o código de chamada foi através de parâmetros. E quanto aos valores de retorno?

## 5.4.2 Retornando valores de métodos anônimos

O delegado Action<T> tem um tipo de retorno void , então você ainda não precisou retornar nada dos seus métodos anônimos. Para demonstrar como você pode fazer isso quando necessário, usaremos o tipo de delegado Predicate<T> do .NET 2.0, que possui esta assinatura:

```
delegado público bool Predicate<T>(T obj)
```

A listagem a seguir mostra um método anônimo criando uma instância de Predicate<T> para retornar se o argumento passado é ímpar ou par. Predicados geralmente são usados na filtragem e correspondência — você poderia usar o código nesta listagem para filtrar uma lista apenas para os elementos pares, por exemplo.

### Listagem 5.7 Retornando um valor de um método anônimo

```
Predicado<int> isEven = delegado(int x) { return x % 2 == 0; };

Console.WriteLine(isEven(1));
Console.WriteLine(isEven(4));
```

A nova sintaxe é quase certamente o que você esperava: você retorna o valor apropriado como se o método anônimo fosse um método normal. Você pode esperar ver um tipo de retorno declarado próximo à palavra-chave delegada , mas não há necessidade. O compilador verifica se todos os valores de retorno possíveis são compatíveis com o tipo de retorno declarado do tipo delegado no qual está tentando converter o método anônimo.

**APENAS DO QUE VOCÊ ESTÁ VOLTANDO?** Quando você retorna um valor de um método anônimo, na verdade ele está retornando apenas do método anônimo — não está retornando do método que cria a instância delegada.

É fácil examinar algum código, ver a palavra-chave return e pensar que é um ponto de saída do método atual, então tome cuidado.

Como mencionei antes, relativamente poucos delegados no .NET 2.0 retornam valores, embora, como você verá na parte 3 deste livro, o .NET 3.5 use essa ideia *com muito* mais frequência, especialmente com o LINQ. Porém , há outro tipo de delegado razoavelmente comum no .NET 2.0: Comparison<T>, que pode ser usado ao classificar coleções. É o equivalente delegado da interface IComparer<T> . Muitas vezes você só precisa de uma ordem de classificação específica em uma situação, então faz sentido ser capaz de especificar essa ordem in-line, em vez de expô-la como um método dentro do resto da classe. A listagem a seguir demonstra isso, imprimindo os arquivos no diretório C:\, ordenando-os primeiro por nome e depois (separadamente) por tamanho.

### Listagem 5.8 Usando métodos anônimos para classificar arquivos de maneira simples

```
static void SortAndShowFiles(string title, Comparison<FileInfo> sortOrder) {

    FileInfo[] arquivos = new DirectoryInfo(@"C:\").GetFiles();

    Array.Sort(arquivos,sortOrder);
```

```

Console.WriteLine(título); foreach (arquivo
FileInfo em arquivos) {

    Console.WriteLine (" {0} ({1} bytes)", arquivo.Nome, arquivo.Length);
}

}

...

SortAndShowFiles("Ordenado por nome:", delegado(FileInfo f1, FileInfo f2)
{ return f1.Nome.CompareTo(f2.Nome); }

);

SortAndShowFiles("Ordenado por comprimento:", delegado(FileInfo f1, FileInfo f2)
{ return f1.Length.CompareTo(f2.Length); }

);

```

Se você não estivesse usando métodos anônimos, precisaria de um método separado para cada ordem de classificação. Em vez disso, a listagem 5.8 deixa claro o que você classificará em cada caso, exatamente onde chamar SortAndShowFiles. (Às vezes você chamará Sort diretamente no ponto em que o método anônimo é chamado. Na listagem 5.8, você está executando a mesma sequência de busca/classificação/exibição duas vezes, apenas com ordens de classificação diferentes, então encapsulei essas etapas em seu próprio método.)

Às vezes, um atalho sintático especial é aplicável. Se você não se importa com os parâmetros de um delegado, não precisa declará-los. Vamos ver como isso funciona.

### 5.4.3 Ignorando parâmetros delegados

Ocasionalmente, você deseja implementar um delegado que não dependa dos valores de seus parâmetros. Você pode querer escrever um manipulador de eventos cujo comportamento seja apropriado apenas para um evento e não dependa dos argumentos do evento – salvando o trabalho do usuário, por exemplo. Os manipuladores de eventos do exemplo da listagem 5.1 se ajustam perfeitamente a esse critério. Nesse caso, você pode omitir totalmente a lista de parâmetros, usando apenas a palavra-chave delegada e depois um bloco de código como ação para o método. A listagem a seguir é equivalente à listagem 5.1, mas usa a sintaxe mais curta.

#### Listagem 5.9 Assinando eventos com métodos anônimos que ignoram parâmetros

```

Botão botão = novo Botão(); button.Text = "Clique
em mim"; button.Click += delegado
{ Console.WriteLine("LogPlain"); };
button.KeyPress += delegado { Console.WriteLine("LogKey"); }; button.MouseClick += delegado
{ Console.WriteLine("LogMouse"); };

Formulário formulário = new
Formulário(); formulário.AutoSize
= verdadeiro; formulário.Controls.Add(botão);
Aplicativo.Run(formulário);

```

Normalmente você teria que escrever cada assinatura assim:

```
button.Click += delegado(objeto remetente, EventArgs e) { ... };
```

Isso desperdiça muito espaço por pouca razão – você não precisa dos valores dos parâmetros, então o compilador permite que você não os especifique.

Achei esse atalho muito útil quando se trata de implementar meus próprios eventos. Por exemplo, fico cansado de ter que realizar uma verificação de nulidade antes de gerar um evento. Uma maneira de contornar isso é garantir que o evento comece com um manipulador, que nunca será removido. Contanto que o manipulador não faça nada, tudo que você perde é um pouquinho de desempenho. Antes do C# 2, você tinha que criar explicitamente um método com a assinatura correta, o que geralmente não valia o benefício, mas agora você pode escrever código como este:

```
evento público EventHandler Click = delegado {};
```

A partir daí, você pode simplesmente chamar Click sem nenhum teste de nulidade.

Você deve estar ciente de uma armadilha relacionada a esse recurso de curva de parâmetro — se o método anônimo puder ser convertido em vários tipos de delegado (por exemplo, para chamar diferentes sobrecargas de método), o compilador precisará de mais ajuda. Para mostrar o que quero dizer, vamos usar o mesmo exemplo problemático que vimos com conversões de grupos de métodos: iniciar um novo thread. Existem quatro construtores de thread no .NET 2.0:

```
thread público (início ParameterizedThreadStart) Thread público (início  
ThreadStart) Thread público (início  
ParameterizedThreadStart, int maxStackSize) Thread público (início ThreadStart, int maxStackSize)
```

Estes são os dois tipos de delegado envolvidos:

```
delegado público void ThreadStart() delegado público  
void ParameterizedThreadStart (objeto obj)
```

Agora, considere as três tentativas a seguir para criar um novo thread:

```
novo Thread(delegado) {Console.WriteLine("t1"); } ); new  
Thread(delegate(objeto o) { Console.WriteLine("t2"); } ); novo Thread(delegado  
{ Console.WriteLine("t3"); } );
```

A primeira e a segunda linhas contêm listas de parâmetros – o compilador sabe que não pode converter o método anônimo da primeira linha em ParameterizedThreadStart ou converter o método anônimo da segunda linha em ThreadStart. Essas linhas são compiladas porque há apenas uma sobrecarga de construtor aplicável em cada caso. A terceira linha, porém, é ambígua — o método anônimo pode ser convertido em qualquer tipo de delegado, portanto, ambas as sobrecargas do construtor de parâmetro único são aplicáveis. Nessa situação, o compilador levanta as mãos e emite um erro. Você pode resolver isso especificando explicitamente a lista de parâmetros ou convertendo o método anônimo para o tipo de delegado correto.

Esperamos que o que você viu sobre métodos anônimos até agora tenha provocado alguma reflexão sobre seu próprio código e feito você considerar onde poderia usar essas técnicas com bons resultados. Na verdade, mesmo que métodos anônimos só pudessem fazer o que você já viu, eles seriam muito úteis. Mas os métodos anônimos envolvem mais do que apenas evitar a inclusão de um método extra em seu código. Métodos anônimos são

Implementação em C# 2 de um recurso conhecido em outros lugares como *fechamentos* por meio de *variáveis capturadas*. A próxima seção explica esses dois termos e mostra como os métodos anônimos podem ser extremamente poderosos — e confusos se você não tomar cuidado.

## 5.5 Capturando variáveis em métodos anônimos

Não gosto de ter que dar avisos, mas acho que faz sentido incluir um aqui: se este tópico é novo para você, então não comece esta seção até que você se sinta razoavelmente acordado e tenha um pouco de tempo para gastar nisso. Não quero alarmá-lo desnecessariamente, e você deve se sentir confiante de que não há nada aqui tão insanamente complicado que você não será capaz de entendê-lo com um pouco de esforço. É só que variáveis capturadas podem ser um pouco confusas no início, em parte porque eles derrubam alguns dos seus conhecimento e intuição existentes.

Mas continue com isso! O retorno pode ser *enorme* em termos de simplicidade do código e legibilidade. Este tópico também será crucial quando olharmos para expressões lambda e LINQ em C# 3, então vale a pena o investimento.

Vamos começar com algumas definições.

### 5.5.1 Definindo fechamentos e diferentes tipos de variáveis

O conceito de *encerramentos* é antigo, implementado pela primeira vez no Scheme, mas vem ganhando mais destaque nos últimos anos, à medida que mais linguagens convencionais o adotam. A ideia básica é que uma função<sup>6</sup> é capaz de interagir com um ambiente além dos parâmetros fornecidos a ele. Isso é tudo em termos abstratos, mas para entender como isso se aplica ao C# 2, precisamos de mais alguns termos:

- ÿ Uma *variável externa* é uma variável ou parâmetro local (excluindo os parâmetros `ref` e `out`) cujo escopo inclui um método anônimo. Esta referência também conta como uma variável externa de qualquer método anônimo dentro de um membro de instância de uma classe.

- ÿ Uma *variável externa capturada* (geralmente abreviada para *variável capturada*) é uma variável externa usada em um método anônimo. Voltando aos encerramentos, a parte da função é o método anônimo, e o ambiente com o qual ele pode interagir é o conjunto de variáveis por ele capturadas.

Tudo isso é muito seco e pode ser difícil de imaginar, mas o principal objetivo é que um método anônimo pode usar variáveis locais definidas no mesmo método que o declara. Esse pode não parecer grande coisa, mas em muitas situações é extremamente útil - você pode usar informações contextuais que você tem em mãos, em vez de ter que configurar informações extras apenas para armazenar dados que você já conhece. Veremos alguns exemplos concretos úteis em breve, eu prometo — mas primeiro vale a pena dar uma olhada em algum código para esclarecer essas definições.

A Listagem 5.10 fornece um exemplo com diversas variáveis locais, e é uma única método, portanto ele não pode ser executado sozinho. Não vou explicar como funcionaria ou

---

<sup>6</sup> Esta é a terminologia geral da ciência da computação, não a terminologia C#.

o que faria ainda; Eu só quero discutir como as diferentes variáveis são classificadas. Novamente, usaremos o tipo delegado MethodInvoker para simplificar.

#### Listagem 5.10 Exemplos de tipos de variáveis com relação a métodos anônimos

```
void EnclosingMethod() {
    int variávelexterna = 5; string
    captureVariable = "capturado";
    if (DateTime.Now.Hour == 23) {

        int normalLocalVariable = DateTime.Now.Minute;
        Console.WriteLine(normalLocalVariable);
    }

    MethodInvoker x = delegado() {
        string anonLocal = "local para método anônimo"; Console.WriteLine(capturedVariable
        + anonLocal);
    };
    x();
}
```

**B** Variável externa  
(não capturada)

**C** Variável externa capturada  
por método anônimo

**D** Variável local de  
método normal

**E** Variável local do  
método anônimo

**F** Captura de  
variável externa

Vamos passar por todas as variáveis, das mais simples às mais complicadas:

ÿ **normalLocalVariable D** não é uma variável externa porque não existem métodos anônimos dentro de seu escopo. Ele se comporta exatamente da mesma maneira que as variáveis locais sempre se comportaram.

ÿ **anonLocal E** também não é uma variável externa, mas é local para o método anônimo, não para EnclosingMethod. Ele só existirá (em termos de estar presente em um quadro de pilha em execução) quando a instância delegada for invocada.

ÿ **outerVariable B** é uma variável externa porque o método anônimo é declarado dentro de seu escopo. Mas o método anônimo não se refere a isso, portanto não é capturado.

ÿ A variável capturada **C** é uma variável externa porque o método anônimo é declarado dentro de seu escopo e é *capturado* em virtude de ser usado em F.

Ok, agora você entende a terminologia, mas não estamos muito mais perto de ver o que as variáveis capturadas fazem. Suspeito que você poderia adivinhar o resultado se executasse o método da listagem 5.10, mas há alguns outros casos que provavelmente o surpreenderiam.

Começaremos com um exemplo simples e avançaremos para outros mais complexos.

#### 5.5.2 Examinando o comportamento das variáveis capturadas

Quando uma variável é capturada, na verdade é a *variável* que é capturada pelo método anônimo, e não seu valor no momento em que a instância delegada foi criada. Você verá mais tarde que isso tem consequências de longo alcance, mas primeiro você precisa entender o que isso significa para uma situação relativamente simples.

A listagem a seguir possui uma variável capturada e um método anônimo que imprime e altera a variável. Você verá que as alterações feitas na variável fora do método anônimo são visíveis dentro do método anônimo e vice-versa.

#### Listagem 5.11 Acessando uma variável dentro e fora de um método anônimo

```
string capturado = "antes de x ser criado";  
  
MethodInvoker x = delegado {  
  
    Console.WriteLine(capturado); capturado =  
    "alterado por x";  
};  
  
capturado = "diretamente antes de x ser invocado"; x();  
  
  
Console.WriteLine(capturado);  
  
capturado = "antes da segunda invocação"; x();
```

A saída da listagem 5.11 é a seguinte:

```
diretamente antes de x ser invocado alterado  
por x antes da  
segunda invocação
```

Vejamos como isso acontece. Primeiro, você declara a variável capturada e define seu valor com uma string literal perfeitamente normal. Até agora, não há nada de especial sobre a variável. Você então declara x e define seu valor usando um método anônimo que captura as capturas . A instância delegada sempre imprimirá o valor atual capturado e então o definirá como “alterado por x”. Não esqueça que criar esta instância delegada *não a executa*.

Para deixar absolutamente claro que apenas criar a instância delegada não lê a variável e armazena seu valor em algum lugar, agora você altera o valor de capturado para “diretamente antes de x ser invocado”. Você então invoca x pela primeira vez. Ele lê o valor capturado e o imprime – a primeira linha da saída. Ele define o valor de capturado como “alterado por x” e retorna. Quando a instância delegada retorna, o método normal continua da maneira usual. Ele imprime o valor atual capturado, fornecendo a segunda linha de saída.

O método normal então altera o valor de capture mais uma vez (desta vez para “antes da segunda invocação”) e invoca x pela segunda vez. O valor atual capturado é impresso, fornecendo a última linha de saída. A instância delegada altera o valor de capturado para “alterado por x” e retorna, ponto em que o método normal ficou sem código e pronto.

São muitos detalhes sobre como um pequeno trecho de código funciona, mas há apenas uma ideia crucial nele: a variável *capturada* é a mesma que o restante do método usa. Para algumas pessoas, isso é difícil de entender; para outros, isso acontece naturalmente. Não se preocupe se for complicado no início – ficará mais fácil com o tempo.

Mesmo que você tenha entendido tudo facilmente até agora, você pode estar se perguntando por que quero fazer nada disso. Já era hora de termos um exemplo que fosse realmente útil.

### 5.5.3 Qual é o sentido das variáveis capturadas?

Simplificando, as variáveis capturadas eliminam a necessidade de escrever classes extras apenas para armazenar as informações que um delegado precisa para agir, além do que é passado por meio de parâmetros. Antes de ParameterizedThreadStart existir, se você quisesse iniciar um novo thread (não threadpool) e fornecer algumas informações - o URL de uma página a ser buscada, por exemplo - você teria que criar um tipo extra para armazenar o URL e colocar a ação de a instância delegada ThreadStart nesse tipo. Mesmo com ParameterizedThreadStart, seu método teve que aceitar um parâmetro do tipo object e convertê-lo no tipo que você realmente queria. Foi uma maneira feia de conseguir algo que deveria ser simples.

Como outro exemplo, suponha que você tenha uma lista de pessoas e queira escrever um método que retorne uma segunda lista contendo todas as pessoas menores de uma determinada idade. List<T> possui um método chamado FindAll que retorna outra lista de tudo que corresponde ao predicado especificado. Antes de métodos anônimos e variáveis capturadas, não faria muito sentido que List<T>.FindAll existisse, por causa de todos os obstáculos que você teria que passar para criar o delegado certo para começar. Teria sido mais simples fazer toda a iteração e copiar manualmente. Com o C# 2, porém, você pode fazer tudo isso facilmente:

```
List<Pessoa> FindAllYoungerThan(List<Pessoa> pessoas, limite interno) {
    return pessoas.FindAll(delegate (Pessoa pessoa) { return pessoa.Idade <
        limite; });
}
```

Aqui você está capturando o parâmetro limit dentro da instância delegada - se você tivesse métodos anônimos, mas não variáveis capturadas, você poderia ter realizado um teste contra um limite codificado, mas não um que foi passado para o método como um parâmetro. ter. Espero que você concorde que essa abordagem é interessante: ela expressa exatamente o que você deseja fazer com muito menos confusão sobre como exatamente *isso* deveria acontecer do que você veria em uma versão C# 1. (É ainda mais claro em C# 3, admito...)<sup>7</sup> É relativamente raro você se deparar com uma situação em que precisa escrever em uma variável capturada, mas, novamente, isso pode ter sua utilidade.

Ainda comigo? Bom. Até agora, você usou apenas a instância delegada dentro do método que a cria. Isso não levanta muitas questões sobre o tempo de vida das variáveis capturadas — mas o que aconteceria se a instância delegada escapasse para o grande mundo mau? Como ele reagiria depois que o método que o criou terminasse?

---

<sup>7</sup> Caso você esteja se perguntando: return people.Where(person => person.Age < limit);

### 5.5.4 A vida útil prolongada das variáveis capturadas

A maneira mais simples de abordar este tópico é estabelecer uma regra, dar um exemplo e depois pensar no que aconteceria se a regra não existisse. Aqui vamos nós:

*Uma variável capturada dura pelo menos enquanto qualquer instância delegada referente a ela.*

Não se preocupe se ainda não fizer muito sentido – é para isso que serve o exemplo. A listagem a seguir mostra um método que *retorna* uma instância delegada. Essa instância delegada é criada usando um método anônimo que captura uma variável externa. Então, o que acontecerá quando o delegado for invocado após o retorno do método?

#### Listagem 5.12 Demonstração de uma variável capturada tendo seu tempo de vida estendido

```
static MethodInvoker CreateDelegateInstance() {
    contador interno = 5;

    MethodInvoker ret = delegado {
        Console.WriteLine(contador); contador++;
    };
    ret();
    retornar ret;
}
...
MethodInvoker x = CreateDelegateInstance(); x(); x();
```

A saída da listagem 5.12 consiste nos números 5, 6 e 7 em linhas separadas. A primeira linha da saída vem da invocação da instância delegada em *Create-DelegateInstance*, portanto, faz sentido que o valor do contador esteja disponível nesse ponto. Mas e depois que o método retornar?

Normalmente você consideraria que o contador estava na pilha, então quando o quadro de pilha para *CreateDelegateInstance* for destruído, você esperaria que o contador desaparecesse efetivamente... e ainda assim as invocações subsequentes da instância delegada parecem continuar a usá-lo.

O segredo é desafiar a suposição de que o contador está na pilha em primeiro lugar. Não é. Na verdade, o compilador criou uma classe extra para armazenar a variável.

O método *CreateDelegateInstance* tem uma referência a uma instância dessa classe para que possa usar o contador, e o delegado tem uma referência à mesma instância, que reside no heap da maneira normal. Essa instância não é elegível para coleta de lixo até que o delegado esteja pronto para ser coletado.

Alguns aspectos dos métodos anônimos são muito específicos do compilador (diferentes compiladores poderiam alcançar a mesma semântica de maneiras diferentes), mas é difícil ver como o comportamento especificado poderia ser alcançado sem usar uma classe extra para *armazenar a variável capturada*. Observe que se você apenas capturar isso, nenhum tipo extra será necessário — o compilador apenas cria um método de instância para atuar como a ação do delegado. Como mencionei antes, você provavelmente não deveria se preocupar muito com os detalhes da pilha e do heap, mas

vale a pena saber o que o compilador é capaz de fazer, caso você fique confuso sobre como o comportamento especificado é possível.

Ok, então as variáveis locais podem permanecer mesmo após o retorno de um método. Você pode estar se perguntando o que eu poderia oferecer a seguir: que tal vários delegados capturando diferentes instâncias da mesma variável? Parece loucura, então é exatamente o tipo de coisa que você deveria estar esperando agora.

### 5.5.5 Instanciações de variáveis locais

Em um dia bom, as variáveis capturadas agem exatamente da maneira que eu esperava à primeira vista. Num dia ruim, ainda fico surpreso quando não tomo cuidado. Quando há problemas, quase sempre é porque esqueci quantas “instâncias” de variáveis locais estou realmente criando. Diz-se que uma variável local é *instanciada* cada vez que a execução entra no escopo onde foi declarada.

Aqui está um exemplo simples comparando dois pedaços de código muito semelhantes:

<pre>interno único; for (int i = 0; i &lt; 10; i++) {     único = 5;     Console.WriteLine(único + i); }</pre>	<pre>for (int i = 0; i &lt; 10; i++) {     int múltiplo = 5;     Console.WriteLine(múltiplo + i); }</pre>
--	---

Nos velhos tempos, era razoável dizer que trechos de código como esse eram semanticamente idênticos. Na verdade, eles geralmente compilariam na mesma IL – e ainda o farão, se não houver métodos anônimos envolvidos. Todo o espaço para variáveis locais é alocado na pilha no início do método, portanto não há custo para redeclarar a variável para cada iteração do loop.<sup>8</sup> Em nossa nova terminologia, a única variável será instanciada apenas uma vez, mas a variável múltipla será instanciada 10 vezes — é como se houvesse 10 variáveis locais, todas chamadas múltiplas, que foram criadas uma após a outra.

Tenho certeza de que você pode ver onde estou indo: quando uma variável é capturada, é a “instância” relevante da variável que é capturada. Se você capturasse vários dentro do loop, a variável capturada na primeira iteração seria diferente da variável capturada na segunda vez e assim por diante. A listagem a seguir mostra exatamente esse efeito.

#### Listagem 5.13 Capturando múltiplas instanciações de variáveis com múltiplos delegados

```
List<MethodInvoker> list = new List<MethodInvoker>();
for (int índice = 0; índice < 5; índice++) {
```

contador interno = índice \* 10;

 **B** Instancia o contador

<sup>8</sup> Na minha opinião, também é mais limpo declarar novamente a variável, a menos que você precise explicitamente manter seu valor entre iterações.

```

lista.Add(delegado {

    Console.WriteLine(contador);
    contador++;
});

}

foreach (MethodInvoker t na lista) {
    t();
}

} lista[0](); lista[0]
(); lista[0]();
();

lista[1]);

```

A Listagem 5.13 cria cinco instâncias de delegação C diferentes – uma para cada vez que você percorre o loop. Invocar o delegado imprimirá o valor do contador e depois o incrementará. Como o contador é declarado *dentro* do loop, ele é instanciado para cada iteração B e cada delegado captura uma variável diferente. Quando você percorre e invoca cada delegado D, você vê os diferentes valores inicialmente atribuídos ao contador: 0, 10, 20, 30, 40. Apenas para esclarecer, quando você volta para a primeira instância do delegado e a executa mais três vezes E, ele continua de onde a variável de contador daquela instância parou: 1, 2, 3. Finalmente você executa a segunda instância delegada F, e ela continua de onde a variável de contador *daquela* instância estava

parou: 11.

Como você pode ver, cada uma das instâncias delegadas capturou uma variável diferente. Antes de deixarmos este exemplo, devo salientar o que teria acontecido se você tivesse capturado o índice – a variável declarada pelo loop for – em vez do contador. Neste caso, todos os delegados teriam compartilhado a mesma variável. A saída seriam os números de 5 a 13; 5 primeiro porque a última atribuição ao índice antes do término do loop o definiria como 5 e, em seguida, incrementaria a mesma variável, independentemente de qual delegado estivesse envolvido. Você veria o mesmo comportamento com um loop foreach (em C# 2–4): a variável declarada pela parte inicial do loop é instanciada apenas uma vez.

É fácil errar! Se você quiser capturar o valor de uma variável de loop para aquela iteração específica do loop, introduza outra variável dentro do loop, copie o valor da variável de loop nela e capture essa nova variável — efetivamente o que você fez na Listagem 5.13 com o contador variável.

**ISSO MUDA NO C# 5...** Embora o comportamento em um loop for seja razoável - afinal, a variável parece ter sido declarada apenas uma vez - é mais surpreendente no caso foreach . Na verdade, *quase sempre* é errado capturar uma variável de iteração foreach em um método anônimo que existirá além da iteração imediata. (Tudo bem se a instância delegada for usada apenas dentro dessa iteração.) Isso causou problemas para tantos desenvolvedores que a equipe do C# mudou a semântica do foreach para o C# 5 para fazê-lo agir de forma mais natural – como se cada iteração tivesse sua própria variável separada. Consulte a seção 16.1 para obter mais detalhes.

Para nosso exemplo final, vejamos algo realmente desagradável: compartilhar algumas variáveis capturadas, mas não outras.

#### 5.5.6 Misturas de variáveis compartilhadas e distintas

Antes de mostrar o próximo exemplo, deixe-me dizer que *não* é um código que eu recomendaria. Na verdade, o objetivo de apresentá-lo é mostrar como, se você tentar usar variáveis capturadas de uma maneira muito complicada, as coisas podem ficar complicadas muito rapidamente. A listagem a seguir cria duas instâncias de delegação, cada uma capturando “as mesmas” duas variáveis. Mas a história fica mais complicada quando você olha o que realmente foi capturado.

##### Listagem 5.14 Capturando variáveis em diferentes escopos. Aviso: código desagradável à frente!

```
delegados MethodInvoker[] = new MethodInvoker[2];

int fora = 0;                                     ← B Instancia a variável uma vez

for (int i = 0; i < 2; i++) {

    int dentro = 0;                               ← C instancia variável várias vezes

    delegados[i] = delegado {
        Console.WriteLine ("({0},{1})", fora, dentro); fora++; dentro++;
    };
}

MethodInvoker primeiro = delegados[0];
MethodInvoker segundo = delegados[1];

primeiro();
primeiro();
primeiro();

segundo();
segundo();
```

Quanto tempo você levaria para prever o resultado da listagem 5.14 (mesmo com as anotações)? Francamente, demoraria um pouco - mais do que gostaria para entender o código. Porém, apenas como exercício, vejamos o que acontece.

Primeiro considere a variável externa B. O escopo no qual ela é declarada só é inserido uma vez, então é um caso simples — só existe uma, efetivamente. A variável interna C é uma questão diferente – cada iteração do loop instancia uma nova. Isso significa que quando você cria a instância delegada D, a variável externa é compartilhada entre as duas instâncias delegadas, mas cada uma delas tem sua própria variável interna .

Após o término do loop, você chama a primeira instância delegada criada três vezes. Como ele está incrementando ambas as variáveis capturadas a cada vez, e ambas começaram como 0, você vê (0,0), depois (1,1) e depois (2,2). A diferença entre as duas variáveis em termos de escopo torna-se aparente quando você executa a segunda instância delegada. Tem uma variável interna diferente , então ainda tem sua inicial

valor de 0, mas a variável externa é aquela que você já incrementou três vezes.

A saída da chamada duas vezes ao segundo delegado é, portanto, (3,0) e depois (4,1).

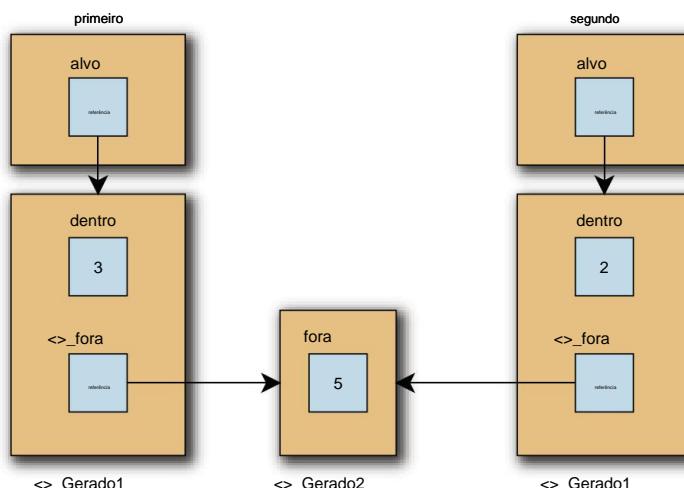
Apenas por uma questão de interesse, vamos pensar em como isso é implementado — pelo menos com o compilador C# 2 da Microsoft. O que acontece é que uma classe extra é gerada para conter a variável externa, e outra é gerada para armazenar uma variável interna *e uma referência à primeira classe extra*. Essencialmente, cada escopo que contém uma variável capturada obtém seu próprio tipo, com uma referência ao próximo escopo que contém uma variável capturada. Neste exemplo, havia duas instâncias do tipo que continham dentro e ambas se referiam à mesma instância do tipo que continha fora. Outras implementações podem variar, mas esta é a maneira mais óbvia de fazer as coisas. A Figura 5.1 mostra os valores após a execução da listagem 5.14. (Os nomes na figura não são aqueles que o compilador geraria, mas são próximos o suficiente. Observe que as instâncias delegadas também teriam outros membros na realidade - no entanto, apenas o destino é interessante aqui.)

Mesmo depois de entender esse código completamente, ele ainda é um bom modelo para experimentar outros elementos de variáveis capturadas. Como observei anteriormente, certos elementos da captura de variáveis são específicos da implementação e muitas vezes é útil consultar a especificação para ver o que é garantido. Mas também é importante *brincar* com o código para ver o que acontece.

É possível que existam situações em que um código como o da listagem 5.14 seja a maneira mais simples e clara de expressar o comportamento desejado, mas eu teria que ver para acreditar e certamente gostaria de comentários no código para explicar o que aconteceria. Então, quando é apropriado usar variáveis capturadas e o que você precisa observar?

### 5.5.7 Diretrizes e resumo das variáveis capturadas

Esperamos que esta seção tenha convencido você a ter *muito* cuidado com as variáveis capturadas. Eles fazem sentido lógico (e quase qualquer mudança para torná-los mais simples seria



**Figura 5.1** Instantâneo de vários escopos de variáveis capturados na memória

provavelmente os tornam menos úteis ou menos lógicos), mas também facilitam a produção de códigos terrivelmente complicados.

Porém, não deixe que isso o desencoraje de usá-los de maneira sensata - eles podem economizar muito código tedioso e, quando usados de maneira adequada, podem ser a maneira mais legível de realizar o trabalho. Mas o que é considerado *sensato*?

Aqui estão algumas sugestões para usar variáveis capturadas:

ÿ Se o código que não usa variáveis capturadas for tão simples quanto o código que o faz, não os use.

ÿ Antes de capturar uma variável declarada por uma instrução `for` ou `foreach`, considere se seu delegado sobreviverá além da iteração do loop e se você deseja que ele veja os valores subsequentes daquela variável. Caso contrário, crie outra variável dentro do loop que apenas copie o valor *desejado*. (Em C# 5 você não precisa se preocupar com instruções `foreach`, mas ainda precisa tomar cuidado com instruções `for`.)

ÿ Se você criar múltiplas instâncias delegadas (seja em um loop ou explicitamente) que capturam variáveis, pense se deseja que elas capturem a mesma variável.

ÿ Se você capturar uma variável que não muda de fato (seja no método anônimo ou no corpo do método), você não precisa se preocupar tanto. ÿ Se as instâncias delegadas que você cria nunca escapam do método – em outras palavras, elas nunca são armazenadas em nenhum outro lugar, ou retornadas, ou usadas para iniciar threads – a vida é muito mais simples.

ÿ Considere o tempo de vida estendido de qualquer variável capturada em termos de coleta de lixo. Normalmente isso não é um problema, mas se você capturar um objeto que seja caro em termos de memória, poderá ser significativo.

O primeiro ponto é a regra de ouro. Simplicidade é uma coisa boa, portanto, sempre que o uso de uma variável capturada tornar seu código mais simples depois de você considerar a complexidade inerente adicional de forçar os mantenedores do seu código a entender o que a variável capturada faz, use-a. Você precisa incluir essa complexidade extra em suas considerações, só isso – não opte apenas pela contagem mínima de linhas.

Abordamos muito nesta seção e estou ciente de que pode ser difícil de entender. Listei as coisas mais importantes a serem lembradas a seguir, para que, se você precisar voltar a esta seção mais tarde, você pode refrescar sua memória sem ter que ler tudo novamente:

ÿ A variável é capturada - não seu valor no ponto da instância delegada criação.

ÿ Variáveis capturadas têm vida útil estendida pelo menos até aquela da captura delegar.

ÿ Vários delegados podem capturar a mesma variável... ÿ ... mas dentro de loops, a mesma declaração de variável pode efetivamente se referir a diferentes “instâncias” de variáveis.

ÿ Declarações de loop for criam variáveis que permanecem durante a duração do loop—eles não são instanciados em cada iteração. O mesmo vale para instruções foreach anteriores ao C# 5.

ÿ Tipos extras são criados, quando necessário, para armazenar variáveis capturadas.

ÿ Tenha cuidado! Simples é quase sempre melhor que inteligente.

Você verá mais variáveis sendo capturadas quando olharmos para o C# 3 e suas expressões lambda, mas por enquanto você pode ficar aliviado ao saber que terminamos nosso resumo do novos recursos de delegação do C# 2.

## 5.6 Resumo

O C# 2 mudou radicalmente as maneiras pelas quais os delegados podem ser criados e, ao fazer isso, então abriu a estrutura para um estilo de programação mais funcional. Lá existem mais métodos no .NET 2.0 que usam delegados como parâmetros do que havia em .NET 1.0/1.1, e essa tendência continua no .NET 3.5. O tipo `List<T>` é o melhor exemplo disso e é um bom teste para verificar suas habilidades no uso de anônimos.

métodos e variáveis capturadas. Programar desta forma requer uma abordagem ligeiramente diferente mentalidade - você deve ser capaz de dar um passo atrás e considerar qual é o objetivo final, e se é melhor expresso na maneira tradicional do C# ou se é um funcional abordagem torna as coisas mais claras.

Todas as alterações no tratamento de delegados são úteis, mas acrescentam complexidade ao linguagem, especialmente quando se trata de variáveis capturadas. Fechamentos são sempre complicados em termos de determinar exatamente como o ambiente disponível é compartilhado, e C# não é diferente neste aspecto. A razão pela qual o conceito durou tanto tempo, porém, é que pode tornar o código mais simples de entender e mais imediato. O ato de equilíbrio entre complexidade e simplicidade é sempre difícil e vale a pena ser cauteloso para começar. Mas com o tempo você deverá melhorar no trabalho com variáveis capturadas e na compreensão de como elas se comportam. LINQ incentiva seu uso mesmo além disso, e uma grande parte do código C# moderno e idiomático usa fechamentos com frequência.

Os métodos anônimos não são a única mudança no C# 2 que envolve o compilador criando tipos extras nos bastidores e fazendo coisas tortuosas com variáveis que parecem ser locais. Você verá muito mais sobre isso no próximo capítulo, onde o compilador efetivamente constrói uma máquina de estados completa para você, a fim de tornar tudo mais fácil para você. para implementar iteradores.

# Implementando iteradores da maneira mais fácil

---

## Este capítulo cobre

- ÿ Implementando iteradores em C# 1
- ÿ Blocos iteradores em C# 2
- ÿ Exemplo de uso do iterador
- ÿ Iteradores como corrotinas

O padrão iterador é um exemplo de *padrão comportamental* – um padrão de design que simplifica a comunicação entre objetos. É um dos padrões mais simples de entender e incrivelmente fácil de usar. Em essência, ele permite acessar todos os elementos em uma sequência de itens sem se preocupar com o tipo de sequência — um array, uma lista, uma lista vinculada ou nenhuma das opções acima. Isso pode ser eficaz para construir um *pipeline de dados*, onde um item de dados entra no pipeline e passa por diversas transformações ou filtros diferentes antes de sair pelo outro lado. Na verdade, este é um dos padrões centrais do LINQ, como você verá na parte 3 do livro.

No .NET, o padrão do iterador é encapsulado pelas interfaces `IEnumerator` e `IEnumerable` e seus equivalentes genéricos. (A nomeação é lamentável – o padrão normalmente é chamado de *iteração* para evitar confundi-lo com outros significados da palavra *enumeração*. Usei *iterador* e *iterável* ao longo deste capítulo.) Se um tipo implementa

IEnumerable, isso significa que pode ser iterado; chamar o método GetEnumerator retornará a implementação do IEnumerator , que é o próprio iterador. Você pode pensar no iterador como um cursor de banco de dados: uma posição dentro da sequência. O iterador só pode avançar dentro da sequência e pode haver muitos iteradores operando na mesma sequência ao mesmo tempo.

Como linguagem, o C# 1 possui suporte integrado para consumir iteradores usando a instrução foreach . Isso facilita a iteração sobre coleções – mais fácil do que usar um loop for direto – e é bastante expressivo. A instrução foreach é compilada para chamadas aos métodos GetEnumerator e MoveNext e à propriedade Current , com suporte para descartar o iterador posteriormente se IDisposable tiver sido implementado. É um pedaço pequeno, mas útil, de açúcar sintático.

No entanto, em C# 1, *implementar* um iterador é uma tarefa relativamente difícil. O C# 2 torna isso muito mais simples, o que às vezes pode fazer com que valha a pena implementar o padrão do iterador em casos em que anteriormente ele teria causado mais trabalho do que economizado.

Neste capítulo, veremos o que é necessário para implementar um iterador e o suporte fornecido pelo C# 2. Depois de examinarmos a sintaxe em detalhes, examinaremos alguns exemplos do mundo real, incluindo uma interessante (se um pouco estranho) uso da sintaxe de iteração em uma biblioteca de simultaneidade da Microsoft. Deixe de fornecer os exemplos até o final da descrição, porque não há *muito* o que aprender, e os exemplos ficarão muito mais claros quando você entender o que o código está fazendo. Se você realmente quiser ler os exemplos primeiro, eles estão nas seções 6.3 e 6.4.

Como em outros capítulos, vamos começar analisando como era a vida antes do C# 2. Veremos implementar um iterador da maneira mais difícil.

### 6.1 C# 1: O problema dos iteradores manuscritos

Você já viu um exemplo

de implementação de iterador na seção 3.4.3, quando vimos o que acontece quando você itera em uma coleção genérica. De certa forma, isso foi mais difícil do que uma implementação real de iterador C# 1 teria sido, porque você implementou as interfaces genéricas também - mas também foi mais fácil de outras maneiras, porque na verdade não estava iterando nada útil.

Para contextualizar os recursos do C# 2, primeiro implementaremos um iterador que seja o mais simples possível e, ao mesmo tempo, forneça funcionalidade real. Suponha que você queira um novo tipo de coleção baseado em um buffer circular. Neste exemplo, você implementará IEnumerable para que os usuários da sua nova classe possam iterar facilmente todos os valores da coleção. Iremos ignorar a essência da coleção aqui e nos concentrar apenas no lado da iteração. Sua coleção armazenará seus valores em um array (objeto[] — sem genéricos aqui), e a coleção terá o recurso interessante de que você pode definir seu ponto de partida lógico — portanto, se o array tivesse cinco elementos, você poderia definir o ponto de início para 2 e espere que os elementos 2, 3, 4, 0 e 1 sejam retornados. Não mostrarei o código completo do buffer circular aqui, mas ele está no código para download.

Para facilitar a demonstração da classe, você fornecerá os valores e o ponto inicial no construtor, portanto, deverá ser capaz de escrever código como o seguinte para iterar na coleção.

**Listagem 6.1 Código usando o novo tipo de coleção (ainda não implementado)**

```
objeto[] valores = {"a", "b", "c", "d", "e"};
Coleção IterationSample = new IterationSample(valores, 3); foreach (objeto x na coleção) {

    Console.WriteLine(x);
}
```

A execução da listagem 6.1 deve (eventualmente) produzir saída de d, e , a, be finalmente c porque você especificou um ponto inicial de 3. Agora que você sabe o que precisa alcançar, vamos dar uma olhada no esqueleto da classe, como mostrado na listagem a seguir.

**Listagem 6.2 Esqueleto do novo tipo de coleção, sem implementação de iterador**

```
usando o sistema;
usando System.Collections;

classe pública IterationSample: IEnumerable {

    valores de objeto[]; int
    pontoinicial;

    public IterationSample(objeto[] valores, int ponto inicial) {

        this.values = valores; this.pontoinicial
        = pontoinicial;
    }

    public IEnumerator GetEnumerator() {

        lançar new NotImplementedException();
    }
}
```

Você ainda não implementou GetEnumerator , mas o restante do código está pronto para uso. E como você escreve o código GetEnumerator ? A primeira coisa a entender é que você precisa armazenar algum *estado* em algum lugar. Um aspecto importante do padrão do iterador é que você não retorna todos os dados de uma só vez – o cliente solicita um elemento de cada vez. Isso significa que você precisa acompanhar o quanto já percorreu seu array. A natureza com estado dos iteradores será importante quando observarmos o que o compilador C# 2 faz, portanto, fique atento ao estado exigido neste exemplo.

Onde esse estado deveria morar? Suponha que você tentou colocá-lo na classe IterationSample , fazendo com que implementasse IEnumerator e também IEnumerable. À primeira vista, parece um bom plano – afinal, os *dados* estão no lugar certo, inclusive no ponto de partida. Seu método GetEnumerator poderia simplesmente retornar isso. Mas há um grande problema com essa abordagem: se GetEnumerator for chamado diversas vezes, vários iteradores independentes deverão ser retornados. Por exemplo, você deve ser capaz de usar duas instruções foreach , uma dentro da outra, para obter todos os pares de valores possíveis. Os dois iteradores precisam ser independentes, o que sugere que você precisa criar um novo objeto a cada vez

GetEnumerator é chamado. Você ainda poderia implementar a funcionalidade diretamente dentro de IterationSample, mas então você teria uma classe que não teria uma única responsabilidade clara – seria bastante confuso.

Em vez disso, vamos criar outra classe para implementar o próprio iterador. Você pode usar o fato de que em C# um tipo aninhado tem acesso aos membros privados de seu tipo envolvente, o que significa que você pode armazenar uma referência ao pai IterationSample, junto com o estado de quantas iterações você executou até agora. Isso é mostrado na listagem a seguir.

#### Listagem 6.3 Classe aninhada implementando o iterador da coleção

```
classe IterationSampleIterator: IEnumerator {
    IterationSample pai; posição interna;

    interno IterationSampleIterator(IterationSample pai) {
        this.parent = pai; posição = -1;
    }

    public bool MoveNext() {
        if (posição! = pai.valores.Comprimento) {
            posição++;
        }
        return posição < parent.values.Length;
    }

    objeto público Atual {
        pegar
        {
            if (posição == -1 || posição == pai.valores.Comprimento)
            {
                lançar new InvalidOperationException();
            }
            int índice = posição + parent.startingPoint; índice = índice%
                pai.valores.Comprimento; retornar pai.valores[índice];
        }
    }

    public void Redefinir() {
        posição = -1;
    }
}
```

Quanto código para realizar uma tarefa tão simples! Você se lembra da coleção original de valores que está iterando sobre **B** e acompanha onde estaria em um array simples baseado em zero **C**. Para retornar um elemento, você compensa esse índice pelo valor inicial.

ponto G. De acordo com a interface, você considera que seu iterador inicia logicamente antes do primeiro elemento D, portanto o cliente terá que chamar MoveNext antes de usar a propriedade Current pela primeira vez. O incremento condicional em E torna o teste em F simples e correto, mesmo que MoveNext seja chamado novamente após ser relatado pela primeira vez que não há mais dados disponíveis. Para redefinir o iterador, você define a posição lógica antes do primeiro elemento H.

A maior parte da lógica envolvida é bastante direta, embora haja bastante espaço para erros ocasionais; minha primeira implementação falhou em seus testes unitários exatamente por esse motivo. A boa notícia é que funciona e você só precisa implementar IEnumerable em IterationSample para completar o exemplo:

```
public IEnumerator GetEnumerator() {  
    retornar novo IterationSampleIterator(este);  
}
```

Não reproduzirei o código combinado aqui, mas ele está disponível no site do livro, incluindo a listagem 6.1, que agora produz o resultado esperado.

Vale a pena ter em mente que este é um exemplo relativamente simples — não há muitos estados para acompanhar e não há nenhuma tentativa de verificar se a coleção mudou entre as iterações. Com uma carga tão grande envolvida para implementar um iterador simples, você não deveria se surpreender com o quão raramente esse padrão foi implementado em C# 1. Os desenvolvedores geralmente ficavam felizes em usar foreach nas coleções fornecidas pela estrutura, mas de forma mais direta (e coleção -específico) quando se trata de suas próprias coleções.

Foram necessárias cerca de 40 linhas de código para implementar o iterador em C# 1. Vamos ver se o C# 2 consegue fazer melhor.

## 6.2 C# 2: Iteradores simples com instruções de rendimento

Sempre fui o tipo de pessoa que gosta de ficar acordada até meia-noite na véspera de Natal para poder abrir um presente assim que chegar o dia de Natal. Da mesma forma, acho quase impossível esperar um período significativo de tempo antes de mostrar como a solução é organizada em C# 2.

### 6.2.1 Apresentando blocos iteradores e retorno de rendimento

Este capítulo não existiria se o C# 2 não tivesse um recurso poderoso que reduza a quantidade de código que você precisa escrever para implementar iteradores. Em alguns outros tópicos, a quantidade de código foi apenas ligeiramente reduzida ou as alterações apenas tornaram algo mais elegante. Neste caso, porém, a quantidade de código necessária é reduzida *enormemente*. A listagem a seguir mostra a implementação completa do método GetEnumerator em C# 2.

**Listagem 6.4 Iterando através da coleção de amostras com C# 2 e retorno de rendimento**

```

público IEnumerator GetEnumerator()
{
    for (int índice = 0; índice <valores.Comprimento; índice++) {

        rendimento valores de retorno [(índice + ponto inicial)% valores.Comprimento];
    }
}

```

Quatro linhas de implementação, duas das quais são apenas colchetes. Para ser claro, isso substitui toda a classe `IterationSampleIterator`. Completamente. Pelo menos no código-fonte... Mais tarde você verá o que o compilador fez nos bastidores e algumas das peculiaridades da implementação que ele forneceu, mas por enquanto vamos dar uma olhada no código-fonte.

O método parece perfeitamente normal até você ver o uso do retorno de rendimento. Isso é o que diz ao compilador C# que este não é um método normal, mas implementado com um *bloco iterador*. O método é declarado para retornar um `IEnumerator` e você só pode usar blocos iteradores para implementar métodos<sup>1</sup> que tenham um tipo de retorno `IEnumerable`, `IEnumerator` ou um dos equivalentes genéricos. O *tipo de rendimento* do bloco iterador é object se o tipo de retorno declarado do método for uma interface não genérica ou, caso contrário, é o argumento de tipo da interface genérica. Por exemplo, um método declarado para retornar `IEnumerable<string>` teria um tipo de rendimento de `string`.

Nenhuma instrução de retorno normal é permitida em blocos iteradores – apenas retorno de rendimento. Todas as instruções de retorno de rendimento no bloco devem tentar retornar um valor compatível com o tipo de rendimento do bloco. No exemplo anterior, você não poderia escrever `yield return 1`; em um método declarado para retornar `IEnumerable<string>`.

**RESTRICOES AO RETORNO DE RENDIMENTO** Existem algumas restrições adicionais nas declarações de rendimento. Você não pode usar `yield return` dentro de um bloco `try` se ele tiver algum bloco `catch`, e você não pode usar `yield return` ou `yield break` (que abordaremos em breve) em um bloco finalmente. Isso não significa que você não pode usar blocos `try/ catch` ou `try/finally` dentro de iteradores — isso apenas restringe o que você pode fazer neles. Se você quiser saber mais sobre por que essas restrições existem, Eric Lippert tem uma série de postagens no blog sobre essas e outras decisões de design envolvendo iteradores: consulte <http://mng.bz/EJ97>.

A grande ideia que você precisa entender quando se trata de blocos iteradores é que, embora você tenha escrito um método que parece ser executado sequencialmente, o que você realmente pediu ao compilador para fazer é criar uma máquina de estado *para* você. Isso é necessário exatamente pela mesma razão pela qual você teve que se esforçar tanto para implementar o iterador em C# 1 – o chamador só quer ver um elemento por vez, então você precisa acompanhar o que estava fazendo quando último retornou um valor.

---

<sup>1</sup> Ou propriedades, como você verá mais adiante. Porém, você não pode usar um bloco iterador em um método anônimo.

Quando o compilador encontra um bloco iterador, ele cria um tipo aninhado para a máquina de estados. Este tipo lembra exatamente onde você está dentro do bloco e os valores das variáveis locais (incluindo parâmetros). A classe gerada é *um pouco* semelhante à implementação escrita que você escreveu anteriormente, no sentido de que ela mantém todo o estado necessário como variáveis de instância. Vamos pensar no que esta máquina de estado tem que fazer para para implementar o iterador:

- ÿ Tem que ter algum estado inicial.
- ÿ Sempre que MoveNext é chamado, ele precisa executar o código do método GetEnumerator até que você esteja pronto para fornecer o próximo valor (em outras palavras, até que você atinja uma instrução de retorno de rendimento).
- ÿ Quando a propriedade Current é usada, ela deve retornar o último valor obtido. ÿ Ele precisa saber quando você terminou de gerar valores para que MoveNext possa retornar falso.

O segundo ponto desta lista é complicado, porque a máquina de estado sempre precisa reiniciar o código a partir do ponto em que foi alcançado anteriormente. Acompanhar as variáveis locais (conforme aparecem no método) não é muito difícil — elas são representadas por variáveis de instância na máquina de estado. O aspecto de reiniciar é mais complicado, mas a boa notícia é que, a menos que você mesmo esteja escrevendo um compilador C#, não precisa se preocupar com como isso é feito: o resultado do ponto de vista da caixa preta é que ele simplesmente funciona. Você pode escrever código perfeitamente normal dentro do bloco iterador, e o compilador é responsável por garantir que o fluxo de execução seja exatamente como seria em qualquer outro método. A diferença é que uma instrução de retorno de rendimento parece sair apenas temporariamente do método — você poderia pensar nela como sendo pausada, efetivamente.

A seguir examinaremos o fluxo de execução com mais detalhes e de uma forma mais visual.

### 6.2.2 Visualizando o fluxo de trabalho de um iterador

Pode ser útil pensar em como os iteradores são executados em termos de um diagrama de sequência. Em vez de desenhar o diagrama à mão, a listagem a seguir o imprime. O iterador fornece uma sequência de números (0, 1, 2, -1) e então termina. A parte interessante não são os números fornecidos, mas sim o *fluxo* do código.

#### Listagem 6.5 Mostrando a sequência de chamadas entre um iterador e seu chamador

```
string somente_leitura_estática Padding = new string(' ', 30);

static IEnumerable<int> CreateEnumerable() {

    Console.WriteLine("{0}Início de CreateEnumerable()", Padding);
    for (int i=0; i < 3; i++) {

        Console.WriteLine("{0}Prestes a produzir {1}", Padding, i); rendimento retorno i;
        Console.WriteLine("{0}
        Após o rendimento", Padding);
    }
    Console.WriteLine("{0}Produzindo valor final", Padding);
```

```

rendimento retorno -1;

Console.WriteLine("{0}Fim de CreateEnumerable()", Padding);
}

...
IEnumarable<int> iterável = CreateEnumerable(); IEnumarator<int> iterador =
iterável.GetEnumerator(); Console.WriteLine("Iniciando a iteração"); enquanto (verdadeiro)
{

Console.WriteLine("Chamando MoveNext()..."); resultado bool =
iterator.MoveNext(); Console.WriteLine("... MoveNext
resultado={0}", resultado); if (!resultado){

quebrar;
}
Console.WriteLine("Buscando Atual..."); Console.WriteLine("...
Resultado atual={0}", iterator.Current);
}

```

A Listagem 6.5 não é bonita, principalmente no que diz respeito à iteração. No curso normal dos eventos, você poderia usar apenas um loop foreach , mas para mostrar exatamente o que está acontecendo e quando, tive que dividir o uso do iterador em pedaços. Esse código faz basicamente o que foreach faz, embora foreach também chame Dispose no final. Isso é importante para blocos iteradores, como discutiremos em breve. Como você pode ver, não há diferença na sintaxe dentro do método iterador, embora desta vez você esteja retornando IEnumarable<int> em vez de IEnumarator<int>. Normalmente você só deseja retornar IEnumarator<T> para implementar IEnumarable<T>; se você quiser apenas gerar uma sequência de um método, retorne IEnumarable<T> .

Aqui está o resultado da listagem 6.5:

```

Começando a iterar Chamando
MoveNext()...
Início de CreateEnumerable()
Prestes a render 0
... MoveNext resultado = Verdadeiro
Buscando Atual...
... Resultado atual=0
Chamando MoveNext()...
Depois do rendimento
Prestes a render 1
... MoveNext resultado = Verdadeiro
Buscando Atual...
... Resultado atual=1
Chamando MoveNext()...
Depois do rendimento
Prestes a render 2
... MoveNext resultado = Verdadeiro
Buscando Atual...
... Resultado atual=2
Chamando MoveNext()...
Depois do rendimento

```

```

    ... MoveNext resultado = Verdadeiro
    Rendendo valor final
    Buscando Atual...
    ... Resultado atual=-1
    Chamando MoveNext()...
    Fim de CreateEnumerable()
    ... MoveNext resultado=Falso

```

Há várias coisas importantes a serem observadas nesta saída:

- ÿ Nenhum código em CreateEnumerable é chamado até a primeira chamada para MoveNext.
- ÿ Somente quando você chama MoveNext é que qualquer trabalho real é realizado. Buscando Atual não executa nenhum do seu código.
- ÿ O código para de ser executado no retorno de rendimento e retoma logo depois em a próxima chamada para MoveNext.
- ÿ Você pode ter múltiplas declarações de rendimento e retorno em diferentes locais do método.
- ÿ O código não termina no último retorno de rendimento. Em vez disso, a chamada para MoveNext que faz com que você chegue ao final do método é aquela que retorna falso.

O primeiro ponto é particularmente importante, porque significa que você não pode usar um bloco iterador para qualquer código que precise ser executado imediatamente quando o método for chamado, como código para validação de argumento. Se você colocar a verificação normal em um método implementado com um bloco iterador, ele não se comportará bem. É quase certo que você cairá nisso em algum momento – é um erro extremamente comum e difícil de entender até que você pense no que o bloco iterador está fazendo. Você verá a solução para o problema na seção 6.3.3.

Há duas coisas que você ainda não viu: uma maneira alternativa de interromper a iteração e como os blocos finalmente funcionam nessa forma um tanto estranha de execução. Vamos dar uma olhada neles agora.

### 6.2.3 Fluxo de execução do iterador avançado

Nos métodos normais, a instrução return tem dois efeitos. Primeiro, ele fornece o valor que o chamador vê como valor de retorno. Segundo, ele encerra a execução do método, executando quaisquer blocos finalmente apropriados na saída. Você viu que a instrução yield return sai temporariamente do método, mas apenas até que MoveNext seja chamado novamente, e não examinamos o comportamento dos blocos finalmente . Como você pode *realmente* interromper o método e o que acontece com todos esses bloqueios finais ? Começaremos com uma construção bastante simples: a instrução yield break .

#### FINALIZANDO UM ITERATOR COM YIELD BREAK

Você sempre pode encontrar uma maneira de dar a um método um único ponto de saída, e muitas pessoas trabalham duro para conseguir isso.<sup>2</sup> As mesmas técnicas podem ser aplicadas em blocos iteradores. Mas

---

<sup>2</sup> Acho que os obstáculos que você precisa percorrer para conseguir isso geralmente tornam o código muito mais difícil de ler do que ter vários pontos de retorno, especialmente porque try/finally está disponível para limpeza e você precisa levar em conta a possibilidade de ocorrência de exceções de qualquer maneira. A questão é que isso pode ser feito.

se você quiser sair mais cedo, a declaração de rendimento é sua amiga. Isso efetivamente encerra o iterador, fazendo com que a chamada atual para MoveNext retorne falso.

A listagem a seguir demonstra isso contando até 100, mas parando mais cedo se o tempo acabar. Este código também demonstra o uso de um parâmetro de método em um bloco iterador e prova que o nome do método é irrelevante.<sup>3</sup>

#### Listagem 6.6 Demonstração de quebra de rendimento

```
estático IEnumerable<int> CountWithTimeLimit(limite de data e hora)
{
    for (int i = 1; i <= 100; i++) {

        if (DateTime.Now >= limite)
        {
            quebra de rendimento;  Pára se o tempo acabar
        } rendimento retorno i;
    }
    ...
}

Parada DateTime = DateTime.Now.AddSeconds(2); foreach (int i em
CountWithTimeLimit(stop)) {

    Console.WriteLine("Recebido {0}", i); Thread.Sleep(300);

}
```

Normalmente, ao executar a listagem 6.6, você verá cerca de sete linhas de saída. O loop foreach termina perfeitamente normalmente – no que diz respeito, o iterador acabou de ficar sem elementos para iterar. A instrução yield break se comporta de maneira semelhante a uma instrução return em um método normal.

Até agora, tão simples. Há um último aspecto do fluxo de execução a ser explorado: como e quando finalmente os blocos são executados.

#### EXECUÇÃO DE BLOCOS FINAIS

Você está acostumado a finalmente bloquear a execução sempre que sai do escopo relevante. No entanto, os blocos iteradores não se comportam como métodos normais. Como você viu, uma instrução yield return efetivamente pausa o método em vez de sair dele. Segundo essa lógica, você não esperaria que nenhum bloco final fosse executado nesse ponto, e eles não são. Mas os blocos finalmente apropriados são executados quando uma instrução yield break é atingida, exatamente como você esperaria que ocorressem ao retornar de um método normal.<sup>4</sup>

O uso mais comum de finalmente em um bloco iterador é descartar recursos, normalmente com uma instrução using conveniente. Você verá um exemplo real disso na seção 6.3.2, mas por enquanto estamos apenas tentando ver como e quando os blocos finalmente são

<sup>3</sup> Observe que os métodos que utilizam parâmetros ref ou out não podem ser implementados com blocos

<sup>4</sup> finalmente, os blocos também funcionam conforme o esperado quando a execução sai do escopo relevante sem atingir um retorno de rendimento ou uma instrução de quebra de rendimento. Estou focando apenas no comportamento das duas declarações de rendimento aqui porque é aí que o fluxo de execução é novo e diferente.

executado. A listagem a seguir mostra isso em ação – é o mesmo código da listagem 6.6, mas com um bloco finalmente . As alterações são mostradas em negrito.

### Listagem 6.7 Demonstração de yield break trabalhando com try/finally

```
estático IEnumerable<int> CountWithTimeLimit(limite de data e hora)
{
    tenta
    {
        for (int i = 1; i <= 100; i++) {
            if (DateTime.Now >= limite)
            {
                quebra de rendimento;
            }
            rendimento retorno i;
        }
    }
    finalmente
    {
        Console.WriteLine("Parando!");
    }
}
...
Parada DateTime = DateTime.Now.AddSeconds(2); foreach (int i em
CountWithTimeLimit(parar))
{
    Console.WriteLine("Recebido {0}", i); Thread.Sleep(300);
}
```

← Executa no entanto o loop termina

O bloco final na listagem 6.7 é executado independentemente de o bloco iterador terminar contando até 100 ou devido ao limite de tempo ter sido atingido. (Ele também seria executado se o código lançasse uma exceção.) Mas há outras maneiras de tentar evitar que o bloco finalmente seja chamado... Vamos tentar ser sorrteiros.

Você viu que o código no bloco iterador só é executado quando MoveNext é chamado. Então, o que acontece se você nunca ligar para MoveNext? Ou se você ligar algumas vezes e depois parar? Vamos considerar mudar a parte de chamada da listagem 6.7 para isto:

```
Parada DateTime = DateTime.Now.AddSeconds(2); foreach (int i em
CountWithTimeLimit(stop)) {
    Console.WriteLine ("Recebido {0}", i); se (eu > 3)

    {
        Console.WriteLine("Retornando"); retornar;

    }
    Thread.Sleep(300);
}
```

Aqui você não para no início do código do iterador — você para no início do código *usando o iterador*. O resultado talvez seja surpreendente:

```
Recebido 1
Recebido 2
Recebido 3
Recebido 4
Retornando
Parando!
```

Você pode ver que o código está sendo executado após a instrução `return` no loop `foreach`. Isso normalmente não acontece a menos que um bloco final esteja envolvido – e neste caso há dois! Você já conhece o bloco finalmente no método iterador, mas a questão é o que está causando sua execução.

Dei uma dica sobre isso anteriormente: `foreach` chama `Dispose` no iterador retornado por `GetEnumerator` em seu próprio bloco final (assim como a instrução `using`). Quando você chama `Dispose` em um iterador criado com um bloco iterador antes de terminar a iteração, a máquina de estado executa quaisquer blocos finalmente que estejam no escopo onde o código está atualmente “pausado”. Essa é uma explicação complicada e um tanto detalhada, mas o resultado é mais simples de expressar: contanto que os chamadores usem um loop `foreach`, finalmente funciona dentro dos blocos iteradores da maneira que você deseja.

Você pode facilmente provar que é a chamada para `Dispose` que aciona isso usando o iterador manualmente:

```
Parada DateTime = DateTime.Now.AddSeconds(2); IEnumerable<int>
iterável = CountWithTimeLimit(stop); IEnumerator<int> iterador = iterável.GetEnumerator();

iterador.MoveNext();
Console.WriteLine("Recebido {0}", iterator.Current);

iterador.MoveNext();
Console.WriteLine("Recebido {0}", iterator.Current);
```

Desta vez, a linha de parada nunca é impressa. Se você adicionar explicitamente uma chamada para `Dispose`, verá a linha extra na saída novamente. É relativamente raro você querer encerrar um iterador antes de ele terminar, e é relativamente raro você iterar manualmente em vez de usar `foreach`, mas se *fizer isso*, lembre-se de envolver o iterador em uma instrução `using`.

Já cobrimos a maior parte do comportamento dos blocos iteradores, mas antes de encerrarmos esta seção, vale a pena considerar algumas curiosidades relacionadas à implementação atual da Microsoft.

#### 6.2.4 Peculiaridades na implementação

Se você compilar blocos iteradores com o compilador Microsoft C# 2 e observar o IL resultante no ildasm ou no Reflector, verá o tipo aninhado que o compilador gerou para você nos bastidores. No meu caso, ao compilar a listagem 6.4, ela foi chamada `IterationSample.<GetEnumerator>d__0` (onde os colchetes angulares tornam-no um nome indescritível - nada a ver com genéricos). Não vou detalhar exatamente o que é gerado aqui, mas vale a pena dar uma olhada no Reflector para ter uma ideia do que está acontecendo, de preferência com a especificação da linguagem ao seu lado, aberta na seção 10.14

(“Iteradores”); a especificação define diferentes estados em que o tipo pode estar, e essa descrição torna o código gerado mais fácil de seguir. A maior parte do trabalho é realizada no MoveNext, que geralmente é uma grande instrução de mudança .

Felizmente, como desenvolvedores, não precisamos nos preocupar muito com os obstáculos que o compilador tem que superar. Mas existem algumas peculiaridades sobre a implementação que vale a pena conhecer:

- ÿ Antes de MoveNext ser chamado pela primeira vez, a propriedade Current sempre retornará o valor padrão para o tipo de rendimento do iterador.
- ÿ Após MoveNext retornar falso, a propriedade Current sempre retornará o valor final obtido. ÿ Redefinir sempre lança uma exceção em vez de redefinir como fez sua implementação manual. Este é um comportamento necessário, estabelecido na especificação. ÿ A classe aninhada sempre implementa as formas genérica e não genérica de IEnumarator (e a forma genérica e não genérica de IEnumarable, quando apropriado).

Deixar de implementar Reset é razoável – o compilador não consegue descobrir o que você precisaria fazer para redefinir o iterador, ou mesmo se isso é viável. Indiscutivelmente , Reset não deveria estar na interface do IEnumarator para começar, e não me lembro da última vez que o chamei. Muitas coleções não oferecem suporte a isso, portanto, os chamadores geralmente não podem confiar nele.

A implementação de interfaces extras também não causa danos. É interessante que se o seu método retornar IEnumarable, você terá uma classe implementando cinco interfaces (incluindo IDisposable). A especificação da linguagem explica isso em detalhes, mas o resultado é que, como desenvolvedor, você não precisa se preocupar. O fato de ele implementar IEnumarable e IEnumarator é um pouco incomum - o compilador se esforça para garantir que o comportamento esteja correto, independentemente do que você fizer com ele, mas também consegue criar uma única instância do tipo aninhado no caso comum onde você apenas percorre a coleção no mesmo thread que a criou.

O comportamento do Current é estranho – em particular, manter o último item depois de supostamente ter sido removido pode impedir que ele seja coletado como lixo. É possível que isso seja corrigido em uma versão posterior do compilador C#, mas é improvável, pois pode quebrar o código existente (os compiladores Microsoft C# fornecidos com .NET 3.5, 4 e 4.5 se comportam da mesma maneira) . Estritamente falando, está correto do ponto de vista da especificação da linguagem C# 2 — o comportamento da propriedade Current é indefinido nesta situação. Seria melhor, porém, se implementasse a propriedade da maneira sugerida pela documentação do framework, lançando exceções nos momentos apropriados.

Essas são algumas pequenas desvantagens de usar o código gerado automaticamente, mas os chamadores sensatos não terão nenhum problema – e vamos encarar os fatos, você economizou muito código para chegar à implementação. Isso significa que faz sentido usar iteradores de forma mais ampla do que você faria no C# 1. A próxima seção fornece alguns exemplos de código para que você possa verificar sua compreensão dos blocos de iteradores e ver como eles são úteis na vida real, e não na teoria. cenários.

## 6.3 Exemplos de iteradores da vida real

Você já escreveu algum código que é realmente simples por si só, mas que torna seu projeto *muito* mais organizado? Isso acontece comigo de vez em quando e geralmente me deixa mais feliz do que deveria – o suficiente para receber olhares estranhos dos colegas, de qualquer maneira. Esse tipo de prazer infantil é particularmente forte quando se trata de usar um novo recurso de linguagem de uma maneira que é *claramente* mais agradável e você não está fazendo isso apenas para brincar com novos brinquedos.

Mesmo agora, depois de usar iteradores por alguns anos, ainda me deparo com situações em que uma solução usando blocos iteradores se apresenta e o código resultante é breve, limpo e fácil de entender. Nesta seção, compartilharei três desses cenários com você.

### 6.3.1 Iterando as datas em um horário

Enquanto trabalhava em um projeto envolvendo cronogramas, me deparei com alguns loops, todos começando assim:

```
for (DateTime dia = horário.StartDate; dia <= horário.EndDate; dia
    = dia.AddDays(1))
```

Eu estava trabalhando muito nessa área de código e sempre odiei esse loop, mas foi só quando estava lendo o código em voz alta para outro desenvolvedor como pseudocódigo que percebi que estava faltando um truque. Eu disse algo como: “Para cada dia dentro do horário...”

Olhando retrospectivamente, é óbvio que o que eu realmente queria era um loop `foreach`. (Isso pode ter sido óbvio para você desde o início - desculpas se for o caso. Felizmente, não consigo ver você parecendo presunçoso.) O loop é muito melhor quando reescrito da seguinte maneira:

```
foreach (DateTime dia in timetable.DateRange)
```

Em C# 1, eu poderia ter encarado isso como um grande sonho, mas não me preocupei em implementá-lo; você viu como é complicado implementar um iterador manualmente, e o resultado final apenas deixou alguns loops `for` mais organizados neste caso. Em C# 2, porém, foi fácil. Dentro da classe que representa o horário, simplesmente adicionei uma propriedade:

```
public IEnumerable<DateTime> DateRange {
```

```
    pegar
    {
        for (DateTime dia = DataInício;
            dia <= DataFinal; dia =
            dia.AddDays(1))
        {
            dia de retorno do rendimento;
        }
    }
}
```

Isso moveu o loop original para a classe `timetable`, mas tudo bem - é muito melhor que ele seja encapsulado lá, em uma propriedade que apenas percorre os dias, gerando um de cada vez, do que estar no código de negócios que estava lidando com aqueles dias. Se eu

Sempre quis torná-lo mais complexo (pulando finais de semana e feriados, por exemplo), eu poderia fazer isso em um só lugar e colher os frutos em qualquer lugar.

Essa pequena mudança fez uma grande melhoria na legibilidade da base de código. Acontece que parei de refatorar naquele ponto do código comercial. Pensei em introduzir um tipo Range<T> para representar um intervalo de uso geral, mas como eu só precisava dele nesta situação, não fazia sentido despender mais esforço no problema. Acontece que foi uma atitude sábia. Na primeira edição deste livro, criei exatamente esse tipo – mas ele tinha algumas deficiências que eram difíceis de resolver de uma maneira amigável ao livro. Eu o redesenhei significativamente para minha biblioteca de utilitários, mas ainda tenho algumas dúvidas. Tipos como esse muitas vezes parecem mais simples do que realmente são, e logo você acaba com um caso difícil para ser tratado a cada passo. Os detalhes das dificuldades que encontrei não pertencem realmente a este livro - são mais pontos sobre design geral do que sobre C# - mas são interessantes por si só, então os escrevi como um artigo sobre o site do livro (veja <http://mng.bz/GAmS>).

O próximo exemplo é um dos meus favoritos – ele demonstra tudo o que adoro nos blocos iteradores.

### 6.3.2 Iterando sobre linhas em um arquivo

Com que frequência você leu um arquivo de texto linha por linha? É uma tarefa incrivelmente comum. A partir do .NET 4, o framework finalmente fornece um método para tornar isso mais fácil em File.ReadLines, mas se você estiver usando uma versão anterior do framework, poderá escrever sua própria versão com muita facilidade, como mostrarei em as próximas páginas.

Tenho medo de pensar quantas vezes escrevi códigos como este:

```
usando (leitor TextReader = File.OpenText (nome do arquivo)) {  
  
    linha de corda;  
    while ((linha = leitor.ReadLine ()) != null) {  
  
        // Faça algo com linha  
    }  
}
```

Temos quatro conceitos separados aqui:

- ÿ Como obter um TextReader
- ÿ Gerenciando o tempo de vida do TextReader ÿ Iterando
- sobre as linhas retornadas por TextReader.ReadLine ÿ Fazendo algo com cada uma dessas linhas

Apenas o primeiro e o último deles são geralmente específicos para a situação – o gerenciamento do tempo de vida e o mecanismo de iteração são apenas códigos padronizados. (Pelo menos, o gerenciamento vitalício é simples em C#. Graças a Deus por usar instruções!) Existem duas maneiras de melhorar as coisas. Você poderia usar um delegado - escreva um método utilitário que tome um leitor e um delegado como parâmetros, chame o delegado para cada linha do arquivo e feche o leitor no final. Isso é frequentemente usado como exemplo

de encerramentos e delegados, mas há uma alternativa que considero mais elegante e que se adapta muito melhor ao LINQ. Em vez de passar sua lógica para um método como um delegado, você pode usar um iterador para retornar uma única linha por vez do arquivo, para poder usar o loop foreach normal.

Você pode conseguir isso usando um tipo inteiro implementando `IEnumerable<string>` (eu tenho uma classe `LineReader` em minha biblioteca `MiscUtil` para essa finalidade), mas um método independente em outra classe também funcionará bem. É muito simples, como prova a próxima listagem.

#### Listagem 6.8 Fazendo um loop nas linhas de um arquivo usando um bloco iterador

```
static IEnumerable<string> ReadLines(string nome do arquivo) {
    usando (leitor TextReader = File.OpenText (nome do arquivo)) {
        linha de corda;
        while ((linha = leitor.ReadLine()) != null) {
            linha de retorno de rendimento;
        }
    }
    ...
    foreach (linha de string em ReadLines("test.txt")) {
        Console.WriteLine(linha);
    }
}
```

O corpo do método é exatamente o que você tinha antes, exceto que o que você está fazendo com a linha é entregá-lo ao chamador quando ele itera sobre a coleção. Como antes, você abre o arquivo, lê uma linha por vez e então fecha o leitor quando terminar... embora o conceito de “quando você terminar” seja mais interessante neste caso do que com uma instrução `using` em um método normal, onde o controle de fluxo é mais óbvio.

É por isso que é importante que o loop foreach descarte o iterador – porque é isso que garante que o leitor seja limpo. A instrução `using` no método iterador atua como um bloco `try/finally`; esse bloco finalmente será executado se você chegar ao final do arquivo ou chamar `Dispose` no `IEnumerator<string>` quando estiver no meio do caminho. Seria possível chamar o código para abusar do `IEnumerator <string>` retornado por `ReadLines(...).GetEnumerator()` e acabar com um vazamento de recursos, mas esse geralmente é o caso com `IDisposable` - se você não chamar `Dispose`, você poderá vazar recursos. Porém, raramente é um problema, pois o foreach faz a coisa certa. É importante estar ciente desse possível abuso — se você dependesse de algum tipo de bloqueio `try/finally` em um iterador para conceder alguma permissão e depois removê-lo novamente mais tarde, isso realmente seria uma falha de segurança.

Este método encapsula os três primeiros dos quatro conceitos listados anteriormente, mas é um pouco restritivo. É razoável agrupar os aspectos de gerenciamento de tempo de vida e de iteração, mas e se você quiser ler texto de um fluxo de rede? Ou se

você deseja usar uma codificação diferente de UTF-8? Você precisa colocar a primeira parte de volta no controle do chamador, e a abordagem mais óbvia seria alterar a assinatura do método para aceitar um TextReader, assim:

```
static IEnumerable<string> ReadLines(TextReader leitor)
```

Esta é uma má ideia, no entanto. Você deseja assumir a propriedade do leitor para poder limpá-lo de maneira conveniente para o chamador, mas o fato de assumir a responsabilidade pela limpeza significa que você precisa *limpá-lo*, desde que o chamador use o resultado de maneira sensata. O problema é que, se algo acontecer antes da primeira chamada para MoveNext(), você nunca terá nenhuma chance de limpar: nenhum código será executado. O `IEnumerable<string>` em si não é descartável e, ainda assim, teria armazenado esse pedaço de estado, que exigia descarte. Outro problema ocorreria se Getenumerator() fosse chamado duas vezes: isso deveria gerar dois iteradores independentes, mas ambos usariam o mesmo leitor. Você poderia atenuar um pouco isso alterando o tipo de retorno para `IEnumerator<string>`, mas isso significaria que o resultado não poderia ser usado em um loop foreach e você ainda não conseguiria executar nenhum código de limpeza se nunca chegasse tão longe como a primeira chamada MoveNext(). Felizmente, há uma maneira de contornar isso.

Assim como o código não pode ser executado imediatamente, você não precisa do leitor imediatamente. O que você precisa é de uma maneira de atrair o leitor quando precisar. Você poderia usar uma interface para representar a ideia de “Posso fornecer um TextReader quando você quiser”, mas a ideia de uma interface de método único geralmente deve fazer com que você procure um delegado. Em vez disso, vou trapacear um pouco apresentando um delegado que faz parte do .NET 3.5. Está sobrecarregado por diferentes números de parâmetros de tipo, mas você só precisa de um:

```
delegado público TResult Func<TResult>()
```

Como você pode ver, este delegado não possui parâmetros, mas retorna um resultado do mesmo tipo que o parâmetro de tipo. É um fornecedor clássico ou assinatura de fábrica. Nesse caso, você deseja obter um TextReader, para poder usar `Func<TextReader>`. As mudanças no método são simples:

```
static IEnumerable<string> ReadLines(Func<TextReader> provedor) {
```

```
    usando (TextReader leitor = provedor()) {  
  
        linha de corda;  
        while ((linha = leitor.ReadLine()) != null) {  
  
            linha de retorno de rendimento;  
        }  
    }  
}
```

Agora o recurso é adquirido pouco antes de você precisar dele e, nesse ponto, você está no contexto de `IDisposable`, para poder liberar o recurso no momento apropriado. Além disso, se `GetEnumerator()` for chamado várias vezes no valor retornado, cada chamada resultará na criação de um TextReader independente.

Você pode facilmente usar métodos anônimos para adicionar sobrecargas a arquivos abertos, especificando opcionalmente uma codificação:

```
static IEnumerable<string> ReadLines(string nome do arquivo) {
    retornar ReadLines (nome do arquivo, Encoding.UTF8);
}

static IEnumerable<string> ReadLines(string nome do arquivo, codificação codificação) {
    return ReadLines (delegado { return
        File.OpenText (nome do arquivo, codificação); });
}
```

Este exemplo usa genéricos, um método anônimo (que captura parâmetros) e um bloco iterador. Tudo o que falta são tipos de valor anuláveis e você terá todos os principais recursos do C# 2. Usei esse código em diversas ocasiões e ele é sempre muito mais limpo do que o código complicado com o qual começamos. Como mencionei anteriormente, se você estiver usando uma versão recente do .NET, você terá tudo isso disponível em `File.ReadLines` de qualquer maneira, mas ainda é um bom exemplo de quão úteis os blocos iteradores podem ser.

Como exemplo final, vamos experimentar o LINQ pela primeira vez, embora usaremos apenas C# 2.

### **6.3.3 Filtrando itens preguiçosamente usando um bloco iterador e um predicado**

Mesmo que ainda não tenhamos começado a analisar o LINQ adequadamente, tenho certeza de que você tem alguma ideia do que se trata: ele permite consultar dados de maneira simples e poderosa em diversas fontes de dados, como coleções na memória e bancos de dados. O C# 2 não possui nenhuma integração de linguagem para expressões de consulta, nem expressões lambda e métodos de extensão que podem torná-lo tão conciso, mas você ainda pode obter alguns dos mesmos efeitos.

Um dos principais recursos do LINQ é a filtragem com o método `Where`. Você fornece uma coleção e um predicado, e o resultado é uma consulta avaliada lentamente que produzirá apenas os itens da coleção que correspondem ao predicado. É um pouco como `List<T>.FindAll`, mas é preguiçoso e funciona com qualquer `IEnumerable<T>`. Uma das coisas bonitas do LINQ<sup>5</sup> é que a inteligência está no design. É bastante simples implementar o LINQ to Objects como provaremos agora, pelo menos para o método `Where`. Ironicamente, embora a maioria dos recursos da linguagem que fazem o LINQ brilhar façam parte do C# 3, eles são quase todos sobre como você pode acessar métodos como `Where`, e não como eles são implementados.

A listagem a seguir mostra um exemplo completo, incluindo validação de argumento simples, e usa o filtro para exibir todas as diretivas `using` no arquivo de origem que contém o próprio código de amostra.

---

<sup>5</sup> Ou, para ser mais preciso, LINQ to Objects. Os provedores LINQ para bancos de dados e similares são muito mais complexos.

## Listagem 6.9 Implementando o método Where do LINQ usando blocos iteradores

```

public static IEnumerable<T> Where<T>(IEnumerable<T> fonte,
                                         Predicado<T> predicado)
{
    if (fonte == nulo || predicado == nulo) {
        lançar novo ArgumentNullException();
    }
    return WhereImpl(fonte, predicado);
}

private static IEnumerable<T> WhereImpl<T>(IEnumerable<T> fonte,
                                         Predicado<T> predicado)
{
    foreach (item T na fonte) {
        if (predicado(item)) {
            item de retorno de rendimento;
        }
    }
}
...
Ienumerable<string> linhas = LineReader.ReadLines("../FakeLinq.cs");
Predicado<string> predicado = delegado(string
linha) { return line.StartsWith("usando"); };
foreach (string linha em Where(linhas,
predicado)) {
    Console.WriteLine(linha);
}

```

**B** verifica ansiosamente os argumentos

**C** processa dados preguiçosamente

**D** Testa o item atual em relação ao predicado

Este exemplo divide a implementação em duas partes: validação de argumentos e a lógica comercial real de filtragem. É um pouco feio, mas totalmente necessário para um tratamento sensato de erros. Suponha que você coloque tudo no mesmo método — o que aconteceria quando você chamassem `Where<string>(null, null)`? A resposta é *nada...ou*, pelo menos, a exceção desejada não seria lançada. Isso se deve à semântica preguiçosa dos blocos iteradores: nenhum código no corpo do método é executado até a primeira chamada para `Move-Next()`, como você viu na seção 6.2.2. Normalmente, você deseja verificar as pré-condições do método com entusiasmo — não faz sentido atrasar a exceção e isso apenas torna a depuração mais difícil.

A solução padrão para isso é dividir o método pela metade, como na listagem 6.9. Primeiro você verifica os argumentos **B** em um método normal e, em seguida, chama o método implementado usando um bloco iterador para processar preguiçosamente os dados como e quando são solicitados **C**.

O bloco iterador em si é incrivelmente simples: para cada item da coleção original, você testa o predicado **D** e produz o valor se ele corresponder. Se não corresponder, você tenta o próximo item e assim por diante até encontrar algo que *corresponda* ou até ficar sem itens. É simples, mas uma implementação em C# 1 teria sido muito mais difícil de seguir (e não poderia ter sido genérica, é claro).

A parte final do código para demonstrar o método em ação usa o anterior exemplo para fornecer os dados – neste caso, o código-fonte da implementação. O predicado simplesmente testa uma linha para ver se ela começa com “usando” – poderia conter uma lógica muito mais complicada, é claro. Criei variáveis separadas para os dados e o predicado apenas para deixar a formatação mais clara, mas tudo poderia ter sido escrito in-line. É importante notar a principal diferença entre este exemplo e o equivalente que poderia ter sido alcançado usando `File.ReadAllLines` e `Array.FindAll<string>`. Esta implementação é totalmente preguiçosa e contínua. Apenas um único linha do arquivo de origem é sempre necessária na memória por vez. Claro, isso não importaria neste caso específico em que o arquivo é pequeno - mas se você imaginar um arquivo de log multigigabyte, você pode ver os benefícios dessa abordagem.

Espero que esses exemplos tenham lhe dado uma ideia de por que os blocos iteradores são tão importante - bem como talvez um desejo de se apressar e descobrir mais sobre o LINQ. Antes disso, gostaria de mexer um pouco com a sua mente e apresentar-lhe uma visão completa uso bizarro (mas muito legal) de iteradores.

#### 6.4 Código pseudossíncrono com o tempo de execução de simultaneidade e coordenação

O Simultaneidade e Coordenação Runtime (CCR) é uma biblioteca desenvolvida pela Microsoft para oferecem uma maneira alternativa de escrever código assíncrono que seja adequado para tarefas complexas coordenação. No momento em que este artigo foi escrito, ele estava disponível apenas como parte do Microsoft Robotics Studio (consulte <http://www.microsoft.com/robotics>). A Microsoft tem colocado muitos recursos em simultaneidade em vários projetos, mas notavelmente a Task Parallel Library introduzida no .NET 4 e os recursos de linguagem assíncrona no C# 5 (suportado por muitas APIs assíncronas). Mas eu queria usar o CCR para mostrar a você como os blocos iteradores podem mudar todo o modelo de execução. Na verdade, não é coincidência que esta incursão inicial em uma abordagem alternativa à simultaneidade use iterator blocos para alterar o modelo de execução; as semelhanças entre as máquinas de estado gerados para blocos iteradores e aqueles usados para funções assíncronas em C# 5 são impressionante.

O código de exemplo funciona (em serviços fictícios), mas as ideias são mais importante do que os detalhes.

Suponha que você esteja escrevendo um servidor que precisa lidar com muitas solicitações. Como parte de Ao lidar com essas solicitações, primeiro você precisa chamar um serviço da Web para buscar um token de autenticação e, em seguida, usar esse token para obter dados de duas fontes de dados independentes. (digamos, um banco de dados e outro serviço web). Em seguida, você processa esses dados e retorna o resultado. Cada um dos estágios de busca pode demorar um pouco, talvez alguns segundos. Normalmente você pode considerar a rota síncrona simples ou a rota assíncrona de estoque abordagem. A versão síncrona pode ser parecida com isto:

```
HoldingsValue ComputeTotalStockValue(string usuário, string senha)
{
    Token token = AuthService.Check(usuário, senha);
    Ações holdings = DbService.GetStockHoldings(token);
```

```

    Taxas de StockRates = StockService.GetRates(token); retornar
    ProcessStocks(estoque, taxas);
}

```

Isso é fácil de entender, mas se cada solicitação levar 2 segundos, toda a operação levará 6 segundos e amarrará um thread durante todo o tempo em que estiver em execução. Se quiser escalar até centenas de milhares de solicitações executadas em paralelo, você terá problemas.

Agora vamos considerar uma versão assíncrona bastante simples, que evita amarrar um thread quando nada está acontecendo<sup>6</sup> e usa chamadas paralelas sempre que possível:

```

void StartComputingTotalStockValue(string usuário, string senha) {

    AuthService.BeginCheck(usuário, senha, AfterAuthCheck, null);
}

void AfterAuthCheck(resultado IAsyncResult) {

    Token token = AuthService.EndCheck(resultado); IAsyncResult
    holdingsAsync = DbService.BeginGetStockHoldings
        (token, nulo, nulo);
    StockService.BeginGetRates(token, AfterGetRates, holdingsAsync);
}

void AfterGetRates(resultado IAsyncResult) {

    IAsyncResult holdingsAsync = (IAsyncResult)resultado.AsyncState; Taxas de StockRates =
    StockService.EndGetRates(resultado); Ações de participações =
    DbService.EndGetStockHoldings(holdingsAsync); OnRequestComplete(ProcessStocks(estoque, taxas));

}

```

Isto é muito mais difícil de ler e compreender – e esta é apenas uma versão simples. A coordenação das duas chamadas paralelas só é possível de forma simples porque não é necessário passar nenhum outro estado e mesmo assim não é o ideal. Se a chamada de serviço de estoque for concluída rapidamente, você ainda bloqueará um thread do pool de threads aguardando a conclusão da chamada ao banco de dados. Mais importante ainda, está longe de ser óbvio o que está acontecendo, porque o código alterna entre métodos diferentes.

Agora você deve estar se perguntando onde os iteradores entram em cena. Bem, os blocos iteradores fornecidos pelo C# 2 permitem efetivamente pausar a execução atual em determinados pontos do fluxo através do bloco e depois voltar ao mesmo local, com o mesmo estado. As pessoas inteligentes que projetaram o CCR perceberam que isso é exatamente o que é necessário para um estilo de codificação de passagem de continuação. Você precisa informar ao sistema que há certas operações que você precisa executar – incluindo iniciar outras operações de forma assíncrona – mas que você terá prazer em esperar até que as operações assíncronas terminem antes de continuar. Você faz isso fornecendo ao CCR uma implementação de `IEnumerator<ITask>` (onde `ITask` é uma interface definida pelo CCR). Aqui está um código para obter os mesmos resultados usando este estilo:

---

<sup>6</sup> Bem, principalmente. Ainda pode ser ineficiente, como você verá em breve.

```
static IEnumerator<ITask> ComputeTotalStockVal(string user, string pass) {
    token de string = nulo; rendimento
    retorno Arbitro.Receive (falso, AuthService.CcrCheck (usuário, senha),
        delegado(string t) { token = t; });

    IEnumerável<Holding> ações = null;
    IDictionary<string, decimal> taxas = null; rendimento retorno
    Arbitro.JoinedReceive (false, DbService.CcrGetStockHoldings (token),
        StockService.CcrGetRates (token), delegado(IEnumerável<Holding>
            s, IDictionary<string, decimal> r) { ações = s; taxas =
            r; });

    OnRequestComplete(ComputeTotal (estoques, taxas));
}
```

Confuso? Certamente fiquei quando o vi pela primeira vez, mas agora estou impressionado com o quanto legal é. O CCR chama seu código (com uma chamada para MoveNext no iterador) e você executa até e incluindo a primeira instrução de retorno de rendimento. O método CcrCheck dentro do AuthService inicia uma solicitação assíncrona e o CCR aguarda (sem usar um thread dedicado) até que seja concluído, chamando o delegado fornecido para tratar o resultado. Em seguida, ele chama MoveNext novamente e seu método continua. Desta vez, você inicia duas solicitações em paralelo e pede ao CCR para chamar outro delegado com os resultados de ambas as operações quando ambas *terminarem*. Depois disso, MoveNext é chamado pela última vez e você conclui o processamento da solicitação.

Embora seja obviamente mais complicado que a versão síncrona, ainda está tudo em um método, ele é executado na ordem escrita e o próprio método pode conter o estado (nas variáveis locais, que se tornam estado no tipo extra gerado pelo compilador). É totalmente assíncrono, usando o mínimo de threads possível. Não mostrei nenhum tratamento de erros, mas isso também está disponível de uma forma sensata que força você a pensar sobre o problema nos locais apropriados.

Deliberadamente, não entrei em detalhes da classe Arbiter, da interface ITask e assim por diante aqui. Não estou tentando promover a RCC nesta seção, embora seja fascinante ler e experimentar; Suspeito que as funções assíncronas no C# 5 terão muito mais impacto nos desenvolvedores convencionais. Meu objetivo aqui foi mostrar que os iteradores podem ser usados em contextos radicalmente diferentes que têm pouco a ver com coleções tradicionais. No cerne deste uso deles está a ideia de um estado

máquina: dois dos aspectos complicados do desenvolvimento assíncrono são lidar com o estado e pausar efetivamente até que algo interessante aconteça. Os blocos iteradores são uma solução natural para ambos os problemas, embora você veja no Capítulo 15 como o suporte a linguagens mais direcionadas torna as coisas muito mais limpas.

## 6.5 Resumo

C# suporta muitos padrões indiretamente, em termos de ser viável implementá-los em C#. Mas relativamente poucos padrões são suportados *diretamente* em termos de recursos de linguagem direcionados especificamente a um determinado padrão. Em C# 1, o padrão do iterador era

diretamente suportado do ponto de vista do código de chamada, mas não da perspectiva da coleção que está sendo iterada. Escrever uma implementação correta de `IEnumerable` era demorado e propenso a erros, sem ser interessante. No C# 2, o compilador faz todo o trabalho mundano para você, construindo uma máquina de estado para lidar com a natureza de retorno de chamada dos iteradores.

Deve-se notar que os blocos iteradores têm um aspecto em comum com os métodos anônimos que você viu no capítulo 5, embora os recursos reais sejam muito diferentes. Em ambos os casos, tipos extras podem ser gerados e uma transformação de código potencialmente complicada é aplicada à fonte original. Compare isso com C# 1, onde a maioria das transformações para açúcar sintático (`lock`, `using` e `foreach` sendo os exemplos mais óbvios) eram diretas. Você verá essa tendência de compilação mais inteligente continuar em quase todos os aspectos do C# 3.

Mostrei uma funcionalidade relacionada ao LINQ neste capítulo: filtrar uma coleção. `IEnumerable<T>` é um dos tipos mais importantes no LINQ, e se você quiser escrever seus próprios operadores LINQ sobre o LINQ to Objects,<sup>7</sup> você será eternamente grato à equipe do C# por incluir blocos iteradores no linguagem.

Além de ver alguns exemplos reais do uso de iteradores, vimos como uma biblioteca específica os usou de uma forma bastante radical, o que tem pouco a ver com o que provavelmente vem à mente quando você pensa sobre a iteração em um coleção. Vale a pena ter em mente que muitas linguagens também já analisaram esse tipo de problema antes - na ciência da computação, o termo *coroutine* é aplicado a conceitos dessa natureza, e é assim que eles são chamados no conjunto de ferramentas de desenvolvimento de jogos Unity 3D, onde novamente eles são usados para assincronia. Diferentes línguas historicamente os apoiaram em maior ou menor grau, sendo algumas vezes aplicáveis truques para simulá-los. Por exemplo, Simon Tatham tem um excelente artigo sobre como até mesmo C pode expressar corrotinas se você estiver disposto a alterar um pouco os padrões de codificação (veja seu artigo "Corrotinas em C" em <http://mng.bz/H8YX>). Você viu que o C# 2 torna as corrotinas fáceis de escrever e usar.

Agora que você viu algumas mudanças de linguagem importantes e às vezes surpreendentes focadas em alguns recursos principais, o próximo capítulo será uma mudança de ritmo. Ele descreve uma série de pequenas mudanças que tornam o C# 2 mais agradável de trabalhar do que seu antecessor. Os designers aprenderam com os pequenos problemas do passado e produziram uma linguagem que tem menos arestas, mais espaço para lidar com casos estranhos de compatibilidade com versões anteriores e uma história melhor sobre como trabalhar com o código gerado. Cada recurso é relativamente simples, mas existem alguns deles.

<sup>7</sup> Isso é menos assustador e mais divertido do que parece. Veremos algumas diretrizes sobre esse tópico no capítulo 12.

## Concluindo C# 2: os recursos finais

### Este capítulo cobre

- ÿ Tipos parciais
- ÿ Classes estáticas
- ÿ Acesso separado à propriedade getter/setter
- ÿ Aliases de namespace
- ÿ Diretrizes pragmáticas
- ÿ Buffers de tamanho fixo
- ÿ Reuniões de amigos

Até agora, examinamos os quatro maiores novos recursos do C# 2: genéricos, tipos anuláveis, melhorias de delegação e blocos iteradores. Cada um aborda um requisito bastante complexo, e é por isso que os aprofundamos. Os novos recursos restantes do C# 2 eliminam algumas arestas do C# 1. São pequenas imperfeições que os designers da linguagem decidiram corrigir - sejam áreas onde a linguagem precisava de um pouco de melhoria por si só, ou onde a experiência de trabalhar com geração de código e código nativo poderia ser mais agradável.

Com o tempo, a Microsoft recebeu muitos comentários da comunidade C# (e de seus próprios desenvolvedores, sem dúvida) sobre áreas onde o C# não brilhou tão intensamente quanto deveria. Várias mudanças menores foram incluídas no C# 2 junto com as maiores, todas eliminando alguns desses pequenos pontos problemáticos.

Nenhum dos recursos deste capítulo é particularmente difícil e iremos analisá-los rapidamente. Não subestime o quão importantes eles são. Só porque um tópico pode ser explorado em poucas páginas não significa que seja inútil. É provável que você use alguns desses recursos com frequência. Aqui está um rápido resumo dos recursos abordados neste capítulo e seus usos, para que você saiba o que esperar:

ÿ *Tipos parciais*—A capacidade de escrever o código para um tipo em vários arquivos de origem.

Isso é particularmente útil para tipos em que parte do código é gerada automaticamente e o restante é escrito manualmente.

ÿ *Classes estáticas* — Para organizar as classes utilitárias para que o compilador possa identificar quando você está tentando usá-las de forma inadequada e deixar suas intenções mais claras.

ÿ *Acesso separado às propriedades getter/setter*—Finalmente, a capacidade de ter um getter público e um setter privado para propriedades! (Essa não é a única combinação disponível, mas é a mais comum.) ÿ

*Aliases de namespace* — Saídas para situações complicadas em que os nomes de tipos não são exclusivos. ÿ *Diretivas Pragma*—Instruções específicas do compilador para ações como suprimir avisos específicos para uma seção específica do código.

ÿ *Buffers de tamanho fixo*—Mais controle sobre como as estruturas lidam com arrays em código inseguro. ÿ *InternalsVisibleToAttribute (assemblies amigos)*—Um recurso que abrange linguagem, estrutura e tempo de execução, que permite aos assemblies selecionados mais acesso quando necessário.

Você pode estar ansioso para entrar nas coisas sexy do C # 3 a esta altura, e eu não o culpo. Nada neste capítulo vai colocar fogo no mundo — mas cada um desses recursos pode tornar sua vida mais agradável ou, em alguns casos, tirá-lo de um buraco. Tendo diminuído um pouco suas expectativas, o primeiro recurso é realmente muito bacana.

## 7.1 Tipos parciais

A primeira mudança que veremos é em resposta à luta pelo poder que normalmente acontecia ao usar geradores de código com C# 1. Para Windows Forms, o designer no Visual Studio exigia suas próprias regiões de código que não podiam ser tocadas pelos desenvolvedores , dentro do mesmo arquivo que os desenvolvedores tiveram que editar para funcionalidade da interface do usuário. Esta era claramente uma situação frágil.

Em outros casos, os geradores de código criam fontes que são compiladas junto com o código escrito manualmente. No C# 1, adicionar funcionalidades extras envolvia derivar novas classes das geradas automaticamente, o que é feio. Existem muitos outros cenários em que um elo desnecessário na cadeia de herança pode causar problemas ou reduzir o encapsulamento. Por exemplo, se duas partes diferentes do seu código quiserem chamar uma à outra, você

precisa de métodos virtuais para o tipo pai chamar o filho e métodos protegidos para a situação inversa, onde normalmente você usaria dois métodos não virtuais privados.

O C# 2 permite que mais de um arquivo contribua para um tipo, e os IDEs podem estender essa noção para que parte do código usado para um tipo possa nem mesmo ser visível como código-fonte C#. Os tipos criados a partir de vários arquivos de origem são chamados de *tipos parciais*.

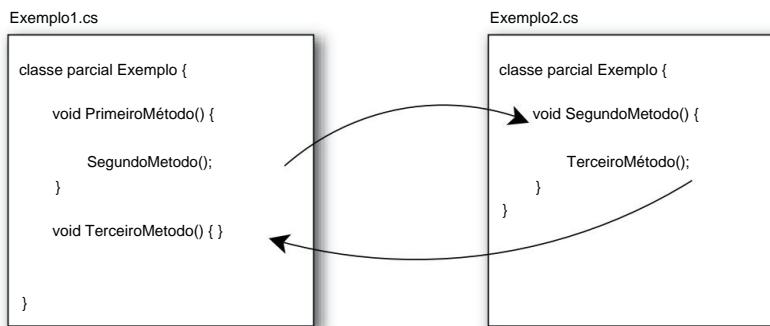
Nesta seção também discutiremos *métodos parciais*, que são relevantes apenas em tipos parciais e permitem uma maneira rica, mas eficiente, de adicionar ganchos escritos manualmente em código gerado automaticamente. Na verdade, esse é um recurso do C# 3 (desta vez baseado no feedback sobre o C# 2), mas é mais lógico discuti-lo quando examinamos os tipos parciais do que esperar até a próxima parte do livro.

### 7.1.1 Criando um tipo com múltiplos arquivos

Criar um tipo parcial é muito fácil - você só precisa incluir a palavra-chave contextual parcial na declaração do tipo em cada arquivo em que ocorre. Um tipo parcial pode ser declarado em quantos arquivos você desejar, embora todos os exemplos neste seção use dois.

O compilador combina efetivamente todos os arquivos de origem antes de compilar. Isso significa que o código em um arquivo pode chamar o código em outro e vice-versa, como mostra a figura 7.1 — não há necessidade de referências diretas ou outros truques.

Você não pode escrever metade de um membro em um arquivo e metade em outro - cada indivíduo membro *real* deve estar completamente contido em seu próprio arquivo. Você não pode iniciar um método em um arquivo e terminá-lo em outro, por exemplo.<sup>1</sup> Existem algumas restrições óbvias sobre as declarações do tipo — as declarações precisam ser compatíveis. Qualquer arquivo pode especificar interfaces a serem implementadas (e elas não precisam ser implementadas nesse arquivo), qualquer arquivo pode especificar o tipo base e qualquer arquivo pode especificar restrições em um parâmetro de tipo. Mas se vários arquivos especificarem um tipo base, esses tipos base deverão ser iguais, e se vários arquivos especificarem restrições de tipo, as restrições deverão ser idênticas. O



**Figura 7.1** O código em tipos parciais é capaz de ver todos os membros do tipo, independentemente do arquivo em que cada membro está.

<sup>1</sup> Há uma exceção aqui: tipos parciais podem conter tipos parciais aninhados espalhados pelo mesmo conjunto de arquivos.

a listagem a seguir mostra um exemplo da flexibilidade oferecida (embora não faça nada mesmo remotamente útil).

### Listagem 7.1 Demonstração de mistura de declarações de tipo parcial

```
//Exemplo1.cs
usando o sistema;

classe parcial Exemplo<TFirst, TSecond>
    : IEquatable<string>
    onde TPrimeiro: classe
{
    bool público Igual a (string outro)
    {
        retorna falso;
    }
}

// Exemplo2.cs
usando o sistema;

classe parcial Exemplo<TFirst, TSecond>
    : EventArgs, IDisposable
{
    público vazio Dispose()
    {
    }
}
```

- B** Especifica a interface e a restrição de parâmetro de tipo
- C** Implementos IEquatable<string>
- D** Especifica a classe base e a interface
- E** implementa IDisposable

Enfatizo que esta listagem tem apenas o propósito de falar sobre o que é legal em uma declaração – os tipos envolvidos foram escolhidos apenas por conveniência e familiaridade. Você pode ver que ambas as declarações (B e D) contribuem para a lista de interfaces que devem ser implementado. Neste exemplo, cada arquivo implementa as interfaces que declara e esse é um cenário comum, mas seria legal adiar a implementação de

**IDisposable** E para Example1.cs e a implementação de **IEquatable<string>** C para Exemplo2.cs. Eu mesmo usei a capacidade de especificar interfaces separadamente da implementação, encapsulando métodos com a mesma assinatura gerada para vários tipos diferentes em uma interface. O gerador de código não sabe sobre o interface, portanto não sabe declarar que o tipo a implementa.

Somente a primeira declaração B especifica quaisquer restrições de tipo e apenas a segunda D especifica uma classe base. Se a primeira declaração B tivesse especificado uma classe base, teria para ser EventArgs, e se a segunda declaração tivesse especificado alguma restrição de tipo, eles teriam que ser exatamente como no primeiro. Em particular, você não pode especificar uma restrição de tipo para TSecond na segunda declaração, embora não seja mencionado na primeira. Ambos os tipos precisam ter o mesmo modificador de acesso, se houver — você não pode tornar uma declaração interna e a outra pública, por exemplo. Essencialmente, as regras relativas à combinação de arquivos permitem flexibilidade na maioria dos casos, ao mesmo tempo que incentivam a consistência.

Em tipos de arquivo único, a inicialização de variáveis de membro e estáticas é garantida ocorrer na ordem em que aparecem no arquivo, mas não há ordem garantida quando vários arquivos estão envolvidos. Confiar na ordem de declaração dentro do arquivo é frágil

para começar - isso deixa seu código aberto a bugs sutis se um desenvolvedor decidir mover as coisas de maneira "menos prejudicial" - então vale a pena evitar essa situação sempre que possível. Você deve evitá-lo *especialmente* com tipos parciais.

Agora que você sabe o que pode e o que não pode fazer, vamos examinar mais de perto por que você deseja *fazer* isso.

### 7.1.2 Usos de tipos parciais

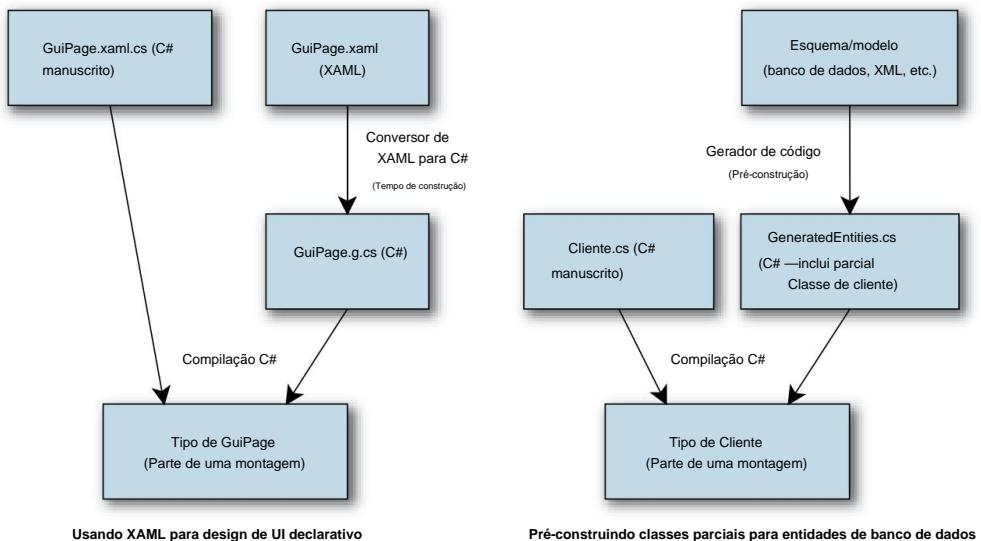
Como mencionei anteriormente, os tipos parciais são úteis principalmente em conjunto com designers e outros geradores de código. Se um gerador de código precisar modificar um arquivo de propriedade de um desenvolvedor, sempre haverá o risco de algo dar errado. Com o modelo de tipos parciais, um gerador de código pode possuir o arquivo onde irá trabalhar e sobreescriver completamente o arquivo inteiro sempre que desejar.

Alguns geradores de código podem optar por não gerar um arquivo C# até que a construção esteja bem encaminhada. Por exemplo, Snippy possui arquivos Extensible Application Markup Language (XAML) que descrevem a interface do usuário. Quando o projeto é compilado, cada arquivo XAML é convertido em um arquivo C# no diretório obj (os nomes dos arquivos terminam com .cs para mostrar que foram gerados) e compilado junto com a classe parcial que fornece código extra para esse tipo (normalmente manipuladores de eventos e código de construção extra). Isso evita completamente que os desenvolvedores ajustem o código gerado, pelo menos sem chegar ao extremo de hackear o arquivo de construção.

Tive o cuidado de usar a frase *gerador de código* em vez de *designer* porque existem muitos geradores de código além de designers. Por exemplo, no Visual Studio, os proxies de serviços da Web são gerados como classes parciais, e você pode ter suas próprias ferramentas que geram código com base em outras fontes de dados. Um exemplo razoavelmente comum disso é o mapeamento objeto-relacional (ORM) – algumas ferramentas ORM usam descrições de entidades de banco de dados a partir de um arquivo de configuração (ou diretamente do banco de dados) e geram classes parciais que representam essas entidades. Da mesma forma, minha porta .NET da estrutura de serialização do Google Proto col Buffers gera classes parciais – um recurso que se mostrou útil mesmo dentro da própria implementação.

Isso torna simples adicionar comportamento ao tipo – substituindo métodos virtuais da classe base, adicionando novos membros com lógica de negócios e assim por diante. É uma ótima maneira de permitir que o desenvolvedor e a ferramenta trabalhem juntos, em vez de brigar constantemente sobre quem está no comando.

Um cenário que às vezes é útil é gerar um arquivo contendo vários tipos parciais e, em seguida, alguns desses tipos são aprimorados em outros arquivos, com um arquivo gerado manualmente por tipo. Voltando ao exemplo do ORM, a ferramenta poderia gerar um único arquivo contendo todas as definições de entidade, e algumas dessas entidades poderiam ter código extra fornecido pelo desenvolvedor, usando um arquivo por entidade. Isso mantém baixo o número de arquivos gerados automaticamente, mas ainda fornece boa visibilidade do código manual envolvido.



**Figura 7.2 Comparação entre pré-compilação XAML e classes de entidade geradas automaticamente**

A Figura 7.2 mostra como os usos de tipos parciais para XAML e entidades são semelhantes, mas com um tempo ligeiramente diferente envolvido quando se trata de criar o gerado automaticamente Código C#.

Um uso um pouco diferente de tipos parciais é como auxílio à refatoração. Às vezes um tipo fica muito grande e assume muitas responsabilidades. Um primeiro passo para dividir o tipo inchado em tipos menores e mais coerentes pode ser dividi-lo em um tipo parcial dois ou mais arquivos. Isso pode ser feito sem riscos e de maneira experimental, movendo métodos entre arquivos até que cada arquivo atenda a uma única preocupação. Apesar de

próximo passo de divisão do tipo ainda está longe de ser automático, deve ser muito mais fácil veja o objetivo final.

Um último uso a ser mencionado: teste unitário. Muitas vezes o conjunto de testes unitários de uma classe pode terminar sendo muito maior do que a própria implementação. Uma maneira de dividir os testes em pedaços mais compreensíveis é usar tipos parciais. Você ainda pode executar facilmente todos os testes para um tipo de uma só vez (já que você ainda tem uma única classe de teste), mas você pode ver facilmente o teste diferentes áreas de funcionalidade em diferentes arquivos. Editando manualmente o projeto arquivo, você pode até ter a mesma expansão pai/filho no Solution Explorer que você veja quando tipos parciais são usados para o código gerado do Visual Studio. Isto não será para gosto de todos, mas descobri que é uma forma útil de gerenciar testes.

Quando os tipos parciais apareceram pela primeira vez no C# 2, ninguém sabia exatamente como eles seriam usados. Um recurso que foi solicitado quase imediatamente foi uma forma de fornecer código extra para métodos gerados chamarem. Essa necessidade foi atendida pelo C# 3 com métodos parciais.

### 7.1.3 Métodos parciais – somente C# 3!

Para reiterar minha explicação anterior, o restante desta parte do livro trata apenas dos recursos do C# 2, mas os métodos parciais não se ajustam a nenhum dos outros recursos do C# 3 e se ajustam bem ao descrever tipos *parciais*. Desculpas por qualquer confusão que isso possa causar.

Voltando ao recurso: às vezes você deseja especificar o comportamento em um arquivo criado manualmente e usar esse comportamento a partir de um arquivo gerado automaticamente. Por exemplo, em uma classe que possui muitas propriedades geradas automaticamente, você pode querer especificar o código a ser executado para validar um novo valor para algumas dessas propriedades. Outro cenário comum é um gerador de código incluir construtores – o código escrito manualmente pode querer se conectar à construção de objetos para definir valores padrão, realizar algum registro e assim por diante.

No C# 2, esses requisitos só podem ser atendidos usando eventos que o código gerado manualmente pode assinar ou fazendo com que o código gerado automaticamente *assuma* que o código manuscrito incluirá métodos com um nome específico – fazendo com que todo o código falhe em compilar, a menos que os métodos relevantes sejam fornecidos. Alternativamente, o código gerado pode fornecer uma classe base com métodos virtuais que não fazem nada por padrão. O código gerado manualmente pode então derivar da classe e substituir alguns ou todos os métodos.

Todas essas soluções são um tanto confusas. Os métodos parciais do C# 3 fornecem efetivamente ganchos *opcionais* que não têm nenhum custo se não forem implementados – quaisquer chamadas para os métodos parciais não implementados são removidas pelo compilador. Isso permite que as ferramentas sejam muito generosas em termos dos ganchos que fornecem. No código compilado, você paga apenas pelo que usa.

É mais fácil entender isso com um exemplo. A listagem a seguir mostra um tipo parcial especificado em dois arquivos, com o construtor no código gerado automaticamente chamando dois métodos parciais, um dos quais é implementado no código gerado manualmente.

#### Listagem 7.2 Um método parcial chamado de um construtor

```
//Gerado.cs usando
System; classe
parcial PartialMethodDemo {

    public PartialMethodDemo() {

        OnConstructorStart();
        Console.WriteLine("Construtor gerado"); OnConstructorEnd();

    }

    void parcial OnConstructorStart(); void parcial
    OnConstructorEnd();
}

// Manuscrito.cs
```

```
usando o sistema;
classe parcial PartialMethodDemo {

    parcial void OnConstructorEnd() {

        Console.WriteLine("Código manual");
    }
}
```

Conforme mostrado na listagem 7.2, os métodos parciais são declarados exatamente como os métodos abstratos: fornecendo a assinatura sem qualquer implementação, mas usando o modificador parcial . Da mesma forma, as implementações reais possuem apenas o modificador parcial , mas são como métodos normais.

Chamar o construtor sem parâmetros de PartialMethodDemo resultaria na impressão do construtor gerado e, em seguida, no código manual . Se você examinasse o IL do construtor, não veria uma chamada para OnConstructorStart porque ele não existe mais — não há nenhum vestígio dele em nenhum lugar do tipo compilado.

Como o método pode não existir, os métodos parciais devem ter um tipo de retorno void e não podem retirar parâmetros . Eles devem ser privados, mas podem ser estáticos e/ou genéricos. Se o método não for implementado em um dos arquivos, toda a instrução que o chama será removida, *incluindo quaisquer avaliações de argumentos*.

Se a avaliação de qualquer um dos argumentos tiver um efeito colateral que você deseja que ocorra independentemente de o método parcial ser implementado ou não, você deverá realizar a avaliação separadamente. Por exemplo, suponha que você tenha o seguinte código:

```
LogEntity(LoadAndCache(id));
```

Aqui LogEntity é um método parcial e LoadAndCache carrega uma entidade do banco de dados e a insere no cache. Você pode querer usar isso:

```
Entidade MyEntity = LoadAndCache(id);
LogEntidade(entidade);
```

Dessa forma, a entidade é carregada e armazenada em cache, independentemente de uma implementação ter sido fornecida para LogEntity. Claro que se a entidade puder ser carregada igualmente mais barato posteriormente, podendo nem ser obrigatória, você deverá deixar o extrato no primeiro formulário e evitar uma carga desnecessária em alguns casos.

Para ser honesto, a menos que você esteja escrevendo seus próprios geradores de código, é mais provável que você implemente métodos parciais do que os declare e chame. Se você estiver apenas implementando-os, não precisará se preocupar com o lado da avaliação dos argumentos.

Em resumo, os métodos parciais em C# 3 permitem que o código gerado interaja com o código escrito à mão de maneira rica, sem quaisquer penalidades de desempenho em situações em que a interação é desnecessária. Esta é uma continuação natural do recurso de tipos parciais do C# 2, que permite um relacionamento muito mais produtivo entre geradores de código e desenvolvedores.

O próximo recurso é totalmente diferente. É uma maneira de contar mais ao compilador sobre a natureza pretendida de um tipo para que ele possa realizar mais verificações tanto no tipo ele mesmo e qualquer código que o utilize.

## 7.2 Classes estáticas

O segundo novo recurso é, de certa forma, completamente desnecessário – apenas torna as coisas mais organizado e elegante quando você escreve *classes utilitárias*.

Todo mundo tem aulas utilitárias. Não vi um projeto significativo em Java ou C# que não tinha pelo menos uma classe composta apenas por métodos estáticos. O exemplo clássico no código do desenvolvedor é um tipo com métodos auxiliares de string, fazendo qualquer coisa, desde escapar, reverter, substituir inteligente – você escolhe. Um exemplo da estrutura é o Classe System.Math .

Os principais recursos de uma classe de utilitário são os seguintes:

- ÿ Todos os membros são estáticos (exceto um construtor privado).
- ÿ A classe deriva diretamente do objeto.
- ÿ Normalmente não há estado algum, a menos que algum cache ou um singleton esteja envolvido.
- ÿ Não há construtores visíveis.
- ÿ A classe é lacrada se o desenvolvedor se lembrar de fazê-lo.

Os dois últimos pontos são opcionais e se não houver construtores visíveis (incluindo protegidos), a classe é *efetivamente* selada de qualquer maneira. Ambos ajudam a tornar o propósito da aula mais óbvio.

A listagem a seguir fornece um exemplo de uma classe de utilitário C# 1. Então veremos como C # 2 melhora as coisas.

### Listagem 7.3 Uma classe de utilitário típica do C# 1

```
classe selada pública NonStaticStringHelper
{
    privado NonStaticStringHelper()
    {

    }

    string estática pública reversa (entrada de string)
    {
        char[] chars = input.ToCharArray();
        Array.Reverse(characters);
        retornar nova string (caracteres);
    }
}
```

**A** Classe de selos para  
**B** evitar derivação

**C** Impede a instânciação  
de outro código

**D** Todos os  
métodos são estáticos

A classe é selada **B** para que ninguém tente derivar dela. A herança deveria ser sobre especialização, e não há nada para se especializar aqui, pois todos os membros são **D** estático , exceto o construtor privado **C**. Esse construtor pode parecer estranho à primeira vista vista - por que tê-lo se é privado e nunca será usado? A razão é que se você não fornece nenhum construtor para uma classe, o compilador C# 1 sempre fornecerá um

*construtor padrão* que é público e sem parâmetros. Neste caso, você não quer nenhum construtor visível, então você tem que fornecer um privado.

Esse padrão funciona razoavelmente bem, mas o C# 2 o torna explícito e evita ativamente que o tipo seja mal utilizado. Primeiro, veremos quais mudanças são necessárias para transformar a listagem 7.3 em uma classe estática adequada, conforme definido em C# 2. Como você pode ver na listagem a seguir, pouca ação é necessária.

#### Listagem 7.4 A mesma classe utilitária da listagem 7.3, mas convertida em uma classe estática C# 2

```
usando o sistema;

classe estática pública StringHelper {

    string estática pública reversa (entrada de string) {

        char[] chars = input.ToCharArray(); Array.Reverse(chareres);
        retornar nova string (chareres);

    }
}
```

Desta vez , você usa o modificador estático na declaração de classe em vez de selado e não inclui nenhum construtor - essas são as únicas diferenças de código. O compilador C# 2 sabe que uma classe estática não deve ter nenhum construtor, portanto não fornece um padrão.

Na verdade, o compilador impõe uma série de restrições na definição da classe:

- ÿ Não pode ser declarado como abstrato ou selado, embora seja implicitamente ambos.
- ÿ Não pode especificar nenhuma interface implementada.
- ÿ Não é possível especificar um tipo base. ÿ Não pode incluir membros não estáticos, incluindo construtores. ÿ
- Não pode incluir nenhum operador. ÿ
- Não pode incluir nenhum membro interno protegido ou protegido .

É importante notar que embora todos os membros devam ser estáticos, você deve torná-los *explicitamente* estáticos. Embora os tipos aninhados sejam membros implicitamente estáticos da classe envolvente, o próprio tipo aninhado pode ser um tipo não estático, se necessário.

O compilador não apenas impõe restrições à definição de classes estáticas — ele também protege contra seu uso indevido. Como sabe que nunca poderá haver instâncias da classe, evita qualquer uso que exija uma. Por exemplo, todos os itens a seguir são inválidos quando StringHelper é uma classe estática:

```
Variável StringHelper = nulo;
StringHelper[] matriz = nulo; public void
Method1(StringHelper x) {} public StringHelper Method1() { return
null; }
List<StringHelper> x = new List<StringHelper>();
```

Nada disso será evitado se a classe seguir o padrão C# 1, mas todos eles são essencialmente inúteis. Resumindo, as classes estáticas em C# 2 não permitem que você faça nada que não podia fazer antes, mas impedem que você faça coisas que *não deveria* fazer de qualquer maneira. Eles também declaram explicitamente suas intenções. Ao tornar uma classe estática, você está dizendo que definitivamente *não deseja* que nenhuma instância seja criada. Não é apenas uma peculiaridade da implementação; é uma escolha de design.

O próximo recurso da lista tem uma sensação mais positiva. Ele é voltado para uma situação específica - embora amplamente encontrada - e permite uma solução que não é feia nem quebra o encapsulamento, que era a escolha disponível no C# 1.

### 7.3 Acesso separado à propriedade getter/setter

Admito que fiquei surpreso quando vi pela primeira vez que o C# 1 não permitia que você tivesse um getter público e um setter privado para propriedades. Essa não é a única combinação de modificadores de acesso proibida pelo C# 1, mas é a mais desejada. Na verdade, em C# 1, tanto o getter quanto o setter precisam ter a mesma acessibilidade — ela é declarada como parte da declaração de propriedade e não como parte do getter ou do setter.

Existem boas razões para querer uma acessibilidade diferente para o getter e o setter. Freqüentemente, você pode querer que alguma validação, registro, bloqueio ou outro código seja executado ao alterar uma variável que suporta a propriedade, mas não deseja tornar a propriedade gravável em código fora da classe. No C# 1, as alternativas eram quebrar o encapsulamento, tornando a propriedade publicamente gravável contra o seu melhor julgamento ou escrever um método SetXXX() na classe para fazer a configuração, o que francamente parece feio quando você está acostumado com propriedades reais.

O C# 2 corrige o problema permitindo que o getter ou o setter tenham explicitamente acesso mais restritivo do que o declarado para a própria propriedade. Isso é mais facilmente visto com um exemplo:

```
nome da sequência;
string pública Nome {
    obter { nome de retorno; }
    conjunto privado {
        // Validação, registro etc. aqui name = value;
    }
}
```

Nesse caso, a propriedade Name é efetivamente somente leitura para todos os outros tipos,<sup>2</sup> mas você pode usar a sintaxe de propriedade familiar para definir a propriedade dentro do próprio tipo. A mesma sintaxe também está disponível para indexadores e também para propriedades. Você *poderia* tornar o set ter mais público que o getter (um getter protegido e um setter público, por exemplo), mas essa é uma situação bastante rara, da mesma forma que as propriedades somente gravação são poucas e distantes entre si em comparação com somente leitura. propriedades.

---

<sup>2</sup> Exceto tipos aninhados, que sempre têm acesso a membros privados de seus tipos envolventes.

**TRIVIA: O ÚNICO LUGAR ONDE “PRIVADO” É NECESSÁRIO** Em todos os outros lugares em C#, o modificador de acesso padrão em qualquer situação é o mais privado possível. Por exemplo, se algo puder ser declarado como privado, o padrão será private se você não especificar nenhum modificador de acesso. Este é um bom elemento de design de linguagem, porque é difícil errar accidentalmente; se quiser que algo seja mais público do que é, você notará quando tentar usá-lo. Mas se você accidentalmente torna algo muito público, o compilador não pode ajudá-lo a identificar o problema. Especificar o acesso de um getter ou setter de propriedade é aquele exceção a esta regra - se você não especificar nada, o padrão é fornecer o getter ou setter o mesmo acesso que a própria propriedade geral.

Observe que você não pode declarar a propriedade em si como privada e tornar o getter público – você só pode tornar um getter ou setter específico *mais* privado do que a propriedade.

Além disso, você não pode especificar um modificador de acesso tanto para o getter quanto para o setter - isso seria bobagem, já que você poderia declarar a propriedade em si como a que for mais público dos dois modificadores.

Esta ajuda ao encapsulamento é extremamente bem-vinda. Ainda não há nada em C# 2 para impedir que outro código da mesma classe ignore a propriedade e vá diretamente para quaisquer que sejam os campos que o apoiam, infelizmente. Como você verá no próximo capítulo, C# 3 corrige isso em um caso específico, mas não em geral.

Passaremos agora de um recurso que você pode querer usar regularmente para um que você deseja evitar na maioria das vezes - isso permite que seu código seja absolutamente explícito em termos dos tipos aos quais se refere, mas com um custo significativo para a legibilidade.

## 7.4 Aliases de namespace

Namespaces destinam-se principalmente a ser um meio de organizar tipos em uma hierarquia útil. Eles também permitem manter nomes totalmente qualificados de tipos distintos, mesmo quando os nomes não qualificados podem ser iguais. Isto não deve ser visto como um convite à reutilização de nomes de tipo não qualificados sem uma boa causa, mas há momentos em que é natural coisa para fazer.

Um exemplo disso é o nome não qualificado Button. Existem duas classes com isso nome no .NET 2.0 Framework: System.Windows.Forms.Button e System.Web.UI.WebControls.Button. Embora ambos sejam chamados de Button, é fácil diferenciá-los por seus namespaces. Isso reflete de perto a vida real – você pode conhecer várias pessoas chamado Jon, mas é improvável que você conheça alguém chamado Jon Skeet. Se você está falando com amigos em um contexto específico, você poderá usar apenas o nome Jon sem especificando de qual você está falando, mas em outros contextos pode ser necessário fornecer informações mais exatas.

A diretiva using do C# 1 (não deve ser confundida com a instrução using que chama Dispose automaticamente) estava disponível em dois tipos: um criava um alias para um namespace ou tipo (por exemplo, usando Out = System.Console;) e o outro introduziu um namespace na lista de contextos que o compilador pesquisaria ao procurar para um tipo (por exemplo, usando System.IO). Em geral, isso era adequado, mas havia

havia algumas situações com as quais a linguagem não conseguia lidar. Em alguns outros casos, o código gerado automaticamente teria que se esforçar para ter certeza absoluta de que os namespaces e tipos corretos estavam sendo usados, independentemente do que acontecesse.

C# 2 corrige esses problemas, trazendo expressividade adicional à linguagem. Agora você pode escrever código que certamente significará o que você deseja, independentemente de quais outros tipos, assemblies e namespaces forem introduzidos. Essas medidas extremas raramente são necessárias fora do código gerado automaticamente, mas é bom saber que elas estão disponíveis quando você precisa delas.

No C# 2, existem três tipos de aliases: os aliases de namespace do C# 1, o alias de namespace *global* e os aliases *externos*. Começaremos com o tipo de alias que já estava presente no C# 1, mas apresentaremos uma nova maneira de usar aliases para garantir que o compilador saiba tratá-lo como um alias em vez de verificar se é o nome de outro namespace ou tipo.

#### 7.4.1 Aliases de namespace qualificados

Mesmo no C# 1, era uma boa ideia evitar aliases de namespace sempre que possível. De vez em quando, você pode descobrir que o nome de um tipo entra em conflito com outro - como no exemplo anterior do Button - então você precisa especificar o nome completo, incluindo o namespace, toda vez que o usa, ou precisa de um alias que diferencie os dois, em algumas maneiras agindo como uma forma abreviada do namespace. A listagem a seguir mostra um exemplo onde os dois tipos de Button são usados, qualificados por um alias.

##### Listagem 7.5 Usando aliases para distinguir entre diferentes tipos de botões

```
usando o sistema;
usando WinForms = System.Windows.Forms; usando WebForms
= System.Web.UI.WebControls;

teste de classe
{
    vazio estático Principal()
    {
        Console.WriteLine(typeof(WinForms.Button));
        Console.WriteLine(typeof(WebForms.Button));
    }
}
```

A Listagem 7.5 compila sem erros ou avisos, embora ainda não seja tão agradável quanto seria se você precisasse lidar apenas com um tipo de Button para começar. Porém, há um problema: e se alguém introduzisse um tipo ou namespace chamado WinForms ou WebForms? O compilador não saberia o que WinForms.Button significava e usaria o tipo ou namespace em vez do alias. Você deseja dizer ao compilador que precisa dele para tratar o WinForms como um alias, mesmo que esteja disponível em outro lugar.

C# 2 introduz a sintaxe *do qualificador de alias ::namespace* para fazer isso, conforme mostrado na listagem a seguir.

**Listagem 7.6 Usando :: para dizer ao compilador para usar aliases**

```
usando o sistema;
usando WinForms = System.Windows.Forms; usando WebForms
= System.Web.UI.WebControls;

classe WinForms {}

teste de classe
{
    vazio estático Principal()
    {
        Console.WriteLine(typeof(WinForms::Button));
        Console.WriteLine(typeof(WebForms::Botão));
    }
}
```

Em vez de WinForms.Button, a listagem 7.6 usa WinForms::Button, e o compilador está satisfeito. Se você alterar o :: de volta para ., você receberá um erro de compilação.

Então, se você usar :: em todos os lugares onde usar um alias, você ficará bem, certo? Bem, não exatamente...

#### 7.4.2 O alias do namespace global

Há uma parte da hierarquia do namespace para a qual você não pode definir seu próprio alias: a raiz dela ou o namespace *global*. Suponha que você tenha duas classes, ambas chamadas Configuration — uma dentro de um namespace de MyCompany e a outra sem nenhum namespace especificado. Como você pode se referir à classe de configuração raiz no namespace MyCompany? Você não pode usar um alias normal e, se apenas especificar Configuration, o compilador usará MyCompany.Configuration.

No C# 1, não havia como contornar isso. Novamente, o C# 2 vem em socorro, permitindo que você use global::Configuration para informar ao compilador exatamente o que você deseja. A listagem a seguir demonstra o problema e a solução.

**Listagem 7.7 Uso do alias de namespace global para especificar exatamente o tipo desejado**

```
usando o sistema;
configuração de classe {}

namespace Capítulo 7
{
    configuração de classe {}

    teste de classe
    {
        vazio estático Principal()
        {
            Console.WriteLine(typeof(Configuração));
            Console.WriteLine(typeof(global::Configuração));
            Console.WriteLine(typeof(global::Chapter7.Test));
        }
    }
}
```

A maior parte da Listagem 7.7 apenas configura a situação – as três linhas dentro de Main são as mais interessantes. A primeira linha imprime Chapter7.Configuration à medida que o compilador resolve Configuration para esse tipo antes de passar para a raiz do namespace. A segunda linha indica que o tipo deve estar no namespace global, então simplesmente imprime Configuration. Inclui a terceira linha para demonstrar que, ao usar o alias global, você ainda pode fazer referência a tipos dentro de namespaces, mas precisa especificar o nome totalmente qualificado.

Neste ponto, você pode acessar qualquer tipo com nome exclusivo, usando o alias do namespace global, se necessário. Se você alguma vez escrever um gerador de código onde o código não precisa ser legível, você pode querer usar esse recurso liberalmente para ter certeza de sempre consultar o tipo correto, independentemente de quaisquer outros tipos que estejam presentes no momento em que o código é compilado. Mas o que você faz se o nome do tipo não for exclusivo mesmo quando você inclui seu namespace? A trama se complica...

### 7.4.3 Aliases externos

No início desta seção, referi-me aos nomes humanos como exemplos de namespaces e contextos. Eu disse especificamente que é improvável que você conheça mais de uma pessoa chamada Jon Skeet. Mas sei que há *mais* de uma pessoa com meu nome, e não é impossível que você conheça dois ou mais de nós. Nesse caso, para especificar a qual deles você se refere, você teria que fornecer mais algumas informações além do nome completo – o motivo pelo qual você conhece a pessoa em particular, ou o país em que ela mora, ou algo similarmente distinto.

O C# 2 permite especificar essas informações extras na forma de um *alias externo* — *um nome que existe não apenas no código-fonte, mas também nos parâmetros que você passa ao compilador*. Para o compilador Microsoft C#, isso significa especificar o assembly que contém os tipos em questão. Suponha que dois assemblies — First.dll e Second.dll — contenham um tipo chamado Demo.Example. Você não pode simplesmente usar o nome totalmente qualificado para distingui-los, pois ambos têm o mesmo nome totalmente qualificado.

Em vez disso, você pode usar aliases externos para especificar o que você quer dizer. A listagem a seguir mostra um exemplo do código C# envolvido, juntamente com a linha de comando necessária para compilá-lo.

#### Listagem 7.8 Trabalhando com diferentes tipos do mesmo tipo em diferentes montagens

```
// Compilar com // csc
Test.cs /r:FirstAlias=First.dll /r:SecondAlias=Second.dll

alias externo FirstAlias; alias externo
SecondAlias;

usando o sistema;
usando FD = FirstAlias::Demo;

teste de classe
{
    vazio estático Principal()
```

 **B** Especifica dois aliases externos

 **C** Refere-se ao alias externo
**C** com alias de namespace

```

{
    Console.WriteLine(typeof(FD.Exemplo));
    Console.WriteLine(typeof(SecondAlias::Demo.Example));
}

```

D Usa alias de namespace  
E alias diretamente

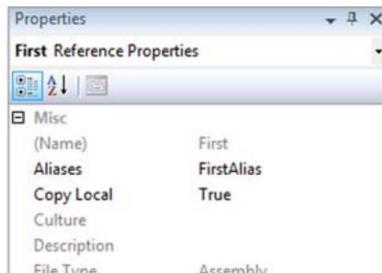
B Usa externo

O código na listagem 7.8 é direto. A primeira coisa que você precisa fazer é apresentar os dois aliases externos B. Depois disso, você pode usá-los por meio de aliases de namespace (C ou D) ou diretamente E. Na verdade, uma diretiva de uso normal sem um alias (como usando FirstAlias::Demo;) teria permitido que você usasse o nome Exemplo sem qualquer qualificação adicional. Um alias externo pode abranger vários assemblies e vários aliases externos podem se referir ao mesmo assembly, embora eu pense com cuidado antes de usar qualquer um desses recursos e principalmente antes de combiná-los.

Para especificar um alias externo no Visual Studio, basta selecionar a referência do assembly em Solution Explorer e modifique o valor Aliases na janela Propriedades, conforme mostrado em figura 7.3.

Espero não precisar persuadi-lo a evite esse tipo de situação sempre que puder. Isto pode ser necessário trabalhar com montagens de diferentes terceiros que tenham usado os mesmos nomes de tipo totalmente qualificados, ponto em que caso contrário você ficaria preso. Onde você tem mais controle sobre a nomenclatura, porém, certifique-se de que seus nomes nunca os levam a este território.

O próximo recurso é quase um meta-recurso. O a funcionalidade exata que ele fornece depende de qual compilador que você está usando, porque seu objetivo é habilitar o controle sobre recursos específicos do compilador. Vamos nos concentrar no compilador da Microsoft.



**Figura 7.3** Parte da janela Propriedades do Visual Studio 2010, mostrando um alias externo de FirstAlias para a Referência de First.dll

## 7.5 Diretrizes do Pragma

Descrever as *diretrivas pragma* em geral é extremamente fácil: uma diretiva pragma é uma diretiva de pré-processamento representada por uma linha que começa com `#pragma`. O resto da linha pode conter qualquer texto. O resultado de uma diretiva pragma não pode mudar o comportamento do programa para violar qualquer coisa dentro da especificação da linguagem C#, mas pode fazer qualquer coisa fora do escopo da especificação. Se o compilador não compreender uma diretiva pragma específica, ele poderá emitir um aviso, mas não um erro.

Isso é basicamente tudo o que a especificação tem a dizer sobre o assunto. O micro O compilador soft C# entende duas diretivas pragma: avisos e somas de verificação.

### 7.5.1 Pragmas de advertência

Ocasionalmente, o compilador C# emite avisos que são justificáveis, mas irritantes. O a resposta correta a um aviso do compilador é *quase sempre* corrigir seu código - raramente é piorou ainda mais ao corrigir a causa do aviso e geralmente melhora.

Mas às vezes há uma boa razão para ignorar um aviso, e é para isso que os pragmas de aviso estão disponíveis. Por exemplo, você criará um campo privado que nunca é lido ou gravado. Quase sempre será inútil... a menos que você saiba que será usado pela reflexão. A listagem a seguir é uma classe completa que demonstra isso.

#### Listagem 7.9 Classe contendo um campo não utilizado

```
classe pública FieldUsedOnlyByReflection {  
    interno x;  
}
```

Se você tentar compilar a listagem 7.9, receberá uma mensagem de aviso como esta:

```
FieldUsedOnlyByReflection.cs(3,9): aviso CS0169: O campo privado  
'FieldUsedOnlyByReflection.x' nunca é usado
```

Essa é a saída do compilador de linha de comando. Na janela Lista de Erros do Visual Studio, você pode ver as mesmas informações (além do projeto em que está), exceto que não recebe o número de aviso (CS0169). Para encontrar o número, você precisa selecionar o aviso e abrir a ajuda relacionada a ele, ou procurar na janela Saída, onde o texto completo é mostrado. Você precisa do número para fazer o código ser compilado sem avisos, conforme mostrado na listagem a seguir.

#### Listagem 7.10 Desativando (e restaurando) o aviso CS0169

```
classe pública FieldUsedOnlyByReflection { #pragma aviso  
  
desabilitar 0169 int x; #pragma aviso  
  
restauração 0169 }
```

A Listagem 7.10 é autoexplicativa — o primeiro pragma desativa o aviso especificado e o segundo o restaura. É uma boa prática desativar os avisos pelo menor tempo possível, para não perder nenhum aviso que realmente devia corrigir. Se você deseja desabilitar ou restaurar vários avisos em uma única linha, basta usar uma lista de números de aviso separados por vírgula. Se você não especificar nenhum número de aviso, o resultado será desabilitar ou restaurar *todos* os avisos de uma só vez, mas isso é uma má ideia em quase todos os cenários imagináveis.

### 7.5.2 Pragmas de soma de verificação

É improvável que você precise da segunda forma de pragma reconhecida pelo compilador da Microsoft. Ele suporta o depurador, permitindo verificar se encontrou o arquivo de origem correto. Normalmente, quando um arquivo C# é compilado, o compilador gera uma soma de verificação do arquivo e a inclui nas informações de depuração. Quando o depurador precisa localizar um arquivo de origem e encontrar várias correspondências potenciais, ele pode gerar a própria soma de verificação para cada um dos arquivos candidatos e ver qual está correto.

Quando uma página ASP.NET é convertida em C#, o arquivo gerado é o que o compilador C# vê. O gerador calcula a soma de verificação da página .aspx e usa um pragma de soma de verificação para informar ao compilador C# para usar essa soma de verificação em vez de calcular uma da página gerada.

A sintaxe do pragma de soma de verificação é

```
#pragma checksum "nome do arquivo" "{guid}" "checksum bytes"
```

O GUID indica qual algoritmo de hash foi usado para calcular a soma de verificação. A documentação da classe `CodeChecksumPragma` fornece GUIDs para SHA-1 e

MD5, caso você deseje implementar sua própria estrutura de compilação dinâmica com suporte ao depurador.

É possível que versões futuras do compilador C# incluam mais pragma diretivas e outros compiladores (como o compilador Mono, mcs) poderiam ter seus suporte próprio para diferentes recursos. Consulte a documentação do seu compilador para o informações mais atualizadas.

O próximo recurso é outro que você talvez nunca use, mas se você usar, é provavelmente tornará sua vida um pouco mais simples.

## 7.6 Buffers de tamanho fixo em código inseguro

Ao chamar código nativo com P/Invoke, não é incomum lidar com com uma estrutura definida para ter um buffer de um comprimento específico dentro dela. Antes de C# 2, essas estruturas eram difíceis de manipular diretamente, mesmo com código inseguro. Agora você pode declarar um buffer do tamanho certo para ser incorporado diretamente com o restante dos dados para a estrutura.

Esse recurso não está disponível apenas para chamar código nativo, embora esse seja seu recurso principal. Você poderia usá-lo para preencher facilmente uma estrutura de dados correspondente diretamente a um arquivo formato, por exemplo. A sintaxe é simples e mais uma vez iremos demonstrá-la com um exemplo. Para criar um campo que incorpore uma matriz de 20 bytes em sua estrutura envolvente, você usaria

dados de bytes fixos[20];

Isso permitiria que os dados fossem usados como se fossem um byte\* (um ponteiro para dados de byte), embora a implementação usada pelo compilador C# seja criar um novo tipo aninhado dentro do tipo de declaração e aplique o novo atributo `FixedBuffer` à variável em si. O CLR então se encarrega de alocar a memória adequadamente.

Uma desvantagem desse recurso é que ele só está disponível em código inseguro: a estrutura envolvente deve ser declarada em um contexto inseguro e você só pode usar o membro do buffer de tamanho fixo em um contexto inseguro. Isso limita as situações em que é usado completo, mas ainda pode ser um bom truque para ter na manga. Além disso, buffers de tamanho fixo são apenas aplicável a tipos primitivos e não podem ser membros de classes (apenas estruturas).

Existem muito poucas APIs do Windows onde esse recurso é diretamente útil.

Inúmeras situações exigem uma matriz fixa de caracteres – o `TIME_ZONE_INFORMATION`

estrutura, por exemplo - mas infelizmente buffers de caracteres de tamanho fixo parecem ser mal tratados pelo P/Invoke, com o empacotador atrapalhando.

A listagem a seguir mostra um exemplo: um aplicativo de console que exibe as cores disponíveis na janela atual do console. Ele usa uma função de API , GetConsoleScreenBufferEx, que foi introduzida no Windows Vista e no Windows Server 2008 e que recupera informações estendidas do console. A listagem a seguir exibe todas as 16 cores em formato hexadecimal (bbggrr).

#### Listagem 7.11 Demonstração de buffers de tamanho fixo para obter informações de cores do console

```

usando o sistema;
usando System.Runtime.InteropServices;

estrutura COORD
{
    short público X, Y;
}

estrutura SMALL_RECT
{
    público curto Esquerda, Superior, Direita, Inferior;
}

estrutura insegura CONSOLE_SCREEN_BUFFER_INFOEX {

    public int StructureSize; public COORD
    ConsoleSize, CursorPosition; Atributos curtos públicos; janela de
    exibição pública SMALL_RECT; public
    COORD MaximumWindowSize; públicos curtos
    PopupAttributes; público intFullScreenSupported;
    público fixo int ColorTable[16];

}

classe estática FixedSizeBufferDemo
{
    const int StdOutputHandle = -11;

    [DllImport("kernel32.dll")]
    static extern IntPtr
    GetStdHandle(int nStdHandle);

    [DllImport("kernel32.dll")]
    estático extern bool GetConsoleScreenBufferInfoEx
        (Identificador IntPtr, informações de referência CONSOLE_SCREEN_BUFFER_INFOEX);

    inseguro static void Main()
    {
        IntPtr Alça = GetStdHandle(StdOutputHandle); Informações
        CONSOLE_SCREEN_BUFFER_INFOEX; informações =
        novo CONSOLE_SCREEN_BUFFER_INFOEX(); info.StructureSize =
        sizeof(CONSOLE_SCREEN_BUFFER_INFOEX); GetConsoleScreenBufferInfoEx(identificador,
        informações de referência);

        for (int i=0; i < 16; i++) {
    }
}

```

```
        Console.WriteLine ("{0:x6}", info.ColorTable[i]);  
    }  
}  
}
```

A Listagem 7.11 utiliza buffers de tamanho fixo para a tabela de cores. Antes dos buffers de tamanho fixo, você poderia ter usado a API com um campo para cada entrada da tabela de cores ou empacotando uma matriz normal como `UnownedType.ByValArray`. Mas isso teria criado um array separado na pilha, em vez de manter todas as informações dentro da estrutura. Isso não é um problema aqui, mas em algumas situações de alto desempenho é bom poder manter grupos de dados juntos. Em uma observação de desempenho diferente, se o buffer fizer parte de uma estrutura de dados no heap gerenciado, será necessário fixá-lo antes de acessá-lo. Se você fizer isso muito, poderá afetar significativamente o coletor de lixo. Estruturas baseadas em pilha não apresentam esse problema, é claro.

Não afirmo que buffers de tamanho fixo sejam um recurso extremamente importante no C# 2 — pelo menos não para a maioria das pessoas. Eu os incluí aqui para serem completos e, sem dúvida, alguém, em algum lugar, os achará inestimáveis.

O recurso final que veremos mal pode ser chamado de recurso da linguagem C# 2, mas é praticamente importante.

7.7 Expondo membros internos a assemblies selecionados Alguns recursos estão obviamente na

linguagem – blocos iteradores, por exemplo. Alguns recursos obviamente pertencem ao tempo de execução, como as otimizações do compilador JIT. Alguns estão claramente em ambos os campos, como os genéricos. Esta última característica tem uma conotação em ambos, mas é tão estranha que não merece menção em nenhuma das especificações. Além disso, utiliza um termo que possui significados diferentes em C++ e VB.NET, acrescentando um terceiro significado à mistura. Para ser justo, todos os termos são usados no contexto de permissões de acesso, mas têm efeitos diferentes.

### 7.7.1 Assemblies de amigos no caso simples

No .NET 1.1 era totalmente correto dizer que algo definido como interno (seja um tipo, um método, uma propriedade, uma variável ou um evento) só poderia ser acessado dentro do mesmo assembly em que foi declarado.<sup>3</sup> Em .NET 2.0, isso ainda é verdade, mas há um novo atributo que permite distorcer um pouco as regras: `InternalsVisibleToAttribute`, geralmente chamado apenas de `InternalsVisibleTo`. (Ao aplicar um atributo cujo nome termina com `Attribute`, o compilador C# aplicará o sufixo automaticamente.)

`InternalsVisibleTo` só pode ser aplicado a um assembly (não a um membro específico) e você pode aplicá-lo várias vezes ao mesmo assembly. Chamarei o assembly que contém o atributo de assembly fonte, embora esta seja uma terminologia não oficial. Ao aplicar o atributo, você deve especificar outro assembly, conhecido como assembly amigo. O resultado é que a montagem amiga pode ver todos os membros internos da

<sup>3</sup> Usar reflexão ao executar com permissões adequadas não conta.

montagem de origem como se fossem públicos. Isto pode parecer alarmante, mas pode ser útil, como você verá em um minuto.

A listagem a seguir mostra isso com três classes em três assemblies diferentes.

#### Listagem 7.12 Demonstração de montagens amigas

```
// Compilado para Source.dll usando
System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("FriendAssembly")]
classe pública Fonte {
    void estático interno InternalMethod() {}

    public static void PublicMethod() {}
}

// Compilado para FriendAssembly.dll public class
Friend {
    static void Principal() {
        Source.InternalMethod();
        Fonte.PublicMethod();
    }
}

// Compilado para EnemyAssembly.dll public class
Enemy {
    static void Principal() {
        // Source.InternalMethod(); Fonte.PublicMethod();
    }
}
```

**Concessões adicionais acesso**

**Usa acesso adicional dentro AmigoAssembléia**

**B** **EnemyAssembly não tem acesso especial**

**Acesse o método público normalmente**

Na listagem 7.12, existe um relacionamento especial entre FriendAssembly.dll e Source.dll, embora ele opere apenas de uma maneira: Source.dll não tem acesso a membros internos de FriendAssembly.dll. Se você descomentar a linha em B, a classe Enemy não será compilada.

Para começar, por que diabos você iria querer abrir sua montagem bem projetada para certas montagens?

### 7.7.2 Por que usar `InternalsVisibleTo`?

Raramente uso `InternalsVisibleTo` entre duas montagens de produção. Posso ver como isso pode ser útil e certamente o usei para acesso extra ao escrever ferramentas, mas meu uso principal sempre foi em testes unitários.

Alguns dizem que você só deve testar a interface pública do código. Pessoalmente, fico feliz em testar tudo o que posso da maneira mais simples possível. Os assemblies amigos tornam isso muito mais fácil: de repente, é trivial testar código que só tem acesso interno sem tomar a atitude duvidosa de tornar os membros públicos apenas para testar ou incluir o

código de teste dentro do assembly de produção. (Ocasionalmente, significa tornar os membros internos para fins de teste onde, de outra forma, poderiam ser privados, mas isso é menos preocupante.)

A única desvantagem disso é que o nome do seu assembly de teste permanece no seu assembly de produção. Em teoria, isso poderia representar um vetor de ataque à segurança se seus assemblies não estiverem assinados e seu código normalmente operar sob um conjunto restrito de permissões. (Qualquer pessoa com total confiança poderia usar a reflexão para acessar os membros em primeiro lugar. Você poderia fazer isso sozinho para testes de unidade, mas é muito mais desagradável.) Se isso acabar sendo um problema genuíno para alguém, ficarei muito surpreso. . Mas traz a opção de assinar montagens em cena. Justamente quando você pensou que este era um pequeno recurso simples e legal...

### 7.7.3 InternalsVisibleTo e assemblies assinados

Se um assembly amigo for assinado, o assembly de origem precisará especificar a chave pública do assembly amigo, para ter certeza de que está confiando no código correto. Você precisa da chave pública completa, não apenas do token da chave pública.

Por exemplo, considere a seguinte linha de comando e saída (reembalada e ligeiramente modificada para formatação) usada para descobrir a chave pública de um Friend Assembly.dll assinado:

```
c:\Users\Jon\Test>sn -Tp FriendAssembly.dll Utilitário de nome forte do
Microsoft (R) .NET Framework versão 3.5.21022.8 Copyright (c) Microsoft Corporation. Todos os direitos reservados.
```

A chave pública é  
0024000004800000940000000602000000240000525341310004000001  
000100a51372c81ccfb8fba9c5fb84180c4129e50f0facdce932cf31fe  
563d0fe3cb6b1d5129e28326060a3a539f287aaaf59affc5aabc4d8f981  
e1a82479ab795f410eab22e3266033c633400463ee7513378bb4ef41fc  
0cae5fb03986d133677c82a865b278c48d99dc251201b9c43edd7bedef  
d4b5306efd0dec7787ec6b664471c2

O token de chave pública é 647b99330b7f792c

O código-fonte da classe Source agora precisaria ter isto como atributo:

```
[montagem:InternalsVisibleTo("FriendAssembly,PublicKey=" +
"0024000004800000940000000602000000240000525341310004000001" +
"000100a51372c81ccfb8fba9c5fb84180c4129e50f0facdce932cf31fe" +
"563d0fe3cb6b1d5129e28326060a3a539f287aaaf59affc5aabc4d8f981" +
"e1a82479ab795f410eab22e3266033c633400463ee7513378bb4ef41fc" +
"0cae5fb03986d133677c82a865b278c48d99dc251201b9c43edd7bedef" +
"d4b5306efd0dec7787ec6b664471c2")]
```

Infelizmente, você precisa ter a chave pública em uma linha ou usar a concatenação de strings – espaços em branco na chave pública causarão uma falha na compilação. Seria muito mais agradável se você pudesse especificar o token em vez da chave inteira, mas felizmente essa feiúra geralmente fica confinada ao AssemblyInfo.cs, então você não precisarávê-lo com frequência.

Em teoria, é possível ter um assembly de origem não assinado e um assembly amigo assinado. Na prática, isso não é muito útil, já que o assembly amigo normalmente deseja ter uma referência ao assembly de origem, e você não pode fazer referência a um assembly não assinado a partir de um que esteja assinado. Da mesma forma, um assembly assinado não pode especificar um assembly amigo não assinado, então normalmente você acaba com ambos os assemblies sendo assinados se um deles estiver.

## 7.8 Resumo

Isso conclui nosso tour pelos novos recursos do C# 2. Os tópicos que examinamos neste capítulo se enquadram amplamente em duas categorias: melhorias “bom ter” que agilizam o desenvolvimento e recursos “espero que você não precise deles”. Isso pode tirar você de situações complicadas quando você precisar delas. Para fazer uma analogia entre C# 2 e melhorias em uma casa, os principais recursos dos capítulos anteriores são comparáveis a acréscimos em escala real. Alguns dos recursos que vimos neste capítulo (como tipos parciais e classes estáticas) são mais parecidos com a redecoração de um quarto, e recursos como apelidos de namespace são semelhantes a alarmes de fumaça - talvez você nunca veja um benefício, mas é bom saber que eles estão lá se você precisar deles.

A gama de recursos do C# 2 é ampla: os designers abordaram muitas das áreas onde os desenvolvedores estavam sentindo dificuldades, sem nenhum objetivo abrangente. Isso não quer dizer que os recursos não funcionem bem juntos – tipos de valor anuláveis não seriam viáveis sem genéricos, por exemplo – mas não há um objetivo único para o qual cada recurso contribua, a menos que você conte a produtividade geral.

Agora que terminamos de examinar o C# 2, é hora de passar para o C# 3, onde a imagem é muito diferente. Quase todos os recursos do C# 3 fazem parte do quadro geral do LINQ, um conglomerado de tecnologias que simplifica enormemente muitas tarefas.

## Parte 3

# C# 3: Revolucionando o acesso a dados

Não há dúvida de que o C# 2 é uma melhoria significativa em relação ao C# 1. Os benefícios Os benefícios dos genéricos, em particular, são fundamentais para outras mudanças, não apenas no C# 2, mas também no C# 3. Mas o C# 2 é, em certo sentido, uma coleção fragmentada de recursos. Não me interpretem mal: eles se encaixam perfeitamente, mas abordam um conjunto de questões individuais. Isso era apropriado naquele estágio de desenvolvimento do C#, mas o C# 3 é diferente.

Quase todos os recursos do C# 3 permitem uma tecnologia específica: LINQ. Muitos dos recursos são úteis fora deste contexto e você certamente não deve se limitar a usá-los apenas quando estiver escrevendo uma expressão de consulta, mas seria igualmente tolo não reconhecer a imagem completa criada pelo conjunto de peças de quebra-cabeça apresentadas nos cinco capítulos seguintes.

Quando escrevi originalmente sobre C# 3 e LINQ em 2007, fiquei muito impressionado em um nível um tanto acadêmico. Quanto mais profundamente você estuda a língua, mais claramente você vê a harmonia entre os vários elementos que foram introduzidos. A elegância das expressões de consulta — e em particular a capacidade de usar a mesma sintaxe para consultas em processo e provedores como LINQ to SQL — era muito atraente. O LINQ era muito promissor.

Agora, anos depois, posso relembrar as promessas e ver como elas se cumpriram. Na minha experiência com a comunidade — especialmente no Stack Overflow — é óbvio que o LINQ foi amplamente adotado e realmente mudou a forma como abordamos muitas tarefas orientadas a dados. Os provedores de banco de dados não estão restritos aos da Microsoft - LINQ to NHibernate e SubSonic são apenas dois dos outros

opções disponíveis. A Microsoft também não parou de inovar no LINQ; no capítulo 12 você verá Parallel LINQ e Reactive Extensions, duas maneiras muito diferentes de lidar com dados que ainda usam os familiares operadores LINQ. E há também o LINQ to Objects — o provedor LINQ mais simples, mais previsível e quase mundano, e o mais difundido no setor. Os dias de escrever mais um loop de filtragem, mais um pedaço de código para encontrar algum valor máximo, mais uma verificação para ver se algum item em uma coleção satisfaz alguma condição acabaram - e já foi.

Apesar da ampla adoção do LINQ, ainda vejo diversas questões que deixam claro que alguns desenvolvedores consideram o LINQ uma espécie de caixa preta mágica. O que acontecerá quando eu usar uma expressão de consulta, em comparação com o uso direto de métodos de extensão? Quando os dados são realmente lidos? Como posso fazê-lo funcionar com mais eficiência? Embora você possa aprender muito sobre LINQ apenas brincando com ele e olhando exemplos em postagens de blog, você aproveitará muito mais vendo como tudo funciona no nível da linguagem e depois aprendendo sobre o que as diversas bibliotecas funcionam. fazer por você.

Este não é um livro sobre LINQ — ainda estou me concentrando nos recursos da linguagem que habilitam o LINQ, em vez de entrar em detalhes sobre considerações de simultaneidade para o Entity Framework e assim por diante. Mas depois de ver os elementos da linguagem individualmente e como eles se encaixam, você estará em uma posição muito melhor para aprender os detalhes de provedores específicos.

# Cortando coisas com um compilador inteligente

## Este capítulo cobre

- ÿ Propriedades implementadas automaticamente
- ÿ Variáveis locais digitadas implicitamente
- ÿ Inicializadores de objetos e coleções
- ÿ Matrizes digitadas implicitamente
- ÿ Tipos anônimos

Começaremos a analisar o C# 3 da mesma forma que terminamos de examinar o C# 2 — com uma coleção de recursos relativamente simples. Estes são apenas os primeiros pequenos passos no caminho para o LINQ. Cada um deles pode ser usado fora desse contexto, mas quase todos são importantes para simplificar o código na medida que o LINQ exige para ser eficaz.

Um ponto importante a ser observado é que, embora dois dos maiores recursos do C# 2 — genéricos e tipos anuláveis — exigissem alterações no CLR, não houve alterações significativas no CLR fornecido com o .NET 3.5. Houve alguns ajustes, mas nada fundamental. A estrutura cresceu para suportar LINQ, e mais alguns recursos foram introduzidos na biblioteca de classes base, mas isso é um assunto diferente. Isso é

vale a pena ter claro quais alterações são apenas na *linguagem C#*, quais são alterações de *biblioteca* e quais são alterações *CLR*.

Quase todos os novos recursos expostos no C# 3 se devem ao fato de o compilador estar disposto a fazer mais trabalho para você. Você viu algumas evidências disso em parte do livro — particularmente com métodos anônimos e blocos iteradores — e o C# 3 continua na mesma linha. Neste capítulo, você conhecerá os seguintes recursos que são novos no C# 3:

- ÿ *Propriedades implementadas automaticamente*—Elimine o trabalho enfadonho de escrever propriedades simples apoiadas diretamente por campos
- ÿ *Variáveis locais digitadas implicitamente*—Reduza a redundância da declaração de variáveis locais inferindo o tipo de variável a partir do valor inicial
- ÿ *Inicializadores de objetos e coleções*—Simplifique a criação e inicialização de objetos em expressões únicas
- ÿ *Matrizes digitadas implicitamente*—Reduza a redundância de expressões de criação de matrizes inferindo o tipo de array a partir do conteúdo
- ÿ *Tipos anônimos*—Permite a criação de tipos ad hoc para conter propriedades

Além de descrever o que os novos recursos fazem, farei recomendações sobre seu uso. Muitos dos recursos do C# 3 exigem certa descrição e restrição por parte do desenvolvedor. Isso não quer dizer que eles não sejam poderosos e incrivelmente úteis - muito pelo contrário - mas a tentação de usar a sintaxe mais recente e moderna não deve anular o impulso em direção a um código claro e legível.

As considerações que discutirei neste capítulo (e no restante do livro) raramente serão preto e branco. Talvez mais do que nunca, a legibilidade está nos olhos de quem vê e, à medida que você se sente mais confortável com os novos recursos, é provável que eles se tornem mais legíveis para você. Devo enfatizar, porém, que, a menos que você tenha bons motivos para supor que será o único a ler seu código, você deve considerar cuidadosamente as necessidades e opiniões de seus colegas.

Isso é o suficiente para olhar para o umbigo por enquanto. Começaremos com um recurso que não deve causar polêmica. Propriedades simples, mas eficazes e implementadas automaticamente apenas tornam a vida melhor.

## 8.1 Propriedades implementadas automaticamente

O primeiro recurso que discutiremos é provavelmente o mais simples de todo o C# 3. É ainda mais simples do que qualquer um dos novos recursos do C# 2. Apesar disso — ou possivelmente por causa disso — ele também é imediatamente aplicável em muitas, muitas situações. Ao ler sobre blocos iteradores no capítulo 6, você pode não ter pensado imediatamente em quaisquer áreas de sua base de código atual que poderiam ser melhoradas usando-os, mas eu ficaria surpreso em encontrar qualquer programa C# 2 não trivial que não pudesse ser modificado para usar propriedades implementadas automaticamente. Esse recurso fabulosamente simples permite expressar propriedades triviais com menos código do que antes.

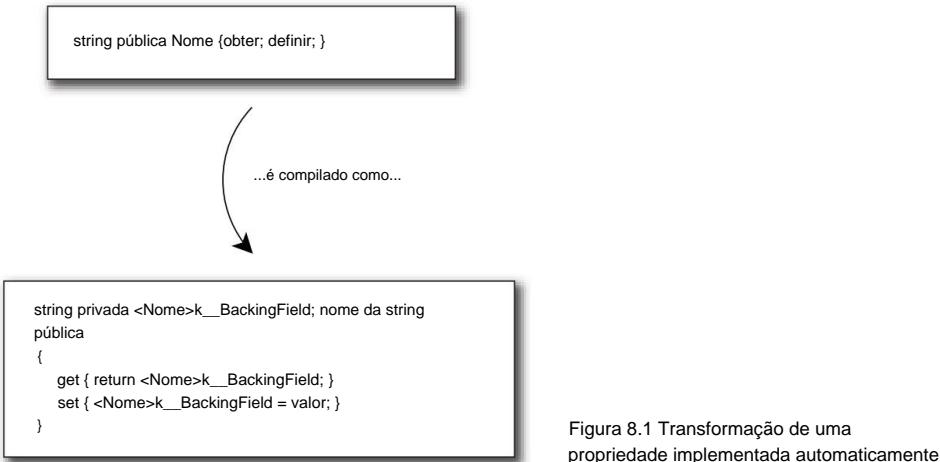


Figura 8.1 Transformação de uma propriedade implementada automaticamente

O que quero dizer com propriedade trivial? Quero dizer, aquele que lê/escreve e que armazena seus valor em uma variável privada simples, sem qualquer validação ou outro costume código. Propriedades triviais ocupam apenas algumas linhas de código, mas isso ainda é muito quando você considera que está expressando um conceito muito simples. C# 3 reduz a verbosidade em aplicando uma transformação simples em tempo de compilação, conforme mostrado na figura 8.1.

O código na parte inferior da figura 8.1 não é C# totalmente válido, é claro. O campo tem um nome indizível para evitar colisões de nomes, da mesma forma que você viu antes para métodos anônimos e blocos iteradores. Mas esse é efetivamente o código gerado pela propriedade implementada automaticamente no topo.

Onde anteriormente você poderia ter ficado tentado a usar uma variável pública por causa de simplicidade, agora há ainda menos desculpas para não usar uma propriedade. Isto é particularmente verdadeiro para código descartável, que todos sabemos que tende a durar muito mais tempo do que previsto.

#### TERMINOLOGIA: PROPRIEDADE AUTOMÁTICA OU IMPLEMENTADA AUTOMATICAMENTE

**PROPRIEDADE?** Quando as propriedades implementadas automaticamente foram discutidas pela primeira vez, muito antes da especificação completa do C# 3 ser publicada, elas foram chamadas propriedades automáticas. Pessoalmente, acho isso menos complicado do que o nome completo e é mais amplamente usado na comunidade. Não há risco de ambiguidade, portanto, no restante deste livro, usarei propriedade automática e propriedade implementada automaticamente como sinônimo.

O recurso do C# 2 que permite especificar acessos diferentes para o getter e o setter ainda está disponível aqui e você também pode criar propriedades automáticas estáticas. Mas propriedades automáticas estáticas são quase sempre inúteis. Embora a maioria dos tipos não afirmam ter membros de instância thread-safe, membros estáticos visíveis publicamente geralmente deve ser thread-safe, e o compilador não faz nada para ajudá-lo nisso respeito. A listagem a seguir dá um exemplo de um sistema automático estático seguro, mas inútil.

propriedade que conta quantas instâncias de uma classe foram criadas, juntamente com propriedades de instância para o nome e a idade de uma pessoa.

#### Listagem 8.1 Contando instâncias de maneira estranha com uma propriedade automática estática

```
classe pública Pessoa {

    string pública Nome {obter; conjunto privado; } public int Idade {obter;
    conjunto privado; }

    private static int InstanceCounter {obter; definir; } objeto private static readonly
    counterLock = new object();

    public InstanceCountingPerson(string nome, int idade) {

        Nome = nome;
        Idade = idade;

        bloquear (counterLock) {
            InstanceCounter++;
        }
    }
}
```

Nesta listagem, você usa um cadeado para garantir que não terá problemas de threading e também precisará usar o mesmo cadeado sempre que acessar a propriedade. Existem alternativas melhores aqui envolvendo a classe Interlocked , mas exigem acesso aos campos. Resumindo, o único cenário em que vejo propriedades automáticas estáticas sendo úteis é onde o getter é público, o setter é privado e o setter é chamado apenas dentro do inicializador de tipo.

As outras propriedades na listagem 8.1, que representam o nome e a idade da pessoa, contam uma história muito mais feliz – usar propriedades automáticas é algo óbvio aqui. Onde você tem propriedades que você teria implementado trivialmente em versões anteriores do C#, não há benefício em *não* usar propriedades automáticas.<sup>1</sup>

Um pequeno problema ocorre se você usar propriedades automáticas ao escrever suas próprias estruturas: todos os seus construtores precisam chamar explicitamente o construtor sem parâmetros — this() — para que o compilador saiba que todos os campos foram definitivamente atribuídos. Você não pode definir os campos diretamente porque eles são anônimos e não pode usar as propriedades até que todos os campos tenham sido definidos. A única maneira de proceder é chamar o construtor sem parâmetros, que definirá os campos com seus valores padrão. Por exemplo, se você quisesse criar uma estrutura com uma única propriedade inteira, isso não seria válido:

```
estrutura pública Foo {

    público int Valor {obter; conjunto privado; }
```

<sup>1</sup> Certamente para propriedades de leitura/gravação, de qualquer maneira. Se estiver criando uma propriedade somente leitura, você poderá optar por usar um campo auxiliar somente leitura e uma propriedade com apenas um getter para retorná-lo. Isso evita que você grave acidentalmente na propriedade dentro da classe, o que seria possível com uma propriedade automática “leitura pública, gravação privada”.

```
público Foo(int valor) {
    este.Valor = valor;
}
```

Mas não há problema se você encadear explicitamente o construtor sem parâmetros:

```
estrutura pública Foo {
    público int Valor {obter; conjunto privado; }
    public Foo(int valor): this() {
        este.Valor = valor;
    }
}
```

Isso é tudo que existe para propriedades implementadas automaticamente. Não há sinos e assobios para eles. Por exemplo, não há como declará-los com valores padrão iniciais e nenhuma maneira de torná-los genuinamente somente leitura (um setter privado é o mais próximo possível).

Se todos os recursos do C# 3 fossem tão simples, poderíamos cobrir *tudo* em um único capítulo. Claro, esse não é o caso, mas existem alguns recursos que não exigem *muitas* explicações. O próximo tópico remove código duplicado em outra situação comum, mas específica: declaração de variáveis locais.

## 8.2 Tipagem implícita de variáveis locais

No capítulo 2 discutimos a natureza do sistema de tipos C# 1. Em particular, afirmei que era estático, explícito e seguro. Isso ainda é verdade em C# 2 e em C# 3 ainda é quase completamente verdade. As partes estáticas e seguras ainda são verdadeiras (ignorando o código explicitamente inseguro, assim como fizemos no capítulo 2) e, na *maioria* das vezes, ele ainda é digitado explicitamente — mas você pode pedir ao compilador para inferir os tipos de variáveis locais para você.<sup>2</sup>

### 8.2.1 Usando var para declarar uma variável local

Para usar a digitação implícita, tudo o que você precisa fazer é substituir a parte do tipo de uma declaração normal de variável local por var. Existem certas restrições (falaremos delas em um momento), mas essencialmente é tão fácil quanto mudar isso:

```
MeuTipo nomevariável = algumValorInicial;
```

para isso:

```
var nomevariável = algumValorInicial;
```

Os resultados das duas linhas (em termos de código compilado) são *exatamente os mesmos*, assumindo que o tipo de someInitialValue seja MyType. O compilador simplesmente pega o tipo de tempo de compilação da expressão de inicialização e faz com que a variável também tenha esse tipo.

<sup>2</sup> O C# 4 muda o jogo mais uma vez, permitindo que você use a digitação dinâmica onde quiser, como verá no capítulo 14. Um passo de cada vez — o C# ainda era totalmente digitado estaticamente até a versão 3, inclusive.

```

int totalAge = 0;
foreach (var person in family)
{
    totalAge += person.Age;
}
(local variable) 'a person
Anonymous Types:
'a is new { string Name, int Age }

```

**Figura 8.2** Passar o mouse sobre var no Visual Studio exibe o tipo da variável declarada.

O tipo pode ser qualquer tipo .NET normal , incluindo genéricos, delegados e interfaces.

A variável ainda é digitada estaticamente; você simplesmente não escreveu o nome do tipo em seu código.

É importante entender isso, pois vai ao cerne do que muitos desenvolvedores temem inicialmente quando veem esse recurso: que var torna o C# dinâmico ou de tipo fraco.

Isso definitivamente não é verdade. A melhor maneira de explicar isso é mostrar algum código inválido:

 INVÁLIDO  
var stringVariable = "Olá, mundo."; stringVariável = 0;

Isso não compila porque o tipo de stringVariable é System.String e você não pode atribuir o valor 0 a uma variável de string. Em muitas linguagens dinâmicas, o código seria compilado, deixando a variável sem nenhum tipo particularmente útil no que diz respeito ao compilador, IDE ou ambiente de tempo de execução. Usar var não é como usar um tipo VARIANT do COM ou VB6. A variável é digitada estaticamente; o tipo acaba de ser inferido pelo compilador. Peço desculpas se pareço estar me esforçando um pouco neste ponto, mas é extremamente importante e tem sido causa de muita confusão.

No Visual Studio, você pode saber qual tipo o compilador usou para a variável passando o mouse sobre a parte var da declaração, conforme mostrado na figura 8.2. Observe como os parâmetros de tipo para o tipo Dicionário genérico também são explicados. Se isso parece familiar, é porque é exatamente o mesmo comportamento que você obtém quando declara variáveis locais explicitamente.

As dicas de ferramentas também não estão disponíveis apenas no momento da declaração. Como você provavelmente esperaria, a dica exibida quando você passa o mouse sobre o nome da variável posteriormente no código também indica o tipo da variável. Isso é mostrado na figura 8.3, onde a mesma declaração é usada e então passei o mouse sobre o uso da variável. Novamente, esse é exatamente o mesmo comportamento que você veria com uma declaração normal de variável local.

```

var namePeopleMap = new Dictionary<string, List<Person>>();
// Other code
Console.WriteLine(namePeopleMap.Count);
(local variable) Dictionary<string, List<Person>> namePeopleMap

```

**Figura 8.3** Passar o mouse sobre o uso de uma variável local digitada implicitamente exibe seu tipo.

Há dois motivos para trazer o Visual Studio nesse contexto. O primeiro é que é mais uma evidência da tipagem estática envolvida - o compilador conhece claramente o tipo da variável. A segunda é ressaltar que você pode descobrir facilmente o tipo envolvidos, mesmo do fundo de um método. Isso será importante quando falarmos sobre os prós e contras do uso da digitação implícita em um minuto. Antes, porém, devo mencionar algumas limitações.

### 8.2.2 Restrições à digitação implícita

Você não pode usar digitação implícita para todas as variáveis em todos os casos. Você só pode usá-lo quando todos os seguintes pontos são verdadeiros:

- ÿ A variável que está sendo declarada é uma variável local, em vez de uma variável estática ou de instância campo.
- ÿ A variável é inicializada como parte da declaração.
- ÿ A expressão de inicialização não é um grupo de métodos ou função anônima<sup>3</sup> (sem fundição).
- ÿ A expressão de inicialização não é nula.
- ÿ Apenas uma variável é declarada na instrução.
- ÿ O tipo que você deseja que a variável tenha é o tipo de tempo de compilação do arquivo inicializa expressão de ção.
- ÿ A expressão de inicialização não envolve a declaração da variável.<sup>4</sup>

O terceiro e quarto pontos são interessantes. Você não pode escrever isto:



INVÁLIDO

```
var starter = delegado() { Console.WriteLine(); };
```

Isso ocorre porque o compilador não sabe que tipo usar. Você *pode* escrever isto:

```
var starter = (ThreadStart) delegado() { Console.WriteLine(); };
```

Mas se você for fazer isso, seria melhor declarar explicitamente a variável em o primeiro lugar. O mesmo é verdade no caso nulo – você poderia lançar o nulo apropriadamente, mas não faria sentido.

Observe que você *pode* usar o resultado de chamadas de método ou propriedades como método de inicialização expressão – você não está limitado a constantes e chamadas de construtor. Por exemplo, você poderia usar isso:

```
var args = Environment.GetCommandLineArgs();
```

Nesse caso, args seria do tipo `string[]`. Na verdade, inicializar uma variável com o O resultado de uma chamada de método é provavelmente a situação mais comum em que a digitação implícita é usado como parte do LINQ. Você verá tudo isso mais tarde — basta ter isso em mente à medida que os exemplos avançam.

<sup>3</sup> O termo *função anônima* abrange métodos anônimos e expressões lambda, nos quais nos aprofundaremos no capítulo 9.

<sup>4</sup> Seria muito incomum fazer isso de qualquer maneira, mas com declarações normais é possível se você se esforçar o suficiente.

Também vale a pena notar que você *pode* usar digitação implícita para as variáveis locais declaradas na primeira parte de uma instrução using, for ou foreach . Por exemplo, os itens a seguir são todos válidos (com os órgãos apropriados, é claro):

```
for (var i = 0; i < 10; i++) usando (var x =
File.OpenText("test.dat")) foreach (var s em
Environment.GetCommandLineArgs())
```

Essas variáveis acabariam com os tipos int, StreamReader e string, respectivamente.

Claro, só porque você *pode* fazer isso não significa que *deva*. Vejamos as razões a favor e contra o uso da digitação implícita.

### 8.2.3 Prós e contras da digitação implícita

A questão de quando é uma boa ideia usar a digitação implícita é a causa de muita discussão na comunidade. As visualizações variam de “em todos os lugares” a “nenhum lugar”, com muitas abordagens mais equilibradas entre os dois. Você verá na seção 8.5 que, para usar outro recurso do C# 3 — tipos anônimos — você geralmente *precisa* usar digitação implícita. Você também poderia evitar tipos anônimos, é claro, mas isso seria jogar o bebê fora junto com a água do banho.

A principal razão *para* usar a digitação implícita (deixando os tipos anônimos de lado por enquanto) não é reduzir o número de pressionamentos de tecla necessários para inserir o código, mas tornar o código menos confuso (e, portanto, mais legível) na tela.

Em particular, quando estão envolvidos genéricos, os nomes dos tipos podem ficar muito longos. As Figuras 8.2 e 8.3 usaram um tipo de Dictionary<string, List<Person>>, que tem 33 caracteres. No momento em que você tem isso duas vezes em uma linha (uma para a declaração e outra para a inicialização), você acaba com uma linha enorme apenas para declarar e inicializar uma única variável. Uma alternativa é usar um alias, mas isso coloca o tipo real envolvido muito longe (pelo menos conceitualmente) do código que o utiliza.

Ao ler o código, não faz sentido ver o mesmo nome de tipo longo duas vezes na mesma linha quando é óbvio que eles *deveriam* ser iguais. Se a declaração não estiver visível na tela, você estará no mesmo barco, independentemente de a digitação implícita ter sido usada ou não (todas as formas que você usaria para descobrir o tipo de variável ainda são válidas) e, se estiver *visível*, a expressão usada para inicializar a variável informa o tipo de qualquer maneira.

Além disso, usar var altera a ênfase do código. Às vezes você deseja que o leitor preste muita atenção aos tipos exatos envolvidos porque eles são significativos.

Por exemplo, embora os tipos genéricos SortedList e SortedDictionary tenham APIs semelhantes, eles têm características de desempenho diferentes e isso pode ser importante para seu trecho de código específico. Outras vezes, tudo o que realmente importa são as operações que estão sendo executadas; você realmente não se importaria se a expressão usada para inicializar a variável mudasse, contanto que você pudesse atingir os mesmos objetivos.<sup>5</sup> Usando

---

<sup>5</sup> Sei que isso soa um pouco como a digitação de um pato: “Contanto que ele possa grunhar, estou feliz”. A diferença é que você ainda está verificando a charlatanice em tempo de compilação, não em tempo de execução.

var permite que o leitor se concentre no *uso* de uma variável em vez da declaração - o *o quê* em vez do *como* do código.

Tudo isso parece bom, então quais são os argumentos *contra* a digitação implícita? Para doxicamente, a legibilidade é o mais importante, apesar de também ser um argumento a favor da digitação implícita! Por não ser explícito sobre que tipo de variável

você está declarando, pode estar dificultando o trabalho ao ler o código.

Isso quebra a mentalidade de "declare o que você está declarando e depois com que valor começará" que mantém a declaração e a inicialização separadas. Até que ponto isso é um O problema depende do leitor e da expressão de inicialização envolvida.

Se você estiver chamando explicitamente um construtor, sempre será bastante óbvio que tipo você está criando. Se você estiver chamando um método ou usando uma propriedade, isso depende de como óbvio que o tipo de retorno é quando se olha para a chamada. Literais inteiros são um exemplo onde é mais difícil adivinhar o tipo de expressão do que você imagina. Como rapidamente você consegue descobrir o tipo de cada uma das variáveis declaradas aqui?

```
var a = 2147483647;
var b = 2147483648;
varc = 4294967295;
var d = 4294967296;
var e = 9223372036854775807;
var f = 9223372036854775808;
```

As respostas são int, uint, uint, long, long e ulong, respectivamente – o tipo usado depende do valor da expressão. Não há nada de novo aqui em termos de manipulação de literais — C# sempre se comportou assim — mas a digitação implícita torna mais fácil escrever código obscuro nesse caso.

O argumento que raramente é explicitamente declarado, mas que acredito estar por trás de grande parte do A preocupação com a digitação implícita é: "Simplesmente não parece certo". Se você escreve em uma linguagem semelhante a C há anos e anos, há algo enervante em todo o negócio, por mais que você diga a si mesmo que ainda é uma digitação estática nos bastidores. Esse pode não ser uma preocupação racional, mas isso não a torna menos real. Se você se sentir desconfortável, provavelmente será menos produtivo. Se as vantagens não superarem seus sentimentos negativos, tudo bem. Dependendo da sua personalidade, você pode tentar se esforçar para se sentir mais confortável com a digitação implícita, mas certamente não é necessário.

#### 8.2.4 Recomendações

Aqui estão algumas recomendações baseadas na minha experiência com digitação implícita. Isso é tudo o que são - recomendações - e você deve se sentir à vontade para aceitá-las com atenção de sal:

- ÿ Se for importante que alguém que esteja lendo o código saiba o tipo da variável de relance, use digitação explícita.
- ÿ Se a variável for inicializada diretamente com um construtor e o nome do tipo for longo (o que geralmente ocorre com genéricos), considere usar tipagem implícita.

ÿ Se o tipo preciso da variável não for importante, mas sua natureza geral for clara

do contexto, use digitação implícita para tirar a ênfase de *como* o código atinge seu  
mire e concentre-se no nível mais alto do que está alcançando.

ÿ Consulte seus colegas de equipe sobre o assunto ao embarcar em um novo projeto.

ÿ Em caso de dúvida, tente uma linha nos dois sentidos e siga seus instintos.

Eu costumava usar digitação explícita para código de produção, exceto em situações onde havia um benefício claro e significativo no uso de digitação implícita. A maioria dos meus usos de digitação implícita foi em código de teste (e código descartável). Hoje em dia sou mais ambivalente e francamente inconsistente. Ficarei feliz em usar a digitação implícita no código de produção apenas por um pouco maior simplicidade, mesmo quando os nomes dos tipos envolvidos não são muito onerosos. Embora consistência em alguns aspectos do estilo de codificação é muito importante, não encontrei isso abordagem combinada para causar quaisquer problemas.

Efetivamente, minha recomendação se resume a *não* usar digitação implícita apenas porque economiza algumas teclas. Onde ele mantém o código mais organizado, permitindo que você se concentre nos elementos mais importantes do código, vá em frente. Usarei digitação implícita extensivamente no resto do livro, pela simples razão de que o código é mais difícil de formatar na impressão do que na tela – não há tanta largura disponível.

Voltaremos à digitação implícita quando examinarmos os tipos anônimos, pois eles criam situações em que você é forçado a pedir ao compilador para inferir os tipos de algumas variáveis. Antes disso, vamos ver como o C# 3 facilita a construção e o preenchimento de um novo objeto em uma expressão.

## 8.3 Inicialização simplificada

Alguém poderia pensar que as linguagens orientadas a objetos teriam simplificado o processamento de objetos. criação há muito tempo. Afinal, antes de começar a usar um objeto, *algo* precisa criá-lo, seja através do seu código diretamente ou de algum tipo de método de fábrica. Apesar disso, poucos recursos da linguagem C# 2 são voltados para facilitar a vida quando se trata de inicialização. Se você não pode fazer o que deseja usando argumentos de construtor, você está basicamente sem sorte – você precisa criar o objeto e então inicializá-lo manualmente com chamadas de propriedade e similares.

Isto é particularmente irritante quando você deseja criar um monte de objetos em de uma só vez, como em uma matriz ou outra coleção. Sem uma forma de expressão única para inicializar um objeto, você é forçado a usar variáveis locais para manipulação temporária ou a criar um método auxiliar que execute a inicialização apropriada com base em parâmetros.

O C# 3 vem em socorro de diversas maneiras, como você verá nesta seção.

### 8.3.1 Definindo alguns tipos de amostra

As expressões que usaremos nesta seção são chamadas de *inicializadores de objetos*. Estas são apenas maneiras de especificar a inicialização que deve ocorrer após a criação de um objeto. Você pode definir propriedades, definir propriedades de propriedades (não se preocupe, é mais simples do que parece), e adicione a coleções acessíveis por meio de propriedades.

Para demonstrar tudo isso, usaremos novamente a classe Person . Ele tem o nome e a idade que usamos antes, expostos como propriedades graváveis. Forneceremos um construtor sem parâmetros e um que aceita o nome como parâmetro. Também adicionaremos uma lista de amigos e o local de residência da pessoa, ambos acessíveis como propriedades somente leitura, mas que ainda podem ser modificados pela manipulação dos objetos recuperados. Uma classe Location simples fornece propriedades Country e Town para representar a casa da pessoa.

A listagem a seguir mostra o código completo das classes.

#### Listagem 8.2 Uma classe Person bastante simples usada para demonstrações adicionais

```
classe pública Pessoa
{
    public int Idade {obter; definir; } string pública
    Nome { get; definir; }

    List<Pessoa> amigos = new List<Pessoa>(); public List<Pessoa>
    Amigos { get { return amigos; } }

    Localização home = new Location(); local público
    Home { get { voltar para casa; } }

    pessoa pública() {}

    Pessoa pública (nome da string)
    {
        Nome = nome;
    }
}

classe pública Localização {

    string pública País {obter; definir; } string pública Cidade
    { get; definir; }
}
```

A Listagem 8.2 é direta, mas vale a pena notar que tanto a lista de amigos quanto o local de residência são criados em branco quando a pessoa é criada, em vez de serem deixados como referências nulas. As propriedades de amigos e localização residencial também são somente leitura. Isso será importante mais tarde — mas por enquanto vamos dar uma olhada nas propriedades que representam o nome e a idade de uma pessoa.

### 8.3.2 Configurando propriedades simples

Agora que você tem um tipo Person , é hora de criar algumas instâncias dele usando os novos recursos do C# 3. Nesta seção, veremos como definir as propriedades Name e Age — falaremos das outras mais tarde.

Os inicializadores de objetos são mais comumente usados para definir propriedades, mas tudo o que é mostrado aqui também se aplica a campos. Porém, em um sistema bem encapsulado, é improvável que você tenha acesso aos campos, a menos que esteja criando uma instância de um tipo dentro do próprio código desse tipo. Vale a pena saber que você pode usar campos, é claro – portanto, no restante da seção, basta ler *propriedade e campo* sempre que o texto indicar *propriedade*.

Com isso resolvido, vamos ao que interessa. Suponha que você queira criar uma pessoa chamada *Tom*, que tem 9 anos. Antes do C# 3, havia duas maneiras de conseguir isso:

```
Pessoa tom1 = new Pessoa(); tom1.Nome  
= "Tom"; tom1.Idade = 9;
```

```
Pessoa tom2 = new Pessoa("Tom"); tom2.Idade =  
9;
```

A primeira versão usa o construtor sem parâmetros e depois define ambas as propriedades. A segunda versão usa a sobrecarga do construtor, que define o nome e depois define a idade. Ambas as opções ainda estão disponíveis em C# 3, mas existem outras alternativas:

```
Pessoa tom3 = new Pessoa() { Nome = "Tom", Idade = 9 };  
Pessoa tom4 = nova Pessoa { Nome = "Tom", Idade = 9 };  
Pessoa tom5 = new Pessoa("Tom") { Idade = 9 };
```

A parte entre colchetes no final de cada linha é o inicializador do objeto. Novamente, é apenas um truque do compilador. O IL usado para inicializar tom3 e tom4 é idêntico e é quase o mesmo usado para tom1.

<sup>6</sup> Previsivelmente, o código do tom5 é quase o mesmo do tom2.

Observe como a inicialização do tom4 omite os parênteses do construtor.

Você pode usar esta abreviação para tipos com um construtor sem parâmetros, que é o que é chamado no código compilado.

Após o construtor ter sido chamado, as propriedades especificadas são definidas da maneira óbvia. Eles são definidos na ordem especificada no inicializador do objeto e você só pode especificar uma propriedade específica uma vez – não é possível definir a propriedade Name duas vezes, por exemplo. (Você poderia chamar o construtor tomando o nome como parâmetro e, em seguida, definir a propriedade Name . Seria inútil, mas o compilador não impediria você de fazer isso.) A expressão usada como valor para uma propriedade pode ser qualquer expressão que não seja uma atribuição — você pode chamar métodos, criar novos objetos (potencialmente usando outro inicializador de objeto), praticamente qualquer coisa.

Você pode estar se perguntando o quanto útil isso é – você salvou uma ou duas linhas de código, mas certamente essa não é uma razão boa o suficiente para tornar a linguagem mais complicada, não é? Porém, há um ponto útil aqui: você não criou apenas um objeto em uma linha – você o criou em uma expressão. Essa diferença pode ser muito importante.

Suponha que você queira criar um array do tipo Person[] com alguns dados predefinidos nele. Mesmo sem usar a digitação implícita de array que você verá mais tarde, o código é simples e legível:

---

<sup>6</sup> Na verdade, o novo valor de tom1 não é atribuído até que todas as propriedades tenham sido definidas. Uma variável local temporária é usada até então. Isso raramente é importante, mas vale a pena saber para evitar confusão se acontecer de você invadir o depurador no meio do inicializador.

```
Pessoa[] família = new Pessoa[] {

    nova Pessoa { Nome = "Holly", Idade = 36 }, nova Pessoa { Nome =
    "Jon", Idade = 36 }, nova Pessoa { Nome = "Tom", Idade = 9 },
    nova Pessoa { Nome = "William" , Idade = 6 }, nova Pessoa
    { Nome = "Robin", Idade = 6 }

};
```

Em um exemplo simples como este, você poderia ter escrito um construtor usando o nome e a idade como parâmetros e inicializado o array de maneira semelhante em C# 1 ou 2. Mas os construtores apropriados nem sempre estão disponíveis, e se houver vários parâmetros do construtor, muitas vezes não fica claro qual deles significa o quê, apenas pela posição. No momento em que um construtor precisa usar cinco ou seis parâmetros, muitas vezes me pego confiando no Intelli Sense mais do que gostaria. Usar os nomes das propriedades é uma grande vantagem para a legibilidade nesses casos.<sup>7</sup> Essa forma de inicializador

de objeto é a que você provavelmente usará com mais frequência. Mas existem dois outros formulários: um para definir subpropriedades e outro para adicionar coleções.

Vejamos primeiro as subpropriedades — propriedades de propriedades.

### 8.3.3 Configurando propriedades em objetos incorporados

Até agora tem sido fácil definir as propriedades Nome e Idade , mas você não pode definir a propriedade Casa da mesma maneira – ela é somente leitura. Você *pode* definir a cidade e o país de uma pessoa, primeiro buscando a propriedade Home e depois definindo as propriedades no resultado. A especificação da linguagem refere-se a isso como definir as propriedades de um *objeto incorporado*.

Só para deixar claro, estamos falando do seguinte código C# 1:

```
Pessoa tom = new Pessoa("Tom"); tom.Idade = 9;
tom.Home.Country
= "Reino Unido"; tom.Home.Town =
"Leitura";
```

Quando você preenche o local inicial, cada instrução faz um get para recuperar a instância Location e, em seguida, um conjunto na propriedade relevante nessa instância.

Não há nada de novo nisso, mas vale a pena desacelerar sua mente para olhar para isso com atenção; caso contrário, é fácil perder o que está acontecendo nos bastidores.

C# 3 permite que tudo isso seja feito em uma expressão, conforme mostrado aqui:

```
Pessoa tom = new Pessoa("Tom") {

    Idade = 9,
    Home = { País = "Reino Unido", Cidade = "Leitura" }
};
```

---

<sup>7</sup> C# 4 fornece uma abordagem alternativa aqui usando argumentos nomeados, que você conhecerá no capítulo 13.

O código compilado para esses trechos é efetivamente o mesmo. O compilador identifica que no lado direito do sinal = está outro inicializador de objeto e aplica as propriedades ao objeto incorporado de forma adequada.

A ausência da palavra-chave new na parte que inicializa Home é significativa. Se você precisar descobrir onde o compilador criará novos objetos e onde definirá propriedades nos existentes, procure ocorrências de new no inicializador.

Cada vez que um novo objeto é criado, a nova palavra-chave aparece *em algum lugar*.

**FORMATANDO O CÓDIGO INICIALIZADOR DE OBJETOS** Assim como acontece com quase todos os recursos do C#, os inicializadores de objetos são independentes de espaços em branco. Você pode recolher o espaço em branco no inicializador do objeto, colocando tudo em uma linha, se desejar. Cabe a você descobrir onde está o ponto ideal para equilibrar filas longas com muitas filas.

Já tratamos da propriedade da Casa , mas e os amigos do Tom? Existem propriedades que você pode definir em List<Person>, mas nenhuma delas adicionará entradas à lista. Chegou a hora do próximo recurso: inicializadores de coleção.

#### 8.3.4 Inicializadores de coleção

Criar uma coleção com alguns valores iniciais é uma tarefa extremamente comum. Até a chegada do C# 3, o único recurso da linguagem que ajudava era a criação de arrays, e mesmo isso era desajeitado em muitas situações. C# 3 possui *inicializadores de coleção*, que permitem usar o mesmo tipo de sintaxe dos inicializadores de array, mas com coleções arbitrárias e com mais flexibilidade.

##### CRIANDO NOVAS COLEÇÕES COM INICIALIZADORES DE COLEÇÃO

Como primeiro exemplo, vamos usar o agora familiar tipo List<T> . No C# 2, você poderia preencher uma lista passando uma coleção existente ou chamando Add repetidamente após criar uma lista vazia. Os inicializadores de coleção em C# 3 adotam a última abordagem.

Suponha que queiramos preencher uma lista de strings com alguns nomes — aqui está o código C# 2 (à esquerda) e o equivalente próximo em C# 3 (à direita):

```
Lista<string> nomes = new Lista<string>(); nomes.Add("Holly");
nomes.Add("Jon");
nomes.Add("Tom");
nomes.Add("Robin");
nomes.Add("William");
```

```
var nomes = new Lista<string> {
    "Holly", "Jon", "Tom",
    "Robin", "William"
};
```

Assim como acontece com inicializadores de objetos, você pode especificar argumentos de construtor, se desejar, ou usar um construtor sem parâmetros, explícita ou implicitamente. O uso da digitação implícita aqui foi parcialmente por razões de espaço — a variável nomes poderia igualmente ter sido declarada explicitamente. Reduzir o número de linhas de código (sem reduzir a legibilidade) é bom, mas há dois benefícios maiores nos inicializadores de coleção: ↴ A parte de criação e inicialização conta como uma

única expressão. ↴ Há muito menos confusão no código.

O primeiro ponto torna-se importante quando você deseja usar uma coleção como argumento para um método ou como um elemento em uma coleção maior. Isso acontece *relativamente* raramente (embora com frequência suficiente para ainda ser útil). O segundo ponto é o verdadeiro motivo pelo qual esse recurso é matador, na minha opinião. Se você olhar o código à direita, poderá ver facilmente as informações de que precisa, com cada informação escrita apenas uma vez. O nome da variável ocorre uma vez, o tipo usado ocorre uma vez e cada um dos elementos da coleção inicializada aparece uma vez. É tudo extremamente simples e muito mais claro que o código C# 2, que contém muitas informações sobre os bits úteis.

Os inicializadores de coleção não estão limitados apenas a listas. Você pode usá-los com qualquer tipo que implemente `IEnumerable`, desde que tenha um método `Add` apropriado para cada elemento no inicializador. Você pode usar um método `Add` com mais de um parâmetro colocando os valores dentro de outro conjunto de colchetes. O uso mais comum para isso é a criação de dicionários. Por exemplo, se você quiser um dicionário mapeando nomes para idades, poderá usar o seguinte código:

```
Dicionário<string,int> nameAgeMap = new Dicionário<string,int> {  
  
    {"Azevinho", 36},  
    {"João", 36},  
    {"Tom", 9}  
};
```

Nesse caso, o método `Add(string, int)` seria chamado três vezes. Se vários métodos `Add` estiverem disponíveis, diferentes elementos do inicializador poderão chamar diferentes overloads. Se nenhuma sobrecarga compatível estiver disponível para um elemento especificado, o código não será compilado. Existem dois pontos interessantes sobre a decisão de design aqui:

- ÿ O fato de o tipo ter que implementar `IEnumerable` nunca é usado pelo compilador.
- ÿ O método `Add` é encontrado apenas pelo nome – não há requisitos de interface especificando-o.

Ambas são decisões pragmáticas. Exigir que `IEnumerable` seja implementado é uma tentativa razoável de verificar se o tipo realmente é algum tipo de coleção, e usar qualquer sobrecarga acessível do método `Add` (em vez de exigir uma assinatura exata) permite inicializações simples, como o exemplo de dicionário anterior.

Um rascunho inicial da especificação C# 3 exigia que `ICollection<T>` fosse implementado, e a implementação do método `Add` de parâmetro único (conforme especificado pela interface) foi chamada em vez de permitir diferentes sobrecargas. Isso parece mais puro, mas há muito mais tipos que implementam `IEnumerable` do que `ICollection<T>`, e usar o método `Add` de parâmetro único seria inconveniente.

Por exemplo, neste caso, você teria forçado a criar explicitamente uma instância de `KeyValuePair<string,int>` para cada elemento do inicializador. Sacrificar um pouco da pureza acadêmica tornou a linguagem muito mais útil na vida real.

## POPULANDO COLEÇÕES DENTRO DE OUTROS INICIALIZADORES DE OBJETOS

Até agora vimos apenas inicializadores de coleção usados de forma independente para criar coleções totalmente novas. Eles também podem ser combinados com inicializadores de objetos para preencher coleções incorporadas. Para demonstrar isso, voltaremos ao exemplo Person .

A propriedade Friends é somente leitura, portanto você não pode criar uma nova coleção e especificá-la como a coleção de amigos, mas pode *adicionar* -la a qualquer coleção retornada pelo getter da propriedade. A maneira como você faz isso é semelhante à sintaxe que você já viu para definir propriedades de objetos incorporados, mas você apenas especifica um inicializador de coleção em vez de uma sequência de propriedades.

Vamos ver isso em ação criando outra instância Person para Tom, desta vez com alguns de seus amigos.

## Listagem 8.3 Construindo um objeto rico usando inicializadores de objeto e coleção

```
Pessoa tom = nova pessoa
{
    Define
    propriedades
    diretamente
    Nome = "Tom",
    Idade = 9,
    Home = { Cidade = "Leitura", País = "Reino Unido" },
    Amigos =
    {
        new Person { Nome = "Alberto" }, new Person("Max"),
        Person { Name = "Zak", Idade =
        7 }, new Person("Ben"), new Person("Alice"),
        {
            Idade = 9,
            Home = { Cidade = "Twyford", País = "Reino Unido" }
        }
    };
}
```

A Listagem 8.3 usa todos os recursos dos inicializadores de objetos e coleções que encontramos.

A parte principal de interesse é o inicializador de coleção, que usa internamente muitas formas diferentes de inicializadores de objetos. Observe que você não está criando uma nova coleção aqui, apenas adicionando uma já existente. (Se a propriedade tivesse um setter, você *poderia* criar uma nova coleção e ainda usar a sintaxe do inicializador de coleção.)

Você poderia ter ido mais longe, especificando amigos de amigos, amigos de amigos de amigos e assim por diante. Mas não foi possível especificar que Tom é amigo de Alberto. Enquanto você ainda está inicializando um objeto, você não tem acesso a ele, portanto não pode expressar relacionamentos cíclicos. Isso pode ser estranho em alguns casos, mas geralmente não é um problema.

A inicialização de coleção dentro de inicializadores de objetos funciona como uma espécie de cruzamento entre inicializadores de coleção independentes e a configuração de propriedades de objetos incorporadas. Para cada elemento no inicializador da coleção, o getter da propriedade da coleção (Friends, neste caso) é chamado e, em seguida, o método Add apropriado é chamado no valor retornado. A coleção não é limpa de forma alguma antes que os elementos sejam adicionados. Por exemplo, se você decidisse que uma pessoa deveria ser sempre sua amiga e adicionasse isso ao

lista de amigos dentro do construtor Person , usar um inicializador de coleção apenas adicionaria amigos extras.

Como você pode ver, a combinação de inicializadores de coleção e de objeto pode ser usada para preencher árvores inteiras de objetos. Mas quando e onde é provável que isso realmente aconteça?

### 8.3.5 Usos de recursos de inicialização

Tentar definir exatamente onde esses recursos são úteis é uma reminiscência de estar em um esboço do Monty Python sobre a Inquisição Espanhola – toda vez que você acha que tem uma lista razoavelmente completa, outro exemplo comum aparece. Mencionarei apenas três exemplos, que espero que o encorajem a considerar onde mais você poderia usá-los.

#### COLEÇÕES CONSTANTES

Não é incomum eu querer algum tipo de coleção (geralmente um mapa) que seja efetivamente constante. Claro, não pode ser uma constante no que diz respeito à linguagem C#, mas pode ser declarada estática e somente leitura, com grandes avisos para dizer que não deve ser alterada. (Geralmente é privado, então é bom o suficiente. Como alternativa, você pode usar Read OnlyCollection<T>.) Normalmente, isso envolvia escrever um construtor estático ou um método auxiliar, apenas para preencher o mapa. Com os inicializadores de coleção do C# 3, é fácil configurar tudo inline.

#### CONFIGURANDO TESTES DE UNIDADE

Ao escrever testes de unidade, frequentemente desejo preencher um objeto apenas para um teste, muitas vezes passando-o como um argumento para o método que estou tentando testar no momento. Escrever toda a inicialização à mão pode ser demorado e também oculta a estrutura essencial do objeto do leitor do código, assim como o código de criação XML pode muitas vezes obscurecer a aparência do documento se você o visualizasse (formatado apropriadamente) em um editor de texto. Com a indentação apropriada dos inicializadores de objetos, a estrutura aninhada da hierarquia de objetos pode se tornar óbvia na própria forma do código, bem como fazer com que os valores se destaquem mais do que seriam de outra forma.

#### O PADRÃO DO CONSTRUTOR

Por vários motivos, às vezes você deseja especificar muitos valores para um único método ou chamada de construtor. A situação mais comum na minha experiência é a criação de um objeto imutável. Em vez de ter um conjunto enorme de parâmetros (que pode se tornar um problema de legibilidade à medida que o significado de cada argumento se torna obscuro<sup>8</sup> ), você pode usar o padrão *construtor* - criar um tipo mutável com propriedades apropriadas e depois passar uma instância do construtor para o construtor ou método. O processo- quadro

O tipo StartInfo é um bom exemplo disso - os designers *poderiam* ter sobre carregado Process.Start com muitos conjuntos diferentes de parâmetros, mas usar ProcessStartInfo torna tudo mais claro.

Os inicializadores de objetos e coleções permitem que você crie o objeto construtor de uma maneira mais clara — você pode até mesmo especificá-lo em linha ao chamar o membro original, se desejar. É certo que você ainda precisa escrever o tipo de construtor em primeiro lugar, mas as propriedades automáticas ajudam nesse aspecto.

<sup>8</sup> Argumentos nomeados em C# 4 ajudam nessa área, é certo.

## &lt;INSIRA SEU USO FAVORITO AQUI&gt;

É claro que existem usos além desses três no código comum, e não quero impedir que você use os novos recursos em outro lugar. Há poucos motivos para *não* usá-los, além de possivelmente confundir os desenvolvedores que ainda não estão familiarizados com o C# 3. Você pode decidir que usar um inicializador de objeto apenas para definir uma propriedade (em vez de defini-la explicitamente em uma instrução separada) é um exagero - isso é uma questão de estética, e não posso lhe dar muita orientação objetiva nisso. Assim como acontece com a digitação implícita, é uma boa ideia testar o código nos dois sentidos e aprender a prever suas próprias preferências de leitura (e as de sua equipe).

Até agora, vimos uma gama bastante diversificada de recursos: implementação fácil de propriedades, simplificação de declarações de variáveis locais e preenchimento de objetos em expressões únicas. No restante deste capítulo, reuniremos gradualmente esses tópicos, usando mais tipagem implícita e mais população de objetos, e criando *tipos* inteiros sem fornecer quaisquer detalhes de implementação.

O próximo tópico parece ser bastante semelhante aos inicializadores de coleção quando você analisa o código que os utiliza. Mencionei anteriormente que a inicialização do array era um pouco desajeitada no C# 1 e 2. Tenho certeza de que você não ficará surpreso ao saber que ela foi simplificada para o C# 3. Vamos dar uma olhada.

#### 8.4 Arrays digitados implicitamente Em C#

1 e 2, inicializar um array como parte de uma declaração de variável e de uma instrução de inicialização era bastante simples, mas se você quisesse fazer isso em qualquer outro lugar, teria que especificar o tipo exato de array envolvido. Por exemplo, isso compila sem nenhum problema:

```
string[] nomes = {"Holly", "Jon", "Tom", "Robin", "William"};
```

Porém, isso não funciona para parâmetros - suponha que você queira fazer uma chamada para MyMethod, declarada como void MyMethod(string[] nomes). Este código não funcionará:

INVÁLIDO

```
MeuMetodo({"Holly", "Jon", "Tom", "Robin", "William"});
```

Em vez disso, você deve informar ao compilador que tipo de array deseja inicializar:

```
MeuMetodo(new string[] {"Holly", "Jon", "Tom", "Robin", "William"});
```

C# 3 permite algo intermediário:

```
MeuMetodo(new[] {"Holly", "Jon", "Tom", "Robin", "William"});
```

Claramente, o compilador precisa descobrir que tipo de array usar. Ele começa formando um conjunto contendo todos os tipos de expressões em tempo de compilação entre colchetes. Se houver exatamente um tipo nesse conjunto para o qual todos os outros possam ser convertidos implicitamente, esse será o tipo do array. Caso contrário (ou se todos os valores forem expressões sem tipo, como valores nulos constantes ou métodos anônimos, sem conversão), o código não será compilado.

Observe que apenas os tipos de expressões são considerados candidatos para o tipo de array geral. Isso significa que ocasionalmente pode ser necessário converter explicitamente um valor para um tipo menos específico. Por exemplo, isso não irá compilar:

INVÁLIDO

```
novo[] { novo MemoryStream(), novo StringWriter() }
```

Não há conversão de `MemoryStream` para `StringWriter` ou vice-versa. Ambos são implicitamente conversíveis em `object` e `IDisposable`, mas o compilador considera apenas os tipos que estão no conjunto original produzido pelas próprias expressões. Se você alterar uma das expressões nesta situação para que seu tipo seja `object` ou `IDisposable`, o código será compilado:

```
novo[] { (IDisposable) novo MemoryStream(), novo StringWriter() }
```

O tipo desta última expressão é implicitamente `IDisposable[]`. É claro que, nesse ponto, você também pode declarar explicitamente o tipo da matriz, assim como faria em C# 1 e 2, para deixar mais claro o que você está tentando alcançar.

Comparados com os recursos anteriores, os arrays digitados implicitamente são um pouco anticlimáticos. Acho difícil ficar entusiasmado com eles, embora eles *simplifiquem* a vida nos casos em que um array é passado como argumento. Os designers não enlouqueceram – há uma situação importante em que essa digitação implícita é absolutamente crucial. É quando você não sabe (e *não pode saber*) o nome do tipo dos elementos do array. Como você pode entrar nesse estado peculiar? Leia...

## 8.5 Tipos anônimos A digitação

implícita, os inicializadores de objetos e de coleções e a digitação implícita de arrays são úteis por si só, em maior ou menor grau. Mas eles também servem a um propósito mais elevado: tornam possível trabalhar com o recurso final deste capítulo, os *tipos anônimos*.

Por sua vez, os tipos anônimos atendem ao propósito maior do LINQ.

### 8.5.1 Primeiros encontros do tipo anônimo

É muito mais fácil explicar tipos anônimos quando você já tem alguma ideia do que são por meio de um exemplo. Lamento dizer que sem o uso de métodos de extensão e expressões lambda, os exemplos nesta seção provavelmente serão um pouco inventados, mas há uma situação do ovo e da galinha aqui: tipos anônimos são mais úteis dentro do contexto dos recursos mais avançados, mas precisamos cobrir os blocos de construção antes de podermos olhar para o quadro geral. Continue assim – *fará sentido no longo prazo*, eu prometo.

Vamos fingir que não temos a classe `Person` e que as únicas propriedades com as quais nos importamos são o nome e a idade. A listagem a seguir mostra como você ainda pode construir objetos com essas propriedades, sem nunca declarar um tipo.

#### Listagem 8.4 Criando objetos de tipo anônimo com propriedades Nome e Idade

```
var tom= new { Nome = "Tom", Idade = 9 }; var azevinho =
novo { Nome = "Azevinho", Idade = 36 }; var jon = new { Nome =
"Jon", Idade = 36 } ;

Console.WriteLine("{0} tem {1} anos", jon.Name, jon.Age);
```

Como você pode ver na listagem 8.4, a sintaxe para inicializar um tipo anônimo é semelhante aos inicializadores de objeto que você viu na seção 8.3.2 – só que o nome do tipo é

faltando entre new e a chave de abertura. Aqui você está usando local digitado implicitamente variáveis porque isso é tudo que você *pode* usar (além do objeto , é claro) - você não tem um nome de tipo para declarar a variável. Como você pode ver na última linha, o tipo tem propriedades para Nome e Idade, ambas podem ser lidas e terão os valores especificado no *inicializador de objeto anônimo* usado para criar a instância, portanto, neste caso o resultado é que Jon tem 36 anos. As propriedades têm os mesmos tipos que as expressões nos inicializadores – string para Name e int para Age. Assim como em inicializadores de objetos normais, as expressões usadas em inicializadores de objetos anônimos podem chamar métodos ou construtores, buscar propriedades, realizar cálculos – tudo o que você precisar fazer.

Agora você pode estar começando a entender por que arrays digitados implicitamente são importantes. Suponha que você queira criar um array contendo toda a família e então iterar através dele para calcular a idade total.<sup>9</sup> A listagem a seguir faz exatamente isso e demonstra algumas outras características interessantes dos tipos anônimos ao mesmo tempo.

**Listagem 8.5 Preenchendo um array usando tipos anônimos e depois encontrando a idade total**

```
var família = novo[]
{
    novo {Nome = "Holly", Idade = 36 },
    novo {Nome = "Jon", Idade = 36 },
    novo {Nome = "Tom", Idade = 9 },
    novo {Nome = "Robin", Idade = 6 },
    novo {Nome = "William", Idade = 6 }
};

int idade total = 0;
foreach (var pessoa na família)
{
    idade total += pessoa.Idade;
}
Console.WriteLine("Idade total: {0}", totalAge);
```

- A: Usa um digitado implicitamente  
Inicializador de matriz B
- B: Usa o mesmo tipo anônimo  
C cinco vezes
- C: Usa digitação implícita para pessoa
- D: Usa digitação implícita para pessoa
- E: Soma idades

Juntando a listagem 8.5 e o que você aprendeu sobre matrizes tipificadas implicitamente na seção 8.4, você pode deduzir algo importante: *todas as pessoas da família têm a mesma identidade. tipo*. Se cada uso de um inicializador de objeto anônimo C se referisse a um tipo diferente, o compilador não pôde inferir um tipo apropriado para o array B. Dentro de qualquer assembly, o compilador trata dois inicializadores de objetos anônimos como o mesmo tipo se houver o mesmo número de propriedades, com os mesmos nomes e tipos na mesma ordem. Em outras palavras, se você trocou as propriedades Name e Age em um dos inicializadores, haveria dois tipos diferentes envolvidos; da mesma forma, se você introduziu uma propriedade extra em uma linha, ou usado um long em vez de um int para a idade de uma pessoa, outro tipo anônimo teria sido introduzido. Nesse ponto, a inferência de tipo para a matriz falharia.

<sup>9</sup> Se você já conhece o LINQ, pode achar que esta é uma maneira curiosa de somar as idades. Eu concordo, ligando family.Sum(p => p.Age) seria muito mais simples, mas vamos dar um passo de cada vez.

```

int totalAge = 0;
foreach (var person in family)
{
    totalAge += person.Age;
}

```

(local variable) 'a person  
Anonymous Types:  
'a is new { string Name, int Age }

**Figura 8.4** Passando o mouse sobre uma variável declarada (implicitamente) ser de um tipo anônimo mostra os detalhes desse tipo anônimo.

**DETALHE DE IMPLEMENTAÇÃO: QUANTOS TIPOS?** Se você decidir olhar para o IL (ou C# descompilado) para um tipo anônimo gerado pelo compilador da Microsoft, esteja ciente de que, embora dois inicializadores de objetos anônimos com o mesmo nomes de propriedades na mesma ordem, mas usando tipos de propriedades diferentes, produzirão dois tipos diferentes; na verdade, eles serão gerados a partir de um único genérico tipo. O tipo genérico é parametrizado, mas os tipos fechados e construídos serão diferentes porque eles receberão argumentos de tipo diferentes para os diferentes inicializadores.

Observe que você pode usar uma instrução foreach para iterar sobre o array, assim como faria qualquer outra coleção. O tipo envolvido é D inferido, e o tipo da variável person é o mesmo tipo anônimo usado no array. Novamente, você pode usar o mesmo variável para instâncias diferentes porque são todas do mesmo tipo.

A Listagem 8.5 também prova que a propriedade Age realmente é fortemente tipada como um int—caso contrário, tentar somar as idades E não seria compilado. O compilador sabe sobre o tipo anônimo, e o Visual Studio está até disposto a compartilhar as informações por meio de dicas de ferramentas, caso você não tenha certeza. A Figura 8.4 mostra o resultado de passar o mouse sobre a pessoa parte da pessoa. Expressão de idade da listagem 8.5.

Agora que você viu os tipos anônimos em ação, vamos voltar e ver o que compilador está realmente fazendo.

### 8.5.2 Membros de tipos anônimos

Tipos anônimos são criados pelo compilador e incluídos no assembly compilado da mesma forma que os tipos extras para métodos anônimos e blocos iteradores. O CLR os trata como tipos perfeitamente comuns, e assim são - se mais tarde você mudar de um tipo anônimo para um tipo normal codificado manualmente com o comportamento descrito em esta seção, você não deverá ver nada mudar.

Os tipos anônimos contêm os seguintes membros:

- ÿ Um construtor que recebe todos os valores de inicialização. Os parâmetros estão na mesma ordem em que foram especificados no inicializador do objeto anônimo, e eles têm os mesmos nomes e tipos.
- ÿ Propriedades públicas somente leitura.
- ÿ Campos privados somente leitura que respaldam as propriedades.
- ÿ Substituições para Equals, GetHashCode e ToString.

É isso. Não há interfaces implementadas, nem recursos de clonagem ou serialização – apenas um construtor, algumas propriedades e os métodos normais do objeto.

O construtor e as propriedades fazem as coisas óbvias. A igualdade entre duas instâncias do mesmo tipo anônimo é determinada de maneira natural, comparando cada valor de propriedade por vez usando o método Equals do tipo de propriedade . A geração do código hash é semelhante, chamando GetHashCode em cada valor de propriedade e combinando os resultados. O método exato para combinar os vários códigos hash para formar um hash composto não é especificado e, de qualquer maneira, você não deve escrever código que dependa dele - você só precisa ter certeza de que duas instâncias iguais retornarão o mesmo hash e duas instâncias desiguais. instâncias *geralmente* retornarão hashes diferentes.

Tudo isso só funciona se as implementações Equals e GetHashCode de todos os diferentes tipos envolvidos como propriedades estiverem em conformidade com as regras normais, é claro.

Como as propriedades são somente leitura, todos os tipos anônimos são imutáveis, desde que os tipos usados para suas propriedades sejam imutáveis. Isso fornece todos os benefícios normais da imutabilidade – ser capaz de passar valores para métodos sem medo de que eles sejam alterados, simples compartilhamento de dados entre threads e assim por diante.

**PROPRIEDADES DE TIPO ANÔNIMO DO VB SÃO MUTÁVEIS POR PADRÃO** Os tipos anônimos também estão disponíveis no Visual Basic 9 em diante. Mas, por padrão, suas propriedades são mutáveis; você precisa declarar quaisquer propriedades que deseja que sejam imutáveis com o modificador Key . Somente propriedades declaradas como chaves são usadas em comparações de hash e igualdade. É fácil ignorar isso ao converter código de uma linguagem para outra.

Estamos quase terminando com os tipos anônimos agora. Mas ainda há um pequeno problema para falar: um atalho para uma situação que é bastante comum no LINQ.

### 8.5.3 Inicializadores de projeção

Os inicializadores de objetos anônimos que você viu até agora são listas de pares nome/valor — Name="Jon", Age=36 e similares. Acontece que sempre usei constantes porque elas servem para exemplos menores, mas em código real muitas vezes você deseja copiar propriedades de um objeto existente. Às vezes você desejará manipular os valores de alguma forma, mas muitas vezes uma cópia direta é suficiente.

Novamente, sem o LINQ é difícil dar exemplos convincentes disso, mas vamos voltar à nossa classe Person e *supor* que tínhamos um bom motivo para querer converter uma coleção de instâncias Person em uma coleção semelhante onde cada elemento tem apenas um nome e um bandeira para dizer se essa pessoa é adulta. Dada uma variável person apropriada , você poderia usar algo assim:

```
novo { Nome = pessoa.Nome, ÉAdulto = (pessoa.Idade >= 18) }
```

Isso funciona, e para apenas uma propriedade a sintaxe para definir o nome (a parte em negrito) não é muito desajeitada, mas se você estivesse copiando várias propriedades ficaria cansativo.

C# 3 fornece um atalho: se você não especificar o nome da propriedade, mas apenas a expressão a ser avaliada para o valor, ele usará a última parte da expressão como o

nome, desde que seja um campo ou propriedade simples. Isso é chamado de *inicializador de projeção*. Isso significa que você pode reescrever o código anterior da seguinte maneira:

```
novo { pessoa.Nome, ÉAdulto = (pessoa.Idade >= 18) }
```

É comum que todos os bits de um inicializador de objeto anônimo sejam inicializadores de projeção — isso normalmente acontece quando você obtém algumas propriedades de um objeto e algumas propriedades de outro, geralmente como parte de uma operação de junção. De qualquer forma, estou me adiantando.

A listagem a seguir mostra o código anterior em ação, usando o método `List<T>.ConvertAll` e um método anônimo.

#### Listagem 8.6 Transformação de Pessoa em nome e bandeira de idade adulta

```
List<Pessoa> família = new List<Pessoa>
{
    nova Pessoa { Nome = "Holly", Idade = 36 }, nova Pessoa { Nome = "Jon",
        Idade = 36 }, nova Pessoa { Nome = "Tom", Idade = 9 }, nova Pessoa
        { Nome = "Robin" , Idade = 6 }, nova Pessoa { Nome = "William" , Idade
        = 6 }

};

var convertido = family.ConvertAll(delegado(Pessoa pessoa)
    { return new { pessoa.Nome, IsAdulto = (pessoa.Idade >= 18) }; }
);

foreach (var pessoa em convertido) {

    Console.WriteLine("{0} é adulto? {1}",
        pessoa.Nome, pessoa.IsAdulto);
}
```

Além do uso de um inicializador de projeção para a propriedade `Name`, a listagem 8.6 mostra o valor da inferência de tipo delegado e dos métodos anônimos. Sem eles, você não poderia ter mantido a digitação forte de `convert`, porque não seria capaz de especificar qual deveria ser o parâmetro de tipo `TOutput` do `Converter`. Do jeito que está, você pode percorrer a nova lista e acessar as propriedades `Name` e `IsAdult` como se estivesse usando qualquer outro tipo.

Não perca muito tempo pensando em inicializadores de projeção neste momento - o importante é estar ciente de que eles existem para não ficar confuso quando os vir mais tarde. Na verdade, esse conselho se aplica a toda esta seção sobre tipos anônimos; portanto, sem entrar em detalhes, vamos ver por que eles estão presentes.

#### 8.5.4 Qual é o objetivo?

Espero que você não esteja se sentindo enganado neste momento, mas simpatizo se estiver. Os tipos anônimos são uma solução bastante complexa para um problema que ainda não encontramos. Mas aposto que você já viu parte do problema antes, na verdade.

Se você já fez algum trabalho real envolvendo bancos de dados, sabe que nem sempre deseja todos os dados disponíveis em todas as linhas que correspondem à sua consulta

critério. Muitas vezes não é um problema buscar mais do que o necessário, mas se você precisar apenas de 2 colunas das 50 da tabela, não se preocuparia em selecionar todas as 50, não é?

O mesmo problema ocorre em código que não é de banco de dados. Suponha que você tenha uma classe que lê um arquivo de log e produz uma sequência de linhas de log com muitos campos. Manter todas as informações pode consumir muita memória se você se preocupar apenas com alguns campos do log. O LINQ permite filtrar essas informações facilmente.

Mas qual é o resultado dessa filtragem? Como você pode manter alguns dados e descartar o resto? Como você pode manter facilmente alguns dados *derivados* que não estão representados diretamente no formato original? Como você pode combinar dados que podem não ter sido inicialmente associados de forma consciente ou que podem ter um relacionamento apenas em uma situação específica? Na verdade, você deseja um novo tipo de dados, mas criar manualmente esse tipo em todas as situações é entediante, especialmente quando você tem ferramentas como o LINQ disponíveis, que tornam o resto do processo tão simples. A Figura 8.5 mostra os três elementos que tornam os tipos anônimos um recurso poderoso.

Se você estiver criando um tipo que é usado apenas em um único método e que contém apenas campos e propriedades triviais, considere se um tipo anônimo seria apropriado. Eu suspeito que normalmente, quando você se inclina para tipos anônimos, você também pode usar o LINQ para ajudá-lo.

Porém, se você estiver usando a mesma sequência de propriedades para o mesmo propósito em vários lugares, talvez queira considerar a criação de um tipo normal para esse propósito, mesmo que ele ainda contenha apenas propriedades triviais. Os tipos anônimos infectam naturalmente qualquer código em que são usados com digitação implícita, o que geralmente é bom, mas pode ser um incômodo em outros momentos. Em particular, significa que você não pode criar facilmente um método para retornar uma instância desse tipo de maneira fortemente tipada. Tal como acontece com os recursos anteriores, use tipos anônimos quando eles realmente tornarem o código mais simples de trabalhar, não apenas porque são novos e interessantes.



**Figura 8.5** Os tipos anônimos permitem manter apenas os dados necessários para uma situação específica, em um formato adaptado a essa situação, sem o tédio de escrever um novo tipo a cada vez.

## 8.6 Resumo

Que conjunto aparentemente misto de recursos! Você viu quatro recursos que são bastante semelhantes, pelo menos na sintaxe: inicializadores de objetos, inicializadores de coleção, arrays digitados implicitamente e tipos anônimos. Os outros dois recursos – propriedades automáticas e variáveis locais digitadas implicitamente – são um pouco diferentes. Da mesma forma, a maioria dos recursos teria sido útil individualmente no C# 2, enquanto matrizes digitadas implicitamente e tipos anônimos apenas pagam o custo de aprender sobre eles quando o restante dos recursos do C# 3 são colocados em ação.

Então, o que esses recursos realmente têm em comum? *Todos eles aliviam o desenvolvedor da codificação tediosa.* Tenho certeza de que você não gosta mais de escrever propriedades triviais do que eu, ou de definir diversas propriedades, uma de cada vez, usando uma variável local — principalmente quando você está tentando construir uma coleção de objetos semelhantes. Os novos recursos do C# 3 não apenas facilitam a *escrita* do código, mas também facilitam a *leitura*, pelo menos quando aplicados de maneira sensata.

No próximo capítulo, veremos um novo recurso importante da linguagem, juntamente com um recurso de estrutura para o qual ele fornece suporte direto. Se você achava que os métodos anônimos facilitavam a criação de delegados, espere até ver as expressões lambda.

# Expressões lambda e árvores de expressão

## Este capítulo cobre

- ÿ Sintaxe da expressão lambda
- ÿ Conversões de lambdas para delegados
- ÿ Classes de estrutura de árvore de expressão
- ÿ Conversões de lambdas em árvores de expressão
- ÿ Por que as árvores de expressão são importantes
- ÿ Mudanças na inferência de tipo e resolução de sobrecarga

No capítulo 5, você viu como o C# 2 tornou os delegados muito mais fáceis de usar devido às conversões implícitas de grupos de métodos, métodos anônimos e tipo de retorno e variação de parâmetro. Isso é suficiente para tornar a assinatura de eventos significativamente mais simples e legível, mas os delegados em C# 2 ainda são muito volumosos para serem usados o tempo todo; uma página de código cheia de métodos anônimos é difícil de ler, e você não gostaria de começar a colocar vários métodos anônimos em uma única instrução regularmente.

Um dos blocos de construção fundamentais do LINQ é a capacidade de criar pipelines de operações, juntamente com qualquer estado exigido por essas operações. Estas operações podem expressar todos os tipos de lógica sobre os dados: como filtrá-los, como ordená-los, como

para unir diferentes fontes de dados e muito mais. Quando consultas LINQ são executadas em processo, essas operações geralmente são representadas por delegados.

Instruções contendo vários delegados são comuns ao manipular dados com LINQ to Objects,<sup>1</sup> e expressões *lambda* em C# 3 tornam tudo isso possível sem sacrificar a legibilidade. (Embora eu mencione a legibilidade, este capítulo usa expressão *lambda* e *lambda* de forma intercambiável.)

**É TUDO GREGO PARA MIM** O termo expressão *lambda* vem do cálculo *lambda*, também escrito como cálculo  $\lambda$ , onde  $\lambda$  é a letra grega lambda. Esta é uma área da matemática e da ciência da computação que trata da definição e aplicação de funções. Já existe há muito tempo e é a base de linguagens funcionais como ML. A boa notícia é que você não precisa saber cálculo lambda para usar expressões lambda em C# 3.

A execução de delegados é apenas parte da história do LINQ. Para usar bancos de dados e outros mecanismos de consulta com eficiência, você precisa de uma representação diferente das operações no pipeline – uma maneira de tratar o código como dados que podem ser examinados programaticamente. A lógica dentro das operações pode então ser transformada em um formato diferente, como uma chamada de serviço web, uma consulta SQL ou LDAP – o que for apropriado.

Embora seja possível construir representações de consultas em uma API específica, geralmente é difícil de ler e sacrifica muito suporte do compilador. É aqui que os lambdas salvam o dia novamente: eles não apenas podem ser usados para criar instâncias delegadas, mas o compilador C# também pode transformá-los em árvores *de expressão* – estruturas de dados que representam a lógica das expressões lambda – para que outro código possa examiná-lo. Resumindo, as expressões lambda são a forma idiomática de representar as operações nos pipelines de dados do LINQ — mas daremos um passo de cada vez, examinando-as de uma forma bastante isolada antes de abraçarmos o LINQ como um todo.

Neste capítulo veremos ambas as maneiras de usar expressões lambda, embora no momento nossa cobertura de árvores de expressão seja relativamente básica – não criaremos nenhum SQL ainda. Com essa teoria sob controle, você deverá estar relativamente confortável com expressões lambda e árvores de expressão quando chegarmos ao que é realmente impressionante no capítulo 12.

Na parte final deste capítulo, examinaremos como a inferência de tipos mudou no C# 3, principalmente devido a lambdas com tipos de parâmetros implícitos. É um pouco como aprender a amarrar cadarços: longe de ser emocionante, mas sem essa habilidade você tropeçará quando começar a correr.

Vamos começar vendo como são as expressões lambda. Começaremos com um anonimato método específico e gradualmente transformá-lo em formas cada vez mais curtas.

---

<sup>1</sup> LINQ to Objects lida com sequências de dados dentro do mesmo processo. Por outro lado, provedores como LINQ to SQL transferem o trabalho para outros sistemas fora de processo – bancos de dados, por exemplo.

## 9.1 Expressões lambda como delegados

De muitas

maneiras, as expressões lambda podem ser vistas como uma evolução dos métodos anônimos do C# 2. As expressões lambda podem fazer quase tudo que os métodos anônimos podem e são quase sempre mais legíveis e compactas.<sup>2</sup> Em particular, o comportamento das variáveis capturadas é exatamente o mesmo nas expressões lambda e nos métodos anônimos. Na forma mais explícita, não existe muita diferença entre as duas, mas as expressões lambda possuem muitos atalhos disponíveis que as tornam compactas em situações comuns. Assim como os métodos anônimos, as expressões lambda têm regras de conversão especiais — o tipo da expressão não é um tipo delegado em si, mas pode ser convertido em uma instância delegada de várias maneiras, tanto implícita quanto explicitamente. O termo *função anônima* abrange métodos anônimos e expressões lambda e, em muitos casos, as mesmas regras de conversão se aplicam a ambos.

Começaremos com um exemplo simples, inicialmente expresso como um método anônimo. Você criará uma instância delegada que usa um parâmetro de string e retorna um int (que é o comprimento da string). Primeiro você precisa escolher um tipo de delegado a ser usado; felizmente, o .NET 3.5 vem com toda uma família de tipos genéricos de delegados para ajudá-lo.

### 9.1.1 Preliminares: Apresentando os tipos de delegado Func<...>

Existem cinco tipos genéricos de delegados Func no namespace System do .NET 3.5. Não há nada de especial em Func — é apenas útil ter alguns tipos genéricos predefinidos que são capazes de lidar com muitas situações. Cada assinatura de delegado leva entre zero e quatro parâmetros, cujos tipos são especificados como parâmetros de tipo.<sup>3</sup> O último parâmetro de tipo é usado para o tipo de retorno em cada caso.

Aqui estão as assinaturas de todos os tipos de delegados Func no .NET 3.5:

```
Função TResult<TResult>()
Função TResult<T,TResult>(T arg)
Função TResult<T1,T2,TResult>(T1 arg1, T2 arg2)
Função TResult<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
Função TResult<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

Por exemplo, Func<string,double,int> é equivalente a um tipo de delegado no formato public delegado int SomeDelegate(string arg1, double arg2)

O conjunto de delegados Action<...> fornece a funcionalidade equivalente quando você deseja um tipo de retorno void. A forma de parâmetro único de Action existia no .NET 2.0, mas o restante é novo no .NET 3.5. Se quatro argumentos não forem suficientes para você, então o .NET 4 tem a resposta: ele expande as famílias Action<...> e Func<...> para receber até 16 argumentos, então Func<T1,...,T16,TResult> tem 17 parâmetros de tipo impressionantes.

Isto é principalmente para ajudar a suportar o Dynamic Language Runtime (DLR) que você conhecerá no capítulo 14, e é improvável que você precise lidar com ele diretamente.

<sup>2</sup> O único recurso disponível para métodos anônimos, mas não para expressões lambda, é a capacidade de ignorar parâmetros de forma concisa.

Consulte a seção 5.4.3 para obter mais detalhes se estiver interessado, mas na prática não é algo que você realmente sentirá falta com expressões lambda.

<sup>3</sup> Você deve se lembrar que conheceu a versão sem nenhum parâmetro (mas um parâmetro de tipo) no capítulo 6.

Para este exemplo, você precisa de um tipo que receba um parâmetro de string e retorne um int, então você pode usar Func<string,int>.

### 9.1.2 Primeira transformação para uma expressão lambda

Agora que você conhece o tipo de delegado, pode usar um método anônimo para criar sua instância de delegado. A listagem a seguir mostra isso e executa a instância delegada posteriormente, para que você possa vê-la funcionando.

#### Listagem 9.1 Usando um método anônimo para criar uma instância delegada

```
Func<string,int> returnLength; returnLength =  
delegado (string texto) { return text.Length; };  
  
Console.WriteLine(returnLength("Olá"));
```

A listagem 9.1 imprime 5, exatamente como seria de esperar. Separei a declaração de return Length da atribuição a ela, para mantê-la em uma linha – é mais fácil acompanhar dessa forma. A expressão do método anônimo está em negrito; essa é a parte que você converterá em uma expressão lambda.

A forma mais prolixo de uma expressão lambda é esta:

```
(lista de parâmetros digitados explicitamente) => { instruções }
```

A parte => é nova no C# 3 e informa ao compilador que você está usando uma expressão lambda. Na maioria das vezes, as expressões lambda são usadas com um tipo delegado que possui um tipo de retorno não nulo, e a sintaxe é um pouco menos intuitiva quando não há resultado. Esta é outra indicação das mudanças no idioma entre C# 1 e C# 3. No C# 1, os delegados geralmente eram usados para eventos e raramente retornavam algo. No LINQ eles geralmente são usados como parte de um pipeline de dados, recebendo entrada e retornando um resultado para dizer qual é o valor projetado ou se o item corresponde ao filtro atual e assim por diante.

Com os parâmetros explícitos e instruções entre colchetes, esta versão parece muito semelhante a um método anônimo. A listagem a seguir é equivalente à listagem 9.1, mas usa uma expressão lambda.

#### Listagem 9.2 Uma primeira expressão lambda prolixo, semelhante a um método anônimo

```
Func<string,int> returnLength; returnLength =  
(string texto) => { return text.Length; };  
  
Console.WriteLine(returnLength("Olá"));
```

Novamente, usei negrito para indicar a expressão usada para criar a instância delegada. Ao ler expressões lambda, ajuda pensar na parte => como “vai para”, então o exemplo na listagem 9.2 poderia ser lido “o texto vai para o texto.Comprimento”. Como esta é a única parte da listagem que é interessante por enquanto, vou mostrá-la sozinha a partir de agora. Você pode substituir o texto em negrito da listagem 9.2 por qualquer uma das expressões lambda listadas nesta seção e o resultado será o mesmo.

As mesmas regras que regem as instruções de retorno em métodos anônimos se aplicam aos lambdas: você não pode tentar retornar um valor de uma expressão lambda com um tipo de retorno void, ao passo que, se houver um tipo de retorno não nulo, todo caminho de código deverá retornar um valor compatível. value.<sup>4</sup> É tudo bastante intuitivo e raramente atrapalha.

Até agora, não economizamos muito espaço nem tornamos as coisas particularmente fáceis de ler. Vamos comece a aplicar os atalhos.

### 9.1.3 Usando uma única expressão como corpo

O formulário que vimos até agora usa um bloco completo de código para retornar o valor. Isso é flexível – você pode ter múltiplas instruções, executar loops, retornar de diferentes locais do bloco e assim por diante, assim como acontece com métodos anônimos. Mas na maioria das vezes, você pode facilmente expressar todo o corpo em uma única expressão, cujo valor é o resultado do lambda.<sup>5</sup> Nesses casos, você pode especificar apenas essa expressão, sem colchetes, retornando instruções ou ponto e vírgula. O formato então é

(lista de parâmetros digitados explicitamente) => expressão

No nosso exemplo, isso significa que a expressão lambda se torna

(string texto) => texto.Comprimento

Isso já está começando a parecer mais simples. Agora, e quanto a esse tipo de parâmetro? O compilador já sabe que as instâncias de Func<string,int> aceitam um único parâmetro de string, então você deve poder apenas nomear esse parâmetro.

### 9.1.4 Listas de parâmetros digitados implicitamente

Na maioria das vezes, o compilador pode adivinhar os tipos de parâmetros sem que você os declare explicitamente. Nestes casos, você pode escrever a expressão lambda como

(lista de parâmetros digitados implicitamente) => expressão

Uma lista de parâmetros digitada implicitamente é apenas uma lista de nomes separados por vírgula, sem os tipos. Você não pode misturar e combinar parâmetros diferentes - ou a lista inteira é digitada explicitamente ou é digitada implicitamente. Além disso, se algum dos parâmetros estiver fora ou fora dos parâmetros, você será forçado a usar digitação explícita. No nosso exemplo, tudo bem, então sua expressão lambda é apenas

(texto) => texto.Comprimento

Isso está ficando bem curto agora. Não há muito mais do que você possa se livrar. Os parênteses parecem redundantes, no entanto.

<sup>4</sup> Os caminhos de código que lançam exceções não precisam retornar um valor, é claro, nem os loops infinitos detectáveis.

<sup>5</sup> Você ainda pode usar essa sintaxe para um delegado com tipo de retorno void se precisar apenas de uma instrução. Você omite o ponto e vírgula e os colchetes, basicamente.

### 9.1.5 Atalho para um único parâmetro

Quando a expressão lambda precisa apenas de um único parâmetro, e esse parâmetro pode ser digitado implicitamente, o C# 3 permite omitir os parênteses, então agora ele tem este formato:

nome do parâmetro => expressão

A forma final da sua expressão lambda é, portanto

texto => texto.Comprimento

Você pode estar se perguntando por que existem tantos casos especiais com expressões lambda – ninguém no resto da linguagem se importa se um método tem um parâmetro ou mais, por exemplo. Bem, o que parece ser um caso muito específico, na verdade acaba sendo *extremamente comum*, e a melhoria na legibilidade ao remover as teses dos parênteses da lista de parâmetros pode ser significativa quando há muitos lambdas em um pequeno trecho de código.

Vale a pena notar que você pode colocar parênteses em torno de toda a expressão lambda se quiser, assim como outras expressões. Ocasionalmente, isso ajuda na legibilidade, como quando você atribui o lambda a uma variável ou propriedade - caso contrário, os símbolos de igual podem ficar confusos, pelo menos no início. Na maioria das vezes é perfeitamente legível, sem qualquer sintaxe extra. A listagem a seguir mostra isso no contexto do nosso código original.

#### Listagem 9.3 Uma expressão lambda concisa

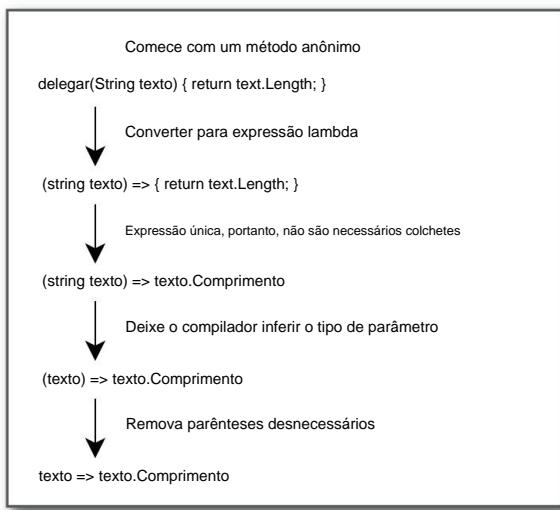
```
Func<string,int> returnLength; returnLength =  
    texto => texto.Comprimento;  
  
Console.WriteLine(returnLength("Olá"));
```

A princípio você pode achar a listagem 9.3 confusa de ler, da mesma forma que métodos anônimos parecem estranhos para muitos desenvolvedores até que eles se acostumem com eles. Em uso normal, você declararia a variável e atribuiria o valor a ela na mesma expressão, tornando-a ainda mais clara. Porém, quando você está acostumado com expressões lambda, pode perceber como elas são concisas. Seria difícil imaginar uma maneira mais curta e clara de criar uma instância delegada.<sup>6</sup> Você poderia alterar o texto do nome da variável para algo como x, e no LINQ completo isso costuma ser útil, mas nomes mais longos fornecem informações valiosas ao leitor.

Mostrei essa transformação ao longo de algumas páginas, mas a Figura 9.1 faz fica claro quanta sintaxe estranha você salvou.

A decisão de usar a forma abreviada para o corpo da expressão lambda, especificando apenas uma expressão em vez de um bloco inteiro, é completamente independente da decisão de usar parâmetros explícitos ou implícitos. Este exemplo nos levou a um caminho para encurtar o lambda, mas poderíamos ter começado por

<sup>6</sup> Isso não quer dizer que seja impossível. Algumas linguagens permitem que os encerramentos sejam representados como simples blocos de código com um nome de variável mágica para representar o caso comum de um único parâmetro.



**Figura 9.1** Atalhos de sintaxe do Lambda

tornando os parâmetros implícitos. Quando você se sentir confortável com expressões lambda, você nem pensará nisso – apenas escreverá a forma mais curta disponível naturalmente.

**FUNÇÕES DE ORDEM SUPERIOR** O próprio corpo de uma expressão lambda pode conter uma expressão lambda, e isso tende a ser tão confuso quanto parece.

Alternativamente, o parâmetro para uma expressão lambda pode ser outro delegado, o que é igualmente ruim. Ambos são exemplos de *funções de ordem superior*. Se você gosta de se sentir atordoado e confuso, dê uma olhada em alguns dos códigos-fonte para download. Embora eu esteja sendo irreverente, essa abordagem é comum na programação funcional e pode ser útil. Basta um certo grau de perseverança para entrar na mentalidade certa.

Até agora lidamos apenas com uma única expressão lambda, colocando-a em diferentes formas. Vejamos alguns exemplos para tornar as coisas mais concretas antes de examinarmos os detalhes.

## 9.2 Exemplos simples usando List<T> e eventos

Quando examinarmos os métodos de extensão no capítulo 10, usaremos expressões lambda o tempo todo. Até então, List<T> e manipuladores de eventos nos dão os melhores exemplos.

Começaremos com listas, usando propriedades implementadas automaticamente, variáveis locais digitadas implicitamente e inicializadores de coleção por uma questão de brevidade. Em seguida, chamaremos métodos que utilizam parâmetros delegados, usando expressões lambda para criar os delegados, é claro.

### 9.2.1 Filtragem, classificação e ações em listas

Lembre-se do método FindAll em List<T> — ele pega um Predicate<T> e retorna um nova lista com todos os elementos da lista original que correspondem ao predicado. O método Sort pega um Comparison<T> e classifica a lista de acordo. Finalmente, o método ForEach utiliza um Action<T> para executar em cada elemento.

A Listagem 9.4 usa expressões lambda para fornecer a instância delegada para cada um desses métodos. Os dados da amostra em questão são o nome e o ano de lançamento de vários filmes. Você imprime a lista original, depois cria e imprime uma lista filtrada apenas de filmes antigos e, em seguida, classifica e imprime a lista original ordenada por nome. (É interessante considerar quanto mais código seria necessário para fazer a mesma coisa em C# 1.)

#### Listagem 9.4 Manipulando uma lista de filmes usando expressões lambda

```

aula de cinema
{
    string pública Nome {obter; definir; } public int Ano
    { get; definir; }
}
...
var filmes = new List<Filme>
{
    novo Filme { Nome = "Tubarão", Ano = 1975 }, novo Filme { Nome
    = "Cantando na Chuva", Ano = 1952 }, novo Filme { Nome = "Some like it Hot", Ano = 1959 },
    novo Filme { Nome = "O Mágico de Oz", Ano = 1939 }, novo filme { Nome = "It's a
    Wonderful Life", Ano = 1946 }, novo filme { Nome = "American Beauty", Ano = 1999 },
    novo filme { Nome = "Alta Fidelidade", Ano = 2000 }, novo Filme { Nome = "Os Suspeitos", Ano
    = 1995 }

};

Action<Filme> print = filme =>
    Console.WriteLine("Nome={0}, Ano={1}",
        filme.Nome, filme.Ano);

filmes.ForEach(imprimir);
filmes.FindAll(filme => filme.Ano <1960)
    .ForEach(imprimir);

filmes.Sort((f1, f2) => f1.Name.CompareTo(f2.Name)); filmes.ForEach(imprimir);

```

**B** Cria delegado de impressão de lista reutilizável

**C** Imprime a lista original

**D** Cria lista filtrada

**E** Classifica a lista original

A primeira metade da listagem 9.4 envolve a configuração dos dados. Este código usa um tipo nomeado apenas para facilitar a vida - um tipo anônimo significaria mais alguns obstáculos neste caso específico.

Antes de usar a lista recém-criada, você cria uma instância delegada B, que usará para imprimir os itens da lista. Você usa essa instância delegada três vezes, e é por isso que criei uma variável para mantê-la, em vez de usar uma expressão lambda separada a cada vez. Ele apenas imprime um único elemento, mas passando-o para List<T>.ForEach você pode despejar a lista inteira no console. Um ponto sutil, mas importante, é que o ponto e vírgula no final desta instrução faz parte da instrução de atribuição, e não da expressão lambda. Se você estivesse usando a mesma expressão lambda como argumento em uma chamada de método, não haveria ponto e vírgula diretamente após Console.WriteLine(...).

A primeira lista que você imprime é apenas a original sem nenhuma modificação C. Você então encontra todos os filmes da sua lista que foram feitos antes de 1960 e os imprime D.

Isso é feito com outra expressão lambda, que é executada para cada filme da lista – basta determinar se um único filme deve ser incluído na lista filtrada. O código-fonte usa a expressão lambda como argumento do método, mas na verdade o compilador criou um método como este:

```
private static bool SomeAutoGeneratedName(filme) {
    filme de retorno.Anو < 1960;
}
```

A chamada do método para `FindAll` é efetivamente esta:

```
filmes.FindAll(novo Predicado<Filme>(SomeAutoGeneratedName))
```

O suporte à expressão lambda aqui é igual ao suporte ao método anônimo em C# 2; é tudo inteligência por parte do compilador. (Na verdade, o compilador da Microsoft é ainda mais inteligente neste caso – ele percebe que pode reutilizar a instância delegada se o código for chamado novamente, então ele o armazena em cache.)

A classificação da lista também é feita usando uma expressão lambda E, que compara dois filmes quaisquer usando seus nomes. Devo confessar que chamar explicitamente `CompareTo` para si mesmo é um pouco feio. No próximo capítulo você verá como o método de extensão `OrderBy` permite expressar pedidos de maneira mais organizada.

Vejamos outro exemplo, desta vez usando expressões lambda com manipulação de eventos.

### 9.2.2 Registrando um manipulador de eventos

Se você pensar no capítulo 5, na seção 5.9 você viu uma maneira fácil de usar métodos anônimos para registrar quais eventos estavam ocorrendo, mas você só poderia usar uma sintaxe compacta porque não se importava em perder as informações dos parâmetros. E se você quisesse registrar a natureza do evento e informações sobre seu remetente e argumentos?

As expressões lambda permitem isso de maneira simples, conforme mostrado na listagem a seguir.

#### Listagem 9.5 Registrando eventos usando expressões lambda

```
static void Log(string título, remetente do objeto, EventArgs e) {
    Console.WriteLine("Evento: {0}", título); Console.WriteLine("Remetente:
{0}", remetente); Console.WriteLine(" Argumentos: {0}", e.GetType());
    foreach (propPropertyDescriptor em TypeDescriptor.GetProperties(e))

    {
        string nome = prop.DisplayName; valor do objeto =
        prop.GetValue(e); Console.WriteLine(" {0}={1}", nome,
        valor);
    }
}
```

```

...
Botão botão = novo botão {Texto = "Clique em mim"}; botão.Clique
    += (src, e) => Log("Clique", src, e);
button.KeyPress += (src, e) => Log("KeyPress", src, e); button.MouseClick += (src, e) => Log("MouseClick",
src, e);

Formulário formulário = novo Formulário {AutoSize = true, Controls = {botão}}; Aplicativo.Run(formulário);

```

A Listagem 9.5 usa expressões lambda para passar o nome e os *parâmetros* do evento para o método Log , que registra detalhes do evento. Você não registra os detalhes da origem do evento, além do que sua substituição ToString retorna, porque uma enorme quantidade de informações está associada aos controles. Mas você usa a reflexão sobre descritores de propriedade para mostrar os detalhes da instância EventArgs passada para você.

Aqui está um exemplo de saída quando você clica no botão:

```

Evento: Clique
Remetente: System.Windows.Forms.Button, Texto: Clique em mim Argumentos:
System.Windows.Forms.MouseEventHandler
    Botão = Esquerda
    Cliques=1
    X=53
    S=17
    Delta=0
    Localização=(X=53,Y=17)
Evento: MouseClick
Remetente: System.Windows.Forms.Button, Texto: Clique em mim Argumentos:
System.Windows.Forms.MouseEventHandler
    Botão = Esquerda
    Cliques=1
    X=53
    S=17
    Delta=0
    Localização=(X=53,Y=17)

```

Tudo isso é possível sem expressões lambda, é claro, mas é muito mais simples do que seria de outra forma.

Agora que você viu lambdas sendo convertidos em instâncias delegadas, é hora de examinar as árvores de expressão, que representam expressões lambda como dados em vez de código.

### 9.3 Árvores de expressão

A ideia de *código como dados* é antiga, mas não tem sido muito usada em linguagens de programação populares. Você poderia argumentar que todos os programas .NET usam o conceito, porque o código IL é tratado como dados pelo JIT, que então o converte em código nativo para ser executado em sua CPU. Porém, isso está profundamente oculto e, embora existam bibliotecas que manipulam IL programaticamente, elas não são amplamente utilizadas.

As árvores de expressão no .NET 3.5 fornecem uma maneira abstrata de representar algum código como uma árvore de objetos. É como o CodeDOM, mas operando em um nível um pouco superior. O principal uso de árvores de expressão está no LINQ e, posteriormente nesta seção, você verá como as árvores de expressão são cruciais para toda a história do LINQ .

C# 3 fornece suporte integrado para conversão de expressões lambda em árvores de expressão, mas antes de abordarmos isso, vamos explorar como elas se encaixam no .NET Framework sem usar nenhum truque de compilador.

### 9.3.1 Construindo árvores de expressão programaticamente

As árvores de expressão não são tão místicas quanto parecem, embora alguns de seus usos pareçam mágicos. Como o nome sugere, são árvores de objetos, onde cada nó da árvore é uma expressão em si. Diferentes tipos de expressões representam as diferentes operações que podem ser executadas no código: operações binárias, como adição; operações unárias, como calcular o comprimento de um array; chamadas de métodos; chamadas de construtor; e assim por diante.

O namespace `System.Linq.Expressions` contém as diversas classes que representam expressões. Todos eles derivam da classe `Expression`, que é abstrata e consiste principalmente em métodos de fábrica estáticos para criar instâncias de outras classes de expressão. No entanto, ele expõe duas propriedades:

- ÿ A propriedade `Type` representa o tipo .NET da expressão avaliada – você pode pensar nela como um tipo de retorno. O tipo de expressão que busca a propriedade `Length` de uma string seria `int`, por exemplo.
- ÿ A propriedade `NodeType` retorna o tipo de expressão representada como um membro da enumeração `ExpressionType`, com valores como `LessThan`, `Multiply` e `Invoke`. Para usar o mesmo exemplo, em `myString.Length` a parte de acesso à propriedade teria um tipo de nó `MemberAccess`.

Existem muitas classes derivadas de `Expression` e algumas delas podem ter muitos tipos de nós diferentes. `BinaryExpression`, por exemplo, representa qualquer operação com dois operandos: aritmética, lógica, comparações, indexação de array e assim por diante. É aqui que a propriedade `NodeType` é importante, pois distingue entre diferentes tipos de expressões representadas pela mesma classe.

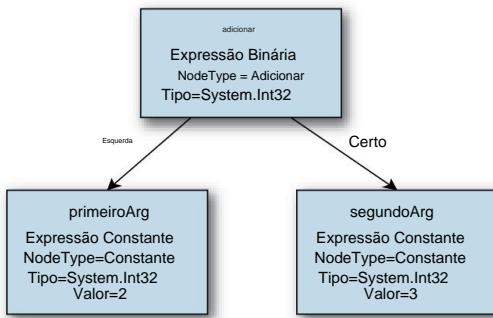
Não pretendo cobrir todas as classes de expressão ou tipos de nós - há muitos, e o MSDN faz um ótimo trabalho ao explicá-los (consulte <http://mng.bz/3vW3>).

Em vez disso, tentaremos ter uma ideia geral do que você pode fazer com árvores de expressão.

Vamos começar criando uma das árvores de expressão mais simples possíveis, somando dois números inteiros constantes. A listagem a seguir cria uma árvore de expressão para representar  $2+3$ .

#### Listagem 9.6 Uma árvore de expressão simples, somando 2 e 3

```
Expressão primeiroArg = Expression.Constant(2); Expressão segundoArg  
= Expression.Constant(3); Expressão add = Expression.Add(firstArg,  
secondArg);  
  
Console.WriteLine(adicionar);
```



**Figura 9.2**  
Representação gráfica da árvore de expressões criada pela Listagem 9.6

A execução da listagem 9.6 produzirá a saída  $(2 + 3)$ , que demonstra que os vários classes de expressão substituem `ToString` para produzir saída legível por humanos. Figura 9.2 descreve a árvore gerada pelo código.

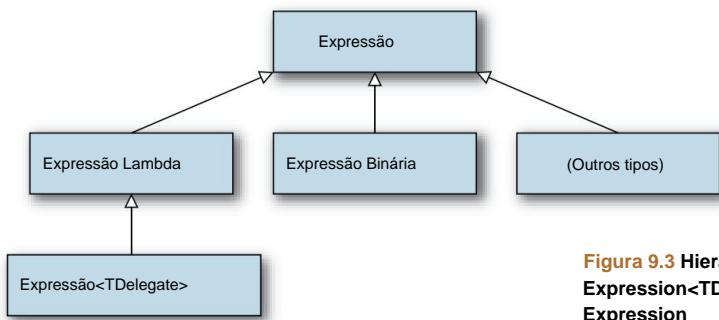
Vale a pena notar que as expressões folha são criadas primeiro no código: você constrói expressões de baixo para cima. Isto é reforçado pelo fato de que as expressões são imutável – depois de criar uma expressão, ela nunca mudará, então você pode armazenar em cache e reutilizar expressões à vontade.

Agora que você construiu uma árvore de expressão, é hora de executá-la.

### 9.3.2 Compilando árvores de expressão em delegados

Um dos tipos derivados de `Expression` é `LambdaExpression`. A classe genérica `Expression<TDelegate>` então deriva de `LambdaExpression`. É tudo um pouco confuso – a figura 9.3 mostra a hierarquia de tipos para deixar as coisas mais claras.

A diferença entre `Expression` e `Expression<TDelegate>` é que o classe genérica é digitada estaticamente para indicar que tipo de expressão é, em termos de tipo de retorno e parâmetros. Obviamente, isso é expresso pelo parâmetro de tipo `TDelegate`, que deve ser do tipo delegado. Por exemplo, a expressão de adição simples leva sem parâmetros e retorna um número inteiro - isso é correspondido pela assinatura de `Func<int>`, então você poderia usar um `Expression<Func<int>>` para representar a expressão de forma estática maneira digitada. Você faz isso usando o método `Expression.Lambda`, que possui diversas sobrecargas. Os exemplos que vimos usam o método genérico, que usa um



**Figura 9.3** Hierarquia de tipos de `Expression<TDelegate>` até `Expression`

parâmetro type para indicar o tipo de delegado que queríamos representar. Consulte o MSDN para alternativas.

Então, qual é o sentido de fazer isso? Bem, LambdaExpression possui um método Compile que cria um delegado do tipo apropriado; Expression<TDelegate> possui outro método com o mesmo nome, mas digitado estaticamente para retornar um delegado do tipo TDelegate.

Este delegado agora pode ser executado da maneira normal, como se tivesse sido criado usando um método normal ou qualquer outro meio. A listagem a seguir mostra isso em ação, com a mesma expressão de antes.

#### Listagem 9.7 Compilando e executando uma árvore de expressão

```
Expressão primeiroArg = Expression.Constant(2); Expressão  
segundoArg = Expression.Constant(3); Expressão add =  
Expression.Add(firstArg, secondArg);  
  
Func<int> compilado = Expression.Lambda<Func<int>>(add).Compile(); Console.WriteLine(compilado());
```

Indiscutivelmente, a listagem 9.7 é uma das maneiras mais complicadas de imprimir 5 que você poderia solicitar. Ao mesmo tempo, também é bastante impressionante. Você está criando programaticamente alguns blocos lógicos e representando-os como objetos normais, e então pedindo ao framework para compilar tudo em código real que pode ser executado.

Talvez você nunca precise usar árvores de expressão dessa maneira, ou mesmo construí-las programaticamente, mas são informações básicas úteis que ajudarão você a entender como o LINQ funciona.

Como eu disse no início desta seção, as árvores de expressão não estão muito distantes do CodeDOM — o Snippy compila e executa código C# que foi inserido como texto simples, por exemplo. Mas existem duas diferenças significativas entre o CodeDOM e as árvores de expressão.

Primeiro, no .NET 3.5, as árvores de expressão só podiam representar expressões únicas. Eles não foram projetados para classes inteiras, métodos ou mesmo apenas instruções. Isso mudou um pouco no .NET 4, onde eles são usados para oferecer suporte à digitação dinâmica — agora você pode criar blocos, atribuir valores a variáveis e assim por diante. Mas ainda existem restrições significativas em comparação com o CodeDOM.

Segundo, C# oferece suporte a árvores de expressão diretamente na linguagem, por meio de lambda expressões. Vamos dar uma olhada nisso agora.

### 9.3.3 Convertendo expressões lambda C# em árvores de expressão

Como você já viu, as expressões lambda podem ser convertidas em instâncias de delegação apropriadas, implícita ou explicitamente. Essa não é a única conversão disponível.

Você também pode pedir ao compilador para construir uma árvore de expressão a partir de sua expressão lambda, criando uma instância de Expression<TDelegate> em tempo de execução. Por exemplo, a listagem a seguir mostra uma maneira muito mais curta de criar a expressão “return 5”, compilá-la e, em seguida, invocar o delegado resultante.

**Listagem 9.8 Usando expressões lambda para criar árvores de expressão**

```
Expressão<Func<int>> return5 = () => 5; Func<int> compilado
= return5.Compile(); Console.WriteLine(compilado());
```

Na primeira linha da listagem 9.8, a parte () => 5 é a expressão lambda. Você não precisa de nenhuma conversão porque o compilador pode verificar tudo no momento. Você poderia ter escrito dez 2+3 em vez de 5, mas o compilador teria otimizado a adição para você.

O ponto importante a ser retirado é que a expressão lambda foi convertida em uma árvore de expressão.

**EXISTEM LIMITAÇÕES** Nem todas as expressões lambda podem ser convertidas em árvores de expressão. Você não pode converter um lambda com um bloco de instruções (mesmo que seja apenas uma instrução return) em uma árvore de expressões — ela deve estar no formato que avalia uma única expressão, e essa expressão não pode conter atribuições. Essa restrição se aplica até mesmo no .NET 4 com suas capacidades estendidas para árvores de expressão. Embora essas sejam as restrições mais comuns, elas não são as únicas — não vale a pena descrever a lista completa aqui, pois esse problema surge muito raramente. Se houver algum problema com uma tentativa de conversão, você descobrirá em tempo de compilação.

Vejamos um exemplo mais complicado para ver como as coisas funcionam, principalmente no que diz respeito aos parâmetros. Desta vez você escreverá um predicado que pega duas strings e verifica se a primeira começa com a segunda. O código é simples quando escrito como uma expressão lambda.

**Listagem 9.9 Demonstração de uma árvore de expressão mais complicada**

```
Expressão<Func<string, string, bool>> expressão =
    (x, y) => x.StartsWith(y);

var compilado = expressão.Compile();

Console.WriteLine(compilado("Primeiro", "Segundo"));
Console.WriteLine(compilado("Primeiro", "Fir"));
```

A árvore de expressão em si é mais complicada, especialmente quando você a converte em uma instância de LambdaExpression. A próxima listagem mostra como isso pode ser construído em código.

**Listagem 9.10 Construindo uma árvore de expressão de chamada de método no código**

```
Método MethodInfo = typeof(string).GetMethod ("StartsWith", new[]
    { typeof(string) });
var alvo = Expression.Parameter(typeof(string),
    "x");
var methodArg = Expression.Parameter(typeof(string), "y");
Expressão[] métodoArgs
= new[] { métodoArg };
```

```
Chamada de expressão = Expression.Call(target, método, métodoArgs);
```

**B** Constrói partes da chamada de método

Cria CallExpression a partir de partes



```
var lambdaParameters = new[] { alvo, métodoArg }; var lambda =
Expression.Lambda<Func<string, string, bool>>
(chamar, lambdaParameters);

var compilado = lambda.Compile();

Console.WriteLine(compilado("Primeiro", "Segundo"));
Console.WriteLine(compilado("Primeiro", "Fir"));
```

Converte chamada em  
LambdaExpressão D

Como você pode ver, a listagem 9.10 é consideravelmente mais envolvente do que a versão com a expressão lambda C#. Mas torna mais óbvio exatamente o que está envolvido na árvore e como os parâmetros são vinculados.

Você começa descobrindo tudo o que precisa saber sobre a chamada de método que forma o corpo da expressão final B: o destino do método (a string na qual você está chamando StartsWith ); o próprio método (como MethodInfo); e a lista de argumentos (neste caso, apenas um). Acontece que o alvo e o argumento do método serão parâmetros passados para a expressão, mas poderiam ser outros tipos de expressões — constantes, resultados de outras chamadas de métodos, avaliações de propriedades e assim por diante.

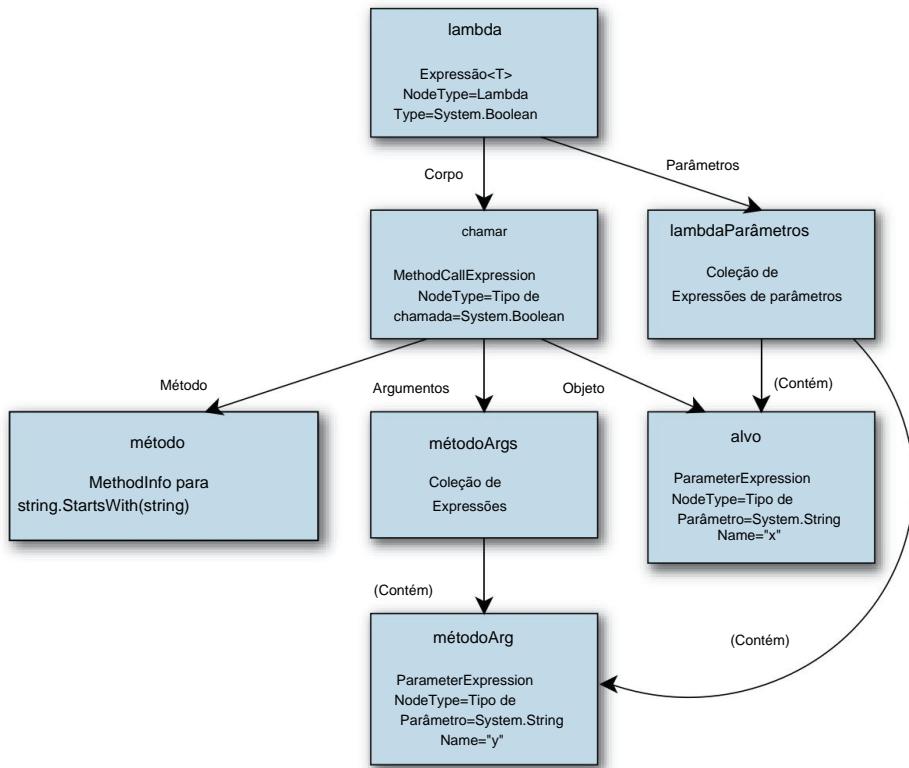
Depois de construir a chamada de método como uma expressão C, você precisa convertê-la em uma expressão lambda D, vinculando os parâmetros conforme avança. Você reutiliza os mesmos valores ParameterExpression criados como informações para a chamada de método: a ordem em que eles são especificados ao criar a expressão lambda é a ordem em que eles serão selecionados quando você eventualmente chamar o delegado.

A Figura 9.4 mostra graficamente a mesma árvore de expressão final. Para ser exigente, mesmo que ainda seja chamada de árvore de expressão, o fato de você reutilizar as expressões de parâmetro (e você precisa fazê-lo - criar uma nova com o mesmo nome e tentar vincular parâmetros dessa forma causa uma exceção no tempo de execução) significa que não é realmente uma árvore no sentido mais puro.

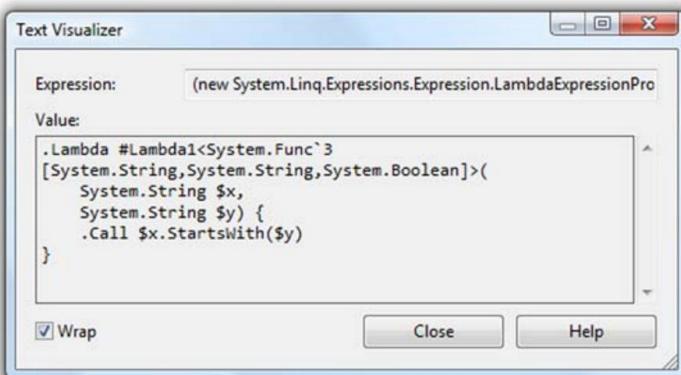
Olhando para a complexidade da figura 9.4 e da listagem 9.10 sem tentar olhar os detalhes, você seria perdoado por pensar que estava fazendo algo realmente complicado, quando na verdade é apenas uma única chamada de método. Imagine como seria a árvore de expressão para uma expressão genuinamente complexa — e então fique grato porque o C# 3 pode criar árvores de expressão a partir de expressões lambda!

Para uma última maneira de analisar a mesma ideia, o Visual Studio 2010 e 2012 fornecem um visualizador integrado para árvores de expressão.<sup>7</sup> Isso pode ser útil se você estiver tentando descobrir como construir uma árvore de expressão no código e você quer ter uma ideia de como deveria ser; escreva uma expressão lambda que faça o que você deseja com alguns dados fictícios, observe a visualização no depurador e, em seguida, descubra como construir árvores semelhantes com as informações que você possui em seu código real. O visualizador depende de alterações no .NET 4, portanto não funcionará com projetos direcionados ao .NET 3.5. A Figura 9.5 mostra a visualização do exemplo StartsWith .

<sup>7</sup> Se você estiver usando o Visual Studio 2008, poderá baixar alguns códigos de amostra do MSDN para criar um visualizador semelhante (consulte <http://mng.bz/g6xd>), mas obviamente é mais fácil usar aquele fornecido com o Visual Studio se você tenha uma versão recente o suficiente.



**Figura 9.4** Representação gráfica de uma árvore de expressão que chama um método e usa parâmetros de uma expressão lambda



**Figura 9.5** Visualização do depurador de uma árvore de expressão

As partes .Lambda e .Call da visualização correspondem às suas chamadas para Expression.Lambda e Expression.Call; \$x e \$y correspondem às expressões de parâmetro. A visualização é a mesma, quer a árvore de expressão tenha sido construída explicitamente por meio de código ou usando uma conversão de expressão lambda.

Um pequeno ponto a ser observado é que embora o compilador C# construa árvores de expressão no código compilado usando código semelhante à listagem 9.10, ele tem um atalho na manga: ele não precisa usar reflexão normal para obter o MethodInfo para string.StartsWith .

Em vez disso, ele usa o método equivalente ao operador typeof . Isso está disponível apenas em IL, não no próprio C#, e o mesmo operador é usado para criar instâncias delegadas a partir de grupos de métodos.

Agora que você viu como as árvores de expressão e as expressões lambda estão vinculadas, vamos dar uma breve olhada em por que elas são tão úteis.

#### 9.3.4 Árvores de expressão no coração do LINQ

Sem expressões lambda, as árvores de expressão teriam relativamente pouco valor.

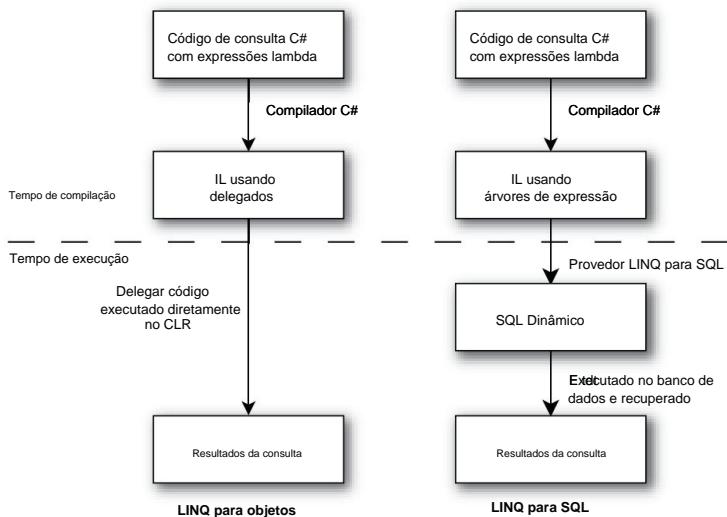
Eles seriam uma alternativa ao CodeDOM nos casos em que você desejasse modelar apenas uma única expressão em vez de instruções, métodos, tipos inteiros e assim por diante, mas o benefício ainda seria limitado.

O inverso também é verdadeiro até certo ponto: sem árvores de expressão, as expressões lambda certamente seriam *menos* úteis. Ter uma forma mais compacta de criar instâncias delegadas ainda seria bem-vinda, e a mudança para um modo de desenvolvimento mais funcional ainda seria viável. As expressões lambda são particularmente eficazes quando combinadas com métodos de extensão, como você verá no próximo capítulo, mas também com árvores de expressão na imagem, as coisas ficam muito mais interessantes.

O que você ganha quando combina expressões lambda, árvores de expressão e métodos de extensão? A resposta é basicamente “o lado da linguagem do LINQ” . A sintaxe extra que você verá no capítulo 11 é a cereja do bolo, mas a história ainda teria sido convincente apenas com esses três ingredientes. Por um longo tempo, você poderia ter *uma* boa verificação em tempo de compilação ou a capacidade de dizer a outra plataforma para executar algum código, geralmente expresso como texto (as consultas SQL são o exemplo mais óbvio). Mas você não poderia fazer as duas coisas ao mesmo tempo.

Ao combinar expressões lambda que fornecem verificações em tempo de compilação com árvores de expressão que abstraem o modelo de execução da lógica desejada, você pode ter o melhor dos dois mundos, dentro do razoável. No cerne dos provedores LINQ fora do processo está a ideia de que você pode produzir uma árvore de expressão a partir de uma linguagem de origem familiar (C#, neste caso) e usar o resultado como um formato intermediário que pode então ser convertido para a linguagem nativa da plataforma de destino – SQL, por exemplo. Em alguns casos, pode não haver tanto uma linguagem nativa simples quanto uma API nativa, fazendo diferentes chamadas de serviço web dependendo do que a expressão representa, talvez. A Figura 9.6 mostra os diferentes caminhos do LINQ to Objects e do LINQ to SQL.

Em alguns casos, a conversão pode tentar executar *toda* a lógica na plataforma de destino, enquanto outros casos podem usar os recursos de compilação das árvores de expressão para



**Figura 9.6** Tanto o LINQ to Objects quanto o LINQ to SQL começam com código C# e terminam com resultados de consulta. A capacidade de executar o código remotamente vem através de árvores de expressão.

execute algumas das expressões localmente e outras em outro lugar. Veremos alguns detalhes dessa etapa de conversão no capítulo 12, mas você deve ter esse objetivo final em mente enquanto exploramos os métodos de extensão e a sintaxe LINQ nos capítulos 10 e 11.

**NEM TODA A VERIFICAÇÃO PODE SER FEITA PELO COMPILADOR** Quando as árvores de expressão são examinadas por algum tipo de conversor, alguns casos geralmente precisam ser rejeitados. Por exemplo, embora seja possível converter uma chamada para `string.StartsWith` em uma expressão SQL semelhante, uma chamada para `string.IsInterned` não faz sentido em um ambiente de banco de dados. As árvores de expressão permitem uma grande segurança em tempo de compilação, mas o compilador só pode verificar se a expressão lambda pode ser convertida em uma árvore de expressão válida; não pode ter certeza de que a árvore de expressão será adequada para seu eventual uso.

Embora os usos mais comuns de árvores de expressão estejam relacionados ao LINQ, nem sempre é esse o caso...

### 9.3.5 Árvores de expressão além do LINQ

Bjarne Stroustrup disse uma vez: “Eu não gostaria de construir uma ferramenta que só pudesse fazer o que eu fui capaz de imaginar”. Embora as árvores de expressão tenham sido introduzidas no .NET principalmente para LINQ, tanto a comunidade quanto a Microsoft encontraram outros usos para elas desde então. Esta seção está longe de ser abrangente, mas pode lhe dar algumas idéias de onde as árvores de expressão podem ajudá-lo.

**OTIMIZANDO O TEMPO DE EXECUÇÃO DA LINGUAGEM DINÂMICA**

Veremos muito mais sobre Dynamic Language Runtime (DLR) no capítulo 14, quando falaremos sobre tipagem dinâmica em C#, mas as árvores de expressão são uma parte essencial da arquitetura. Eles possuem três propriedades que os tornam atraentes para o DLR:

- ÿ Eles são imutáveis, então você pode armazená-los em cache com segurança.
- ÿ Eles são combináveis, então você pode criar comportamentos complexos a partir de construções simples blocos.
- ÿ Eles podem ser compilados em delegados que são compilados JIT em código nativo como normal.

O DLR tem que tomar decisões sobre como lidar com diversas expressões onde o significado pode mudar sutilmente com base em regras diferentes. As árvores de expressão permitem essas regras (e os resultados) sejam transformados em código próximo ao que você escreveria à mão se você conhecesse todas as regras e resultados que viu até agora. É um conceito poderoso e um que permite que o código dinâmico seja executado com uma rapidez surpreendente.

**REFERÊNCIAS À PROVA DE REFATOR PARA MEMBROS**

Na seção 9.3.3 mencionei que o compilador pode emitir referências a valores MethodInfo de forma muito semelhante ao operador typeof . Infelizmente, C# não tem a mesma capacidade, o que significa que a única maneira de contar uma parte de propósito geral e baseada em reflexão código para “usar a propriedade chamada BirthDate definida no meu tipo” foi anteriormente para use uma string literal e certifique-se de que, se alterar o nome da propriedade, você também mude o literal. Usando C# 3, você pode construir uma árvore de expressão representando uma referência de propriedade usando uma expressão lambda. O método pode então dissecar a árvore de expressão, descobrir a propriedade que você quer dizer e fazer o que quiser com a propriedade. Informação. Também pode compilar a árvore de expressão em um delegado e usá-la diretamente, claro.

Como exemplo de como isso pode ser usado, você poderia escrever isto:

```
serializationContext.AddProperty(x => x.BirthDate);
```

O contexto de serialização saberia então que você deseja serializar o BirthDate propriedade, e poderia registrar metadados apropriados e recuperar o valor. (Serialização é apenas uma área onde você pode querer uma referência de propriedade ou método; é bastante comum em código orientado por reflexão.) Se você refatorar a propriedade BirthDate para chamar for DateOfBirth, a expressão lambda também mudará. Claro, não é infalível—não há verificação em tempo de compilação de que a expressão realmente avalia uma propriedade simples; isso deve ser uma verificação em tempo de execução no código AddProperty .

É possível que um dia o C# ganhe a capacidade de fazer isso dentro da linguagem em si. Tal operador já foi nomeado: infoof, pronunciado como “info-of” ou “in-foof”, dependendo do seu nível de despreocupação. Isso está no C# lista de possíveis recursos da equipe por um tempo e, sem surpresa, Eric Lippert blogou sobre isso (veja <http://mng.bz/24y7>), mas ainda não foi aprovado. Talvez em C# 6.

### REFLEXÃO MAIS SIMPLES

O uso final que quero mencionar antes de nos aprofundarmos nas profundezas obscuras da inferência de tipos também diz respeito à reflexão. Como mencionei no capítulo 3, os operadores aritméticos não funciona bem com genéricos, o que torna difícil escrever código genérico para (digamos) somar um série de valores. Marc Gravell usou árvores de expressão com grande efeito para fornecer um genérico Classe de operador e uma classe auxiliar não genérica, permitindo escrever código como este:

```
T runningTotal = valorinicial;
foreach (item T em valores)
{
    runningTotal = Operador.Add(runningTotal, item);
}
```

Isso funcionará até mesmo nos casos em que os valores são de um tipo diferente do valor em execução. total — adicionando uma sequência inteira de valores TimeSpan a um DateTime, por exemplo. Isso é É possível fazer isso em C# 2, mas é significativamente mais complicado devido às maneiras como os operadores são expostos por meio de reflexão, especialmente para os tipos primitivos. Árvores de expressão permitir que a implementação desta magia seja bastante limpa, e o fato de serem compilado em IL normal, que é então compilado em JIT, oferece ótimo desempenho.

Estes são apenas alguns exemplos, e sem dúvida há muitos desenvolvedores ocupados trabalhando em usos completamente diferentes. Mas eles marcam o fim da nossa cobertura direta de expressões lambda e árvores de expressão. Você verá muito mais deles quando olhamos para o LINQ, mas antes de prosseguirmos, há algumas alterações no C# que precisam alguma explicação. Estas são mudanças na inferência de tipo e como o compilador seleciona entre métodos sobrecarregados.

## 9.4 Mudanças na inferência de tipo e resolução de sobrecarga

As etapas envolvidas na inferência de tipo e resolução de sobrecarga foram alteradas em C# 3 para acomodar expressões lambda e tornar os métodos anônimos mais úteis.

Isso pode não ser considerado um novo recurso do C#, mas pode ser importante entender o que o compilador fará. Se você achar detalhes como esse tediosos e irrelevantes, sinta-se à vontade para pular para o resumo do capítulo, mas lembre-se de que esta seção existe, para que você possa lê-lo se encontrar um erro de compilação relacionado a este tópico e não conseguir entender por que seu código não funciona. (Alternativamente, você pode querer voltar para esta seção se você achar que seu código compila, mas você acha que não deveria!)

Mesmo nesta seção, não vou entrar em todos os cantos e recantos – é isso que o a especificação da linguagem é para; os detalhes estão na especificação C# 5, seção 7.5.2 (“Inferência de tipo”). Em vez disso, darei uma visão geral do novo comportamento, fornecendo exemplos de casos comuns. A principal razão para alterar a especificação é permitir expressões lambda para trabalhar de forma concisa, e é por isso que inclui o tópico neste capítulo específico.

Vamos primeiro examinar um pouco mais profundamente quais problemas você enfrentaria se a equipe C# tinha aderido às regras antigas.

### 9.4.1 Razões para mudança: simplificando chamadas de métodos genéricos

A inferência de tipo ocorre em algumas situações. Você já viu isso se aplicar a matrizes digitadas implicitamente e também é necessário quando você tenta converter implicitamente um grupo de métodos em um tipo delegado. Isso pode ser particularmente confuso quando a conversão ocorre quando você usa um grupo de métodos como argumento para outro método. Com o excesso de carga do método que está sendo chamado, a sobrecarga de métodos dentro do grupo de métodos e a possibilidade de envolvimento de métodos genéricos, o conjunto de conversões potenciais pode ser enorme.

De longe, a situação mais comum para inferência de tipos é quando você chama um método genérico sem especificar nenhum argumento de tipo. Isso acontece o tempo todo no LINQ — a maneira como as expressões de consulta funcionam depende muito dessa capacidade. Tudo é tratado tão bem que é fácil ignorar o quanto o compilador precisa trabalhar em seu nome, tudo para tornar seu código mais claro e conciso.

As regras eram *razoavelmente* diretas no C# 2, embora grupos de métodos e métodos anônimos nem sempre fossem tratados tão bem quanto você gostaria. O processo de inferência de tipos não deduziu nenhuma informação deles, levando a situações em que o comportamento desejado era óbvio para os desenvolvedores, mas não para o compilador. A vida é mais complicada em C# 3 devido às expressões lambda. Se você chamar um método genérico usando uma expressão lambda com uma lista de parâmetros digitada implicitamente, o compilador precisará descobrir de quais tipos você está falando antes de poder verificar o corpo da expressão lambda.

Isso é muito mais fácil de ver em código do que em palavras. A listagem a seguir dá um exemplo do tipo de problema ao qual estou me referindo: chamar um método genérico usando uma expressão lambda.

#### Listagem 9.11 Exemplo de código que requer as novas regras de inferência de tipo

```
estático vazio PrintConvertedValue<TInput,TOutput>
    (entrada TInput, conversor<TInput,TOutput> conversor)
{
    Console.WriteLine(conversor(entrada));
}
...
PrintConvertedValue("Sou uma string", x => x.Length);
```

O método `PrintConvertedValue` na listagem 9.11 simplesmente recebe um valor de entrada e um delegado que pode converter esse valor em um tipo diferente. É completamente genérico – não faz suposições sobre os parâmetros de tipo `TInput` e `TOutput`. Agora, observe os tipos de argumentos com os quais você está chamando na linha inferior da listagem. O primeiro argumento é claramente uma string, mas o segundo? É uma expressão lambda, então você precisa convertê-la em `Converter<TInput,TOutput>`, e isso significa que você precisa conhecer os tipos de `TInput` e `TOutput`.

Lembre-se da seção 3.3.2 que as regras de inferência de tipos do C# 2 foram aplicadas a cada argumento individualmente, sem nenhuma maneira de usar os tipos inferidos de um argumento para outro. Nesse caso, essas regras teriam impedido você de encontrar os tipos

de TInput e TOutput para o segundo argumento, então o código na listagem 9.11 não teria sido compilado.

Nosso objetivo final é entender o que faz a listagem 9.11 ser compilada em C# 3 (e ela compila, eu prometo), mas começaremos com algo mais modesto.

#### 9.4.2 Tipos de retorno inferidos de funções anônimas

A listagem a seguir mostra outro exemplo de algum código que parece que deveria ser compilado, mas não o faz de acordo com as regras de inferência de tipo do C# 2.

##### Listagem 9.12 Tentando inferir o tipo de retorno de um método anônimo

```
delegado T MyFunc<T>();
```

← Declara o tipo delegado: Func<T> não está no .NET 2.0

```
static void WriteResult<T>(função MyFunc<T>) {
```

← Declara método genérico com parâmetro delegado

```
    Console.WriteLine(função());
```

...

```
}
```

...

```
WriteResult(delegado { return 5; });
```

← Requer inferência de tipo para T

Compilar a listagem 9.12 em C# 2 gera um erro como este:

```
erro CS0411: Os argumentos de tipo para o método
'Snippet.WriteResult<T>(Snippet.MyFunc<T>)' não podem ser inferidos do uso. Tente especificar os argumentos de
tipo explicitamente.
```

Você pode corrigir o erro de duas maneiras: especificando explicitamente o argumento de tipo (conforme sugerido pelo compilador) ou convertendo o método anônimo em um tipo delegado concreto:

```
WriteResult<int>(delegado { return 5; });
WriteResult((MyFunc<int>)delegate { return 5; });
```

Ambos funcionam, mas são feios. Talvez você *queira que* o compilador execute o mesmo tipo de inferência de tipo que para tipos não delegados, usando o tipo da expressão retornada para inferir o tipo de T. Isso é exatamente o que o C# 3 faz tanto para métodos anônimos quanto para expressões lambda, mas há uma pegar. Embora em muitos casos apenas uma instrução return esteja envolvida, às vezes pode haver mais.

A listagem a seguir é uma versão ligeiramente modificada da listagem 9.12, onde o anonymous method mous às vezes retorna um número inteiro e às vezes retorna um objeto.

##### Listagem 9.13 Código retornando um número inteiro ou um objeto dependendo da hora do dia

```
delegado T MyFunc<T>();
```

```
static void WriteResult<T>(função MyFunc<T>) {
```

```
    Console.WriteLine(função());
```

}

...

```
WriteResult(delegado {
```

```
    if (DateTime.Now.Hour <12)
```

```

{
    retornar 10;           ← O tipo de retorno é int
}
outro
{
    retornar novo objeto();   ← O tipo de retorno é objeto
}
);

```

O compilador usa a mesma lógica para determinar o tipo de retorno nesta situação como faz para arrays digitados implicitamente, conforme descrito na seção 8.4. Ele forma um conjunto de todos os tipos das instruções de retorno no corpo da função anônima<sup>8</sup> (nesse caso, int e object) e verifica se exatamente um dos tipos pode ser convertido implicitamente de todos os outros. Há uma conversão implícita de int para object (via boxing), mas não de object para int, então a inferência é bem-sucedida com object como o tipo de retorno inferido. Se não houver nenhum tipo que corresponda a esse critério (ou mais de um), nenhum tipo de retorno poderá ser inferido e você receberá um erro de compilação.

Agora você sabe como calcular o tipo *de retorno* de uma função anônima, mas o que sobre expressões lambda onde os tipos de parâmetros podem ser definidos implicitamente?

#### 9.4.3 Inferência do tipo bifásico

Os detalhes da inferência de tipo no C# 3 são *muito* mais complicados do que no C# 2. É raro que você precise fazer referência à especificação para o comportamento exato, mas se precisar, recomendo que você anote todos os parâmetros de tipo, argumentos, e assim por diante, em um pedaço de papel, e então siga a especificação passo a passo, anotando cuidadosamente cada ação necessária. Você terminará com uma planilha cheia de variáveis de tipo *fixas* e *não fixas*, com um conjunto diferente de *limites* para cada uma delas. Uma variável de tipo *fixo* é aquela cujo valor o compilador decidiu; caso contrário, não será *corrigido*. Um *limite* é uma informação sobre uma variável de tipo. Além de um monte de anotações, suspeito que você terá dor de cabeça; essas coisas não são bonitas.

Apresentarei uma maneira mais confusa de pensar sobre inferência de tipos — uma que provavelmente servirá tão bem quanto conhecer a especificação e que será muito mais fácil de entender. O fato é que, se o compilador não realizar a inferência de tipo exatamente da maneira que você deseja, é quase certo que isso resultará em um erro de compilação, em vez de um código que é compilado, mas não se comporta corretamente. Se o seu código não for compilado, tente fornecer mais informações ao compilador — é simples assim. Mas aqui está *aproximadamente* o que mudou no C# 3.

A primeira grande diferença é que os argumentos do método funcionam em equipe no C# 3. No C# 2, cada argumento era usado para tentar definir *exatamente alguns parâmetros de tipo*, e o compilador reclamaria se quaisquer dois argumentos apresentassem resultados diferentes para um parâmetro de tipo específico, mesmo que fossem compatíveis. Em C# 3, os argumentos podem

<sup>8</sup> Expressões retornadas que não possuem um tipo, como null ou outra expressão lambda, não estão incluídas neste conjunto. A sua validade é verificada posteriormente, uma vez determinado o tipo de devolução, mas não contribuem para essa decisão.

contribuir com *informações* - tipos que devem ser implicitamente conversíveis no valor fixo final de uma variável de tipo específica. A lógica usada para chegar a esse valor fixo é a mesma dos tipos de retorno inferidos e matrizes digitadas implicitamente.

A listagem a seguir mostra um exemplo disso sem usar nenhuma expressão lambda sôes ou mesmo métodos anônimos.

#### Listagem 9.14 Inferência de tipo flexível combinando informações de múltiplos argumentos

```
static void PrintType<T>(T primeiro, T segundo) {  
  
    Console.WriteLine(typeof(T));  
}  
...  
PrintType(1, novo objeto());
```

Embora o código na listagem 9.14 seja *sintaticamente* válido em C# 2, ele não seria compilado; a inferência de tipo falharia, porque o primeiro parâmetro decidiria que T deve ser int e o segundo parâmetro decidiria que T deve ser objeto. Em C# 3, o compilador determina que T deve ser objeto exatamente da mesma maneira que fez para o tipo de retorno inferido na listagem 9.13. Na verdade, as regras de tipo de retorno inferido são efetivamente um exemplo do processo mais geral em C# 3.

A segunda mudança é que a inferência de tipo agora é realizada em duas fases. A primeira fase trata de argumentos normais onde os tipos envolvidos são conhecidos para começar.

Isso inclui funções anônimas em que a lista de parâmetros é digitada explicitamente.

A segunda fase então entra em ação, onde expressões lambda digitadas implicitamente e grupos de métodos têm seus tipos inferidos. A idéia é ver se alguma das informações que o compilador reuniu até agora é suficiente para descobrir os tipos de parâmetros da expressão lambda (ou grupo de métodos). Se for, o compilador poderá examinar o corpo da expressão lambda e descobrir o tipo de retorno inferido, que geralmente é outro dos parâmetros de tipo que está procurando. Se a segunda fase fornecer mais informações, o compilador passa por ela novamente, repetindo até ficar sem pistas ou ter resolvido todos os parâmetros de tipo envolvidos.

A Figura 9.7 mostra isso em forma de fluxograma, mas tenha em mente que este é um processo pesado. versão simplificada do algoritmo.

Vejamos dois exemplos para mostrar como funciona. Primeiro pegaremos o código que iniciamos a seção com – listagem 9.11:

```
estático vazio PrintConvertedValue<TInput,TOutput>  
    (entrada TInput, conversor<TInput,TOutput> conversor)  
{  
    Console.WriteLine(conversor(entrada));  
}  
...  
PrintConvertedValue("Sou uma string", x => x.Length);
```

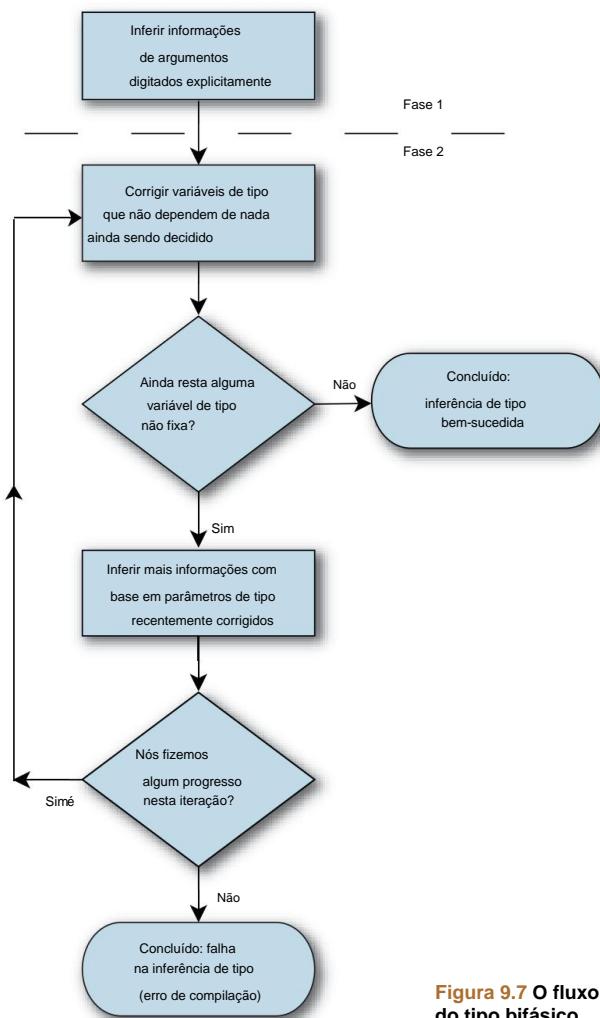


Figura 9.7 O fluxo de inferência do tipo bifásico

Os parâmetros de tipo que você precisa definir aqui são `TInput` e `TOutput`. Os passos realizados são os seguintes:

- 1 A Fase 1 começa.
- 2 O primeiro parâmetro é do tipo `TInput` e o primeiro argumento é do tipo `string`.  
Você infere que deve haver uma conversão implícita de `string` para `TInput`.
- 3 O segundo parâmetro é do tipo `Converter<TInput,TOutput>`, e o segundo argumento é uma expressão lambda digitada implicitamente. Nenhuma inferência é realizada – você não tem informações suficientes.
- 4 A Fase 2 começa.
- 5 `TInput` não depende de nenhum parâmetro de tipo não fixo, portanto é fixo em `string`.

**6** O segundo argumento agora tem um tipo de entrada fixo , mas um tipo de saída não fixo . Você pode considerá-lo como (string x) => x.Length e inferir o tipo de retorno como int . Portanto, uma conversão implícita deve ocorrer de int para TOutput.

**7** A Fase 2 se repete.

**8** TOutput não depende de nada não corrigido, portanto está corrigido para int.

**9** Agora não há parâmetros de tipo não fixos, portanto a inferência é bem-sucedida.

Complicado, né? Ainda assim, ele faz o trabalho — o resultado é o que você deseja (TInput=string, TOutput=int) e tudo é compilado sem problemas.

A importância da repetição da fase 2 é melhor demonstrada com outro exemplo.

A Listagem 9.15 mostra *duas* conversões sendo realizadas, com a saída da primeira se tornando a entrada da segunda. Até que você tenha calculado o tipo de saída da primeira conversão, você não sabe o tipo de entrada da segunda, portanto também não pode inferir seu tipo de saída.

#### Listagem 9.15 Inferência de tipo multiestágio

```
static void ConvertTwice<TInput,TMiddle,TOutput>
    (TEntrada de entrada,
     Converter<TInput,TMiddle> firstConversion,
     Converter<TMiddle,TOutput> segundaConversão)
{
    TMiddle meio = firstConversion(entrada); Saída TOutput =
        secondConversion(meio); Console.WriteLine(saída);

}
...
ConvertTwice("Outra string", text => text.Length,
            length => Math.Sqrt(length));
```

A primeira coisa a notar é que a assinatura do método parece horrível. Não é tão ruim quando você para de ter medo e apenas olha com atenção, e certamente o uso do exemplo torna isso mais óbvio. Você pega uma string e realiza uma conversão nela – a mesma conversão de antes, apenas um cálculo de comprimento. Você então pega esse comprimento (um int) e encontra sua raiz quadrada (um double).

A fase 1 da inferência de tipo informa ao compilador que deve haver uma conversão de string para TInput. Na primeira vez na fase 2, TInput é fixado em string e você infere que deve haver uma conversão de int para TMiddle. Na segunda vez na fase 2, TMiddle é fixado em int e você infere que deve haver uma conversão de double para TOutput. Na terceira vez na fase 2, TOutput é fixado em double e a inferência de tipo é bem-sucedida. Quando a inferência de tipo for concluída, o compilador poderá examinar o código dentro da expressão lambda corretamente.

**VERIFICANDO O CORPO DE UMA EXPRESSÃO LAMBDA** O corpo de uma expressão lambda *não pode ser verificado* até que os tipos de parâmetros de entrada sejam conhecidos. A expressão lambda x => x.Length é válida se x for uma matriz ou uma string, mas inválida em

muitos outros casos. Isso não é um problema quando os tipos de parâmetros são declarados explicitamente, mas com uma lista de parâmetros implícita, o compilador precisa esperar até que seja executada a inferência de tipo relevante antes de tentar descobrir o que a expressão lambda significa.

Esses exemplos mostraram apenas uma mudança funcionando por vez, mas na prática pode haver diversas informações sobre diferentes tipos de variáveis, potencialmente descobertas em diferentes iterações do processo. Em um esforço para salvar sua sanidade (e a minha), não apresentarei exemplos mais complicados — espero que você entenda o mecanismo geral, mesmo que os detalhes exatos sejam nebulosos.

Embora possa parecer que esse tipo de situação ocorrerá tão raramente que não vale a pena ter regras tão complexas para cobri-la, na verdade isso é comum em C# 3, particularmente com LINQ. Você poderia facilmente usar a inferência de tipos extensivamente sem pensar nisso - é provável que ela se torne uma segunda natureza para você. Se falhar e você se perguntar por quê, você pode sempre revisitar esta seção e a especificação da linguagem.

Precisamos abordar mais uma alteração, mas você ficará feliz em saber que é mais fácil do que a inferência de tipo. Vejamos a sobrecarga de métodos.

#### 9.4.4 Escolhendo o método sobrecarregado correto

A sobrecarga ocorre quando há vários métodos disponíveis com o mesmo nome, mas com assinaturas diferentes. Às vezes é óbvio qual método é apropriado, porque é o único com o número certo de parâmetros ou é o único onde todos os argumentos podem ser convertidos nos tipos de parâmetros correspondentes.

A parte complicada surge quando existem vários métodos que *podem* ser o correto. As regras na seção 7.5.3 da especificação (“Resolução de Sobrecarga”) são bastante complicadas (sim, *novamente*), mas a parte principal é a maneira como cada tipo de argumento é convertido no tipo de parâmetro.<sup>9</sup> Por exemplo, considere estas assinaturas de método como se ambos fossem declarados no mesmo tipo:

escrita vazia (int x) escrita  
vazia (duplo y)

O significado de uma chamada para Write(1.5) é óbvio, porque não há conversão implícita de double para int, mas uma chamada para Write(1) é mais complicada. Há uma conversão implícita de *int* para double, portanto ambos os métodos são possíveis. Nesse ponto, o compilador considera a conversão de int para int e de int para double. Uma conversão de

qualquer tipo para si mesmo é definido como *melhor do que* qualquer conversão para um tipo diferente, portanto, o método Write(int x) é melhor que Write(double y) para esta chamada específica.

Quando há vários parâmetros, o compilador precisa garantir que haja o melhor método a ser usado. Um método é melhor que outro se todas as conversões de argumentos envolvidas forem *pelo menos tão boas quanto* as conversões correspondentes no outro método, e pelo menos uma conversão for estritamente melhor. Como um exemplo simples, suponha que você tenha isto:

---

<sup>9</sup> Presumo que todos os métodos sejam declarados na mesma classe. Quando a herança também está envolvida, fica ainda *mais* complicado. Esse aspecto não mudou no C# 3.

```
void Write(int x, duplo y) void Write(duplo x,  
int y)
```

Uma chamada para `Write(1, 1)` seria ambígua e o compilador faria você adicionar uma conversão a pelo menos um dos parâmetros para deixar claro qual método você pretendia chamar. Cada sobrecarga tem uma conversão de argumento melhor, portanto nenhum deles é o melhor método.

Essa lógica ainda se aplica ao C# 3, mas com uma regra extra sobre funções anônimas, que nunca especificam um tipo de retorno. Neste caso, o tipo de retorno inferido (conforme descrito na seção 9.4.2) é utilizado nas regras de melhor conversão.

Vejamos um exemplo do tipo de situação que necessita da nova regra. A listagem a seguir contém dois métodos com o nome `Execute` e uma chamada usando uma expressão lambda.

#### Listagem 9.16 Exemplo de escolha de sobrecarga influenciada pelo tipo de retorno delegado

```
static void Execute(Func<int> ação) {  
  
    Console.WriteLine("ação retorna um int: " + ação());  
  
} static void Execute(Func<double> ação) {  
  
    Console.WriteLine("ação retorna um duplo: " + ação());  
}  
...  
  
Executar(() => 1);
```

A chamada para `Execute` na listagem 9.16 poderia ter sido escrita com um método anônimo ou um grupo de métodos — as mesmas regras são aplicadas qualquer que seja a tipo de conversão envolvida. Qual método `Execute` deve ser chamado? As regras de sobrecarga dizem que quando dois métodos são aplicáveis após a realização de conversões nos argumentos, essas conversões de argumentos são examinadas para ver qual é o melhor. As versões aqui não são de um tipo .NET normal para o tipo de parâmetro - elas são de uma expressão lambda para dois tipos de delegado diferentes. Qual conversão é melhor?

Surpreendentemente, a mesma situação em C# 2 resultaria em um erro de compilação – não havia nenhuma regra de linguagem cobrindo esse caso. Em C# 3, seria escolhido o método com o parâmetro `Func<int>`. A regra extra que foi adicionada pode ser parafraseada desta forma:

*Se uma função anônima puder ser convertida em dois tipos de delegados que tenham a mesma lista de parâmetros, mas tipos de retorno diferentes, as conversões de delegado serão avaliadas pelas conversões do tipo de retorno inferido para os tipos de retorno dos delegados.*

Isso é praticamente um jargão sem se referir a um exemplo. Vejamos a listagem 9.16, onde você está convertendo de uma expressão lambda sem parâmetros e com um tipo de retorno inferido de `int` para `Func<int>` ou `Func<double>`. As listas de parâmetros são as mesmas (vazias) para ambos os tipos de delegados, portanto a regra se aplica. Você então apenas

precisa encontrar a melhor conversão: int para int ou int para double. Isso coloca você em um território mais familiar; como você viu anteriormente, a conversão de int para int é melhor. A Listagem 9.16, portanto, imprime `action` e retorna um int: 1.

#### 9.4.5 Concluindo inferência de tipo e resolução de sobrecarga

Esta seção foi bastante pesada. Eu adoraria ter simplificado, mas é um tópico fundamentalmente complicado. A terminologia envolvida não facilita nada, especialmente porque *tipo de parâmetro* e *parâmetro de tipo* significam coisas completamente diferentes!

Parabéns se você conseguiu passar e realmente entendeu tudo. Não se preocupe se não o fez; espero que da próxima vez que você ler a seção, ela esclareça mais o assunto, principalmente depois de você se deparar com situações em que ela seja relevante para seu próprio código. No momento, aqui estão os pontos mais importantes:

- ÿ Funções anônimas (métodos anônimos e expressões lambda) têm tipos de retorno inferidos com base nos tipos de todas as instruções de retorno. ÿ Expressões lambda só podem ser compreendidas pelo compilador quando os tipos de todos os parâmetros são conhecidos.
- ÿ A inferência de tipo não exige mais que cada argumento chegue independentemente exatamente à mesma conclusão sobre parâmetros de tipo, desde que os resultados permaneçam compatíveis.
- ÿ A inferência de tipo agora é multiestágio: o tipo de retorno inferido de um anônimo função pode ser usada como um tipo de parâmetro para outra.
- ÿ Encontrar o melhor método sobrecarregado quando funções anônimas estão envolvidas leva em consideração o tipo de retorno inferido.

Mesmo essa pequena lista é bastante assustadora em termos da enorme densidade de termos técnicos. Novamente, não se preocupe se tudo não fizer sentido. Na minha experiência, as coisas funcionam do jeito que você deseja na maioria das vezes.

## 9.5 Resumo

No C# 3, as expressões lambda substituem quase inteiramente os métodos anônimos. Métodos anônimos são suportados por uma questão de compatibilidade com versões anteriores, mas o código C# 3 idiomático e recém-escrito conterá poucos deles.

Você viu como as expressões lambda são mais do que apenas uma sintaxe compacta para criação de delegados. Elas podem ser convertidas em árvores de expressão, sujeitas a algumas limitações. As árvores de expressão podem então ser processadas por outro código, possivelmente executando ações equivalentes em diferentes ambientes de execução. Sem essa capacidade, o LINQ ficaria restrito a consultas em processo.

Nossa discussão sobre inferência e sobrecarga de tipos foi um mal necessário até certo ponto; muito poucas pessoas realmente *gostam* de discutir o tipo de regras que são exigidas, mas é importante ter pelo menos uma compreensão superficial do que está acontecendo. Antes que todos sintamos pena de nós mesmos, pense nos pobres designers de linguagem que têm que viver e respirar esse tipo de coisa, certificando-se de que as regras sejam consistentes e

não desmorone em situações desagradáveis. Então tenha pena dos testadores que precisam tentar quebrar a implementação.

Isso é tudo em termos de descrição de expressões lambda, mas você verá muito mais delas no restante do livro. Por exemplo, o próximo capítulo é sobre *métodos de extensão*.

Superficialmente, eles são completamente separados das expressões lambda, mas na realidade os dois recursos são frequentemente usados juntos.

# 10

## Métodos de extensão

### Este capítulo cobre

- ÿ Escrevendo métodos de extensão
- ÿ Métodos de extensão de chamada
- ÿ Encadeamento de métodos
- ÿ Métodos de extensão no .NET 3.5
- ÿ Outros usos para métodos de extensão

Não sou fã de herança. Ou melhor, não sou fã de vários lugares onde a herança foi usada no código que mantive ou nas bibliotecas de classes com as quais trabalhei. Tal como acontece com tantas coisas, é poderoso quando usado corretamente, mas seu design muitas vezes é esquecido e pode se tornar doloroso com o tempo. Às vezes, é usado como uma forma de adicionar comportamento e funcionalidade extras a uma classe, mesmo quando nenhuma informação real sobre o objeto está sendo adicionada – onde nada está sendo especializado.

Às vezes isso é apropriado – se os objetos do novo tipo deveriam carregar os detalhes do comportamento extra – mas muitas vezes não é. Muitas vezes, em primeiro lugar, simplesmente não é possível usar herança dessa forma, como quando você está trabalhando com um tipo de valor, uma classe selada ou uma interface. A alternativa geralmente é escrever vários métodos estáticos, a maioria dos quais aceita uma instância do tipo em questão como pelo menos

um de seus parâmetros. Isso funciona bem, sem a penalidade de design da herança, mas tende a deixar o código feio.

C# 3 introduziu a ideia de *métodos de extensão*, que possuem os benefícios da solução de métodos estáticos e também melhoram a legibilidade do código que os chama. Eles permitem chamar métodos estáticos como se fossem métodos de instância de uma classe completamente diferente.

Não entre em pânico – não é tão louco ou arbitrário quanto parece.

Neste capítulo, veremos primeiro como usar métodos de extensão e como escrevê-los. Em seguida, examinaremos alguns dos métodos de extensão fornecidos pelo .NET 3.5 e veremos como eles podem ser encadeados facilmente. Essa capacidade de encadeamento é uma parte importante do motivo para a introdução de métodos de extensão na linguagem, em primeiro lugar, e é uma parte importante do LINQ. de usar métodos de extensão em vez de <sup>1</sup> Finalmente, consideraremos alguns dos prós e contras métodos estáticos simples.

Porém, primeiro vamos examinar mais de perto por que os métodos de extensão às vezes são desejáveis em comparação com o que está disponível no C# 1 e 2, especialmente quando você cria classes de utilitários.

## 10.1 Vida antes dos métodos de extensão

Você pode estar tendo uma sensação de déjà vu neste momento, porque as classes utilitárias surgiram no capítulo 7, quando examinamos as classes estáticas. Se você escreveu muito código C# 2 antes de usar C# 3, deverá examinar suas classes estáticas — muitos dos métodos nelas contidos podem ser bons candidatos para conversão em métodos de extensão. Isso não quer dizer que todas as classes estáticas existentes sejam adequadas, mas você pode muito bem reconhecer as seguintes características:

ÿ Você deseja adicionar alguns membros a um tipo. ÿ

Você não precisa adicionar mais dados às instâncias do tipo. ÿ Você não pode alterar o tipo em si, porque ele está no código de outra pessoa.

Uma pequena variação disso é onde você deseja trabalhar com uma interface em vez de uma classe, adicionando um comportamento útil enquanto chama apenas métodos na interface. Um bom exemplo disso é `IList<T>`. Não seria bom poder classificar qualquer implementação (mutável) de `IList<T>?` Seria horrível forçar cada implementação da interface a implementar a classificação, mas seria bom do ponto de vista do usuário da lista.

O problema é que `IList<T>` fornece todos os blocos de construção para uma rotina de classificação completamente genérica (vários, na verdade), mas você não pode colocar essa implementação na interface. `IList<T>` poderia ter sido especificado como uma classe abstrata, e a funcionalidade de classificação incluída dessa forma, mas como C# e .NET têm herança única de implementação, isso teria colocado uma restrição significativa nos tipos derivados de isto. Um método de extensão em `IList<T>` permitiria classificar qualquer implementação de `IList<T>`, fazendo parecer que a própria lista fornecia a funcionalidade.

<sup>1</sup> Se você está cansado de ouvir sobre quantos recursos são “uma parte importante do LINQ”, não o culpo, mas isso faz parte de sua grandeza. Existem muitas peças pequenas, mas a soma delas é muito brilhante. O fato de cada recurso poder ser usado de forma independente é um bônus adicional.

Você verá mais tarde que muitas das funcionalidades do LINQ são construídas em métodos de extensão sobre interfaces. No momento, porém, usaremos um tipo diferente para nossos exemplos: System.IO.Stream, a base da comunicação binária no .NET. O próprio Stream é uma classe abstrata com várias classes derivadas concretas, como NetworkStream, FileStream e MemoryStream.

Infelizmente, existem algumas funcionalidades que seriam úteis para incluir no Stream e que simplesmente não existem.

Os recursos ausentes que conheço com mais frequência são a capacidade de ler todo um fluxo na memória como uma matriz de bytes e a capacidade de copiar o conteúdo de um fluxo para outro.<sup>2</sup> Esses dois recursos são frequentemente mal implementados, fazendo suposições sobre fluxos que simplesmente não são válidos - o equívoco mais comum é que Stream.Read preencherá completamente o buffer se os dados não acabarem primeiro.

**AFINAL NÃO TÃO “FALTA”** Um desses recursos foi adicionado ao .NET 4: Stream agora tem um método CopyTo . Isso é útil para demonstrar um aspecto um pouco frágil dos métodos de extensão, e voltaremos a ele na seção 10.2.3. ReadFully ainda está faltando, mas deve ser usado com cuidado de qualquer maneira: você só deve tentar ler um fluxo inteiro se tiver certeza de que ele realmente tem um fim e que todos os dados cabem na memória. Os streams não têm obrigação de ter uma quantidade finita de dados.

Seria bom ter a funcionalidade em um único lugar, em vez de duplicá-la em vários projetos. É por isso que escrevi a classe StreamUtil em minha biblioteca de utilitários diversos. O código real contém uma boa quantidade de verificação de erros e outras funcionalidades, mas a listagem a seguir mostra uma versão reduzida que é mais que adequada para nossas necessidades.

#### Listagem 10.1 Uma classe utilitária simples para fornecer funcionalidade extra para streams

```
usando System.IO;

classe estática pública StreamUtil {

    const int BufferSize = 8192;

    public static void Copy(Entrada de fluxo, Saída de fluxo) {

        byte[] buffer = novo byte[BufferSize]; leitura interna; while ((ler
        =
        input.Read(buffer, 0, buffer.Length)) > 0) {

            saída.Write (buffer, 0, leitura);
        }

    } byte estático público[] ReadFully (entrada de fluxo) {

        usando (MemoryStream tempStream = novo MemoryStream())
}
```

<sup>2</sup> Devido à natureza dos fluxos, essa cópia não *duplica* necessariamente os dados — ela apenas os lê de um fluxo e os grava em outro. Embora *cópia* não seja um termo estritamente preciso nesse sentido, a diferença geralmente é irrelevante.

```

    {
        Copiar(entrada, tempStream); retornar
        tempStream.ToArray();
    }
}
}

```

Os detalhes de implementação não importam muito, embora valha a pena notar que o método ReadFully chama o método Copy — isso será útil para demonstrar mais tarde um ponto sobre métodos de extensão.

A classe é fácil de usar — a listagem a seguir mostra como você pode escrever uma web resposta ao disco, por exemplo.

#### Listagem 10.2 Usando StreamUtil para copiar um fluxo de resposta da web para um arquivo

```

Solicitação WebRequest = WebRequest.Create("http://manning.com"); usando (resposta WebResponse
= request.GetResponse()) usando (Stream responseStream =
response.GetResponseStream() usando (Saída FileStream = File.Create("response.dat")) {

    StreamUtil.Copy(responseStream, saída);
}

```

A Listagem 10.2 é bastante compacta, e a classe StreamUtil se encarregou de fazer o loop e solicitar mais dados ao fluxo de resposta até que todos tenham sido recebidos. Ele fez seu trabalho como classe utilitária de maneira perfeitamente razoável. Mesmo assim, não parece muito orientado a objetos. Seria melhor pedir ao fluxo de resposta que se copiasse para o fluxo de saída, assim como a classe MemoryStream possui um método WriteTo . Não é um grande problema, mas já é um pouco feio.

A herança não ajudaria nessa situação (você deseja que esse comportamento esteja disponível para todos os fluxos, não apenas para aqueles pelos quais você é responsável) e não pode alterar a classe Stream em si , então o que pode fazer? Com o C# 2, você estava sem opções — era preciso ficar com os métodos estáticos e conviver com a falta de jeito. C# 3 permite que você altere sua classe estática para expor seus membros como métodos de extensão, para que você possa fingir que os métodos sempre fizeram parte do Stream . Vamos ver quais mudanças são necessárias.

## 10.2 Sintaxe dos métodos de extensão

Os métodos de extensão são quase embaraçosamente fáceis de criar e também simples de usar. As considerações sobre quando e como usá-los são significativamente mais profundas do que as dificuldades envolvidas em aprender como escrevê-los.

Vamos começar convertendo a classe StreamUtil para que ela tenha alguns métodos de extensão.

### 10.2.1 Declarando métodos de extensão

Você não pode usar qualquer método como método de extensão - ele deve ter as seguintes características:

- ÿ Deve estar em uma classe estática não aninhada e não genérica (e, portanto, deve ser uma classe método estático).
- ÿ Deve ter pelo menos um parâmetro. ÿ O primeiro parâmetro deve ser prefixado com a palavra-chave this . ÿ O primeiro parâmetro não pode ter nenhum outro modificador (como out ou ref). ÿ O tipo do primeiro parâmetro não deve ser do tipo ponteiro.

É isso aí - o método pode ser genérico, retornar um valor, ter parâmetros ref/out diferentes do primeiro, ser implementado com um bloco iterador, fazer parte de uma classe parcial, usar tipos anuláveis - qualquer coisa, desde que as restrições anteriores são atendidas.

Chamaremos o tipo do primeiro parâmetro de *tipo estendido* do método e diremos que o método *estende* esse tipo — neste caso, estamos estendendo Stream. Esta não é a terminologia oficial da especificação, mas é um atalho útil.

A lista anterior não apenas fornece todas as restrições, mas também fornece detalhes do que você precisa fazer para transformar um método estático normal em uma classe estática em um método de extensão - basta adicionar a palavra-chave this . A listagem a seguir mostra a mesma classe da listagem 10.1, mas desta vez com ambos os métodos como métodos de extensão.

#### Listagem 10.3 A classe StreamUtil novamente, mas desta vez com métodos de extensão

```
classe estática pública StreamUtil
{
    const int BufferSize = 8192;

    public static void CopyTo (esta entrada de fluxo, saída de fluxo)
    {
        byte[] buffer = novo byte[BufferSize]; leitura interna; while ((ler
        =
        input.Read(buffer, 0, buffer.Length)) > 0) {

            saída.Write (buffer, 0, leitura);
        }
    }

    byte estático público [] ReadFully (esta entrada de fluxo)
    {
        usando (MemoryStream tempStream = new MemoryStream()) {

            CopyTo(entrada, tempStream); retornar
            tempStream.ToArray();
        }
    }
}
```

Sim, a única grande mudança na listagem 10.3 é a adição dos dois modificadores mostrados em negrito. Também mudei o nome do método de Copy para CopyTo. Como você verá em um minuto, isso permitirá que o código de chamada seja lido com mais naturalidade, embora pareça um pouco estranho no método ReadFully no momento.

Agora, não adianta muito ter métodos de extensão se você não pode usá-los...

### 10.2.2 Chamando métodos de extensão

Mencionei isso de passagem, mas você ainda não viu o que um método de extensão realmente faz. Simplificando, ele finge ser um método de instância de outro tipo – o tipo do primeiro parâmetro do método.

A transformação do código que usa StreamUtil é tão simples quanto a transformação da própria classe do utilitário. Desta vez, em vez de adicionar algo, vamos retirá-lo. A listagem a seguir é uma repetição da listagem 10.2, mas usando a nova sintaxe para chamar CopyTo. Digo “novo”, mas na verdade não é nada novo – é a mesma sintaxe que você sempre usou para chamar métodos de instância.

#### Listagem 10.4 Copiando um stream usando um método de extensão

```
Solicitação WebRequest = WebRequest.Create("http://manning.com"); usando (resposta WebResponse  
= request.GetResponse()) usando (Stream responseStream =  
response.GetResponseStream()) usando (Saída FileStream = File.Create("response.dat")) {  
  
    respostaStream.CopyTo(saída);  
}
```

Na listagem 10.4 pelo menos parece que você está pedindo ao fluxo de resposta para fazer a cópia. Ainda é o StreamUtil que faz o trabalho nos bastidores, mas o código é lido de maneira mais natural. Na verdade, o compilador converteu a chamada CopyTo em uma chamada de método estático normal para StreamUtil.CopyTo, passando o valor de responseStream como o primeiro argumento (seguido pela saída normalmente).

Agora que você pode ver o código em questão, espero que entenda por que mudei o nome do método de Copy para CopyTo. Alguns nomes funcionam tão bem para métodos estáticos quanto para métodos de instância, mas você descobrirá que outros precisam de ajustes para obter o benefício máximo de legibilidade.

Se quiser tornar o código StreamUtil um pouco mais agradável, você pode alterar a linha de ReadFully que chama CopyTo assim:

```
input.CopyTo(tempStream);
```

Neste ponto, a mudança de nome é totalmente apropriada para todos os usos — embora não haja nada que impeça você de usar o método de extensão como um método estático normal, o que é útil quando você está migrando muito código.

Você deve ter notado que nada nessas chamadas de método indica que você está usando um método de extensão em vez de um método de instância regular de Stream. Isso pode ser visto de duas maneiras: é uma coisa boa se o seu objetivo é fazer com que os métodos de extensão se misturem tanto quanto possível e causem pouco alarme, mas é uma coisa ruim se você quiser ver imediatamente o que realmente está acontecendo .

Se estiver usando o Visual Studio, você pode passar o mouse sobre uma chamada de método e obter uma indicação na dica de ferramenta quando for um método de extensão, conforme mostrado na Figura 10.1. O IntelliSense também indica quando está oferecendo um método de extensão, tanto no ícone do método quanto na dica de ferramenta quando ele é selecionado. Claro, você não quer ter que passar o mouse

```
Solicitação WebRequest = WebRequest.Create("http://manning.com"); usando ( resposta WebResponse
= request.GetResponse() usando (Stream responseStream =
response.GetResponseStream()0 usando (Saída FileStream = File.Create("response.dat"))

{
    respostaStream.CopyTo(saída); py ( p );
}
    (extensão) void Stream.CopyTo (saída de fluxo)
```

**Figura 10.1** Passar o mouse sobre uma chamada de método no Visual Studio revela se o método é um método de extensão.

em cada chamada de método que você faz ou tenha muito cuidado com o IntelliSense, mas na maioria das vezes não importa se você está chamando uma instância ou um método de extensão.

Ainda há uma coisa bastante estranha nesse código de chamada: ele não menciona StreamUtil em lugar nenhum! Como o compilador sabe usar o método de extensão em primeiro lugar?

### 10.2.3 Descoberta do método de extensão

É importante saber como chamar métodos de extensão, mas também é importante saber como não *chamá -los* – como evitar opções indesejadas. Para conseguir isso, você precisa saber como o compilador decide quais métodos de extensão usar.

Os métodos de extensão são disponibilizados para o código da mesma forma que as classes são disponibilizadas sem qualificação – com o uso de diretivas. Quando o compilador vê uma expressão que parece estar tentando usar um método de instância, mas nenhum dos métodos de instância é compatível com a chamada do método (se não houver nenhum método com esse nome, por exemplo, ou nenhuma sobrecarga corresponder aos argumentos fornecidos), em seguida, procura um método de extensão apropriado. Ele considera todos os métodos de extensão em todos os namespaces importados e nos namespaces atuais e corresponde àqueles em que há uma conversão implícita do tipo de expressão para o tipo estendido.

#### Detalhe da implementação: como o compilador identifica um método de extensão?

Para descobrir se deve usar um método de extensão, o compilador precisa ser capaz de diferenciar um método de extensão de outros métodos dentro de um método de extensão.

classe estática que possui uma assinatura apropriada. Isso é feito verificando se System.Runtime.CompilerServices.ExtensionAttribute foi aplicado ao método e à classe. Este atributo foi introduzido no .NET 3.5, mas o compilador não verifica de qual assembly o atributo vem. Isso significa que você ainda pode usar métodos de extensão mesmo se seu projeto for direcionado ao .NET 2.0 – você só precisa definir seu próprio atributo com o nome correto no namespace correto.

Você pode então declarar seus métodos de extensão normalmente e o atributo será aplicado automaticamente. O compilador também aplica o atributo ao assembly que contém o método de extensão, mas atualmente não exige isso ao procurar por métodos de extensão.

Introduzir suas próprias cópias de tipos de sistema pode se tornar problemático quando você mais tarde precisa usar uma versão do framework que já defina esses tipos. Se você fizer Para usar esta técnica, vale a pena usar símbolos de pré-processador para declarar o atributo apenas condicionalmente. Você pode então criar uma versão do seu código direcionada ao .NET 2.0 e outro direcionado ao .NET 3.5 e superior.

Se vários métodos de extensão aplicáveis estiverem disponíveis para diferentes tipos estendidos (usando conversões implícitas), escolhe-se o mais adequado com as melhores regras de conversão utilizadas na sobrecarga. Por exemplo, se IDerived herda de IBase, e existe um método de extensão com o mesmo nome para ambos, então o método de extensão IDerived é usado preferencialmente ao do IBase. Novamente, esse recurso é usado em LINQ, como você verá na seção 12.2, onde você conhecerá a interface `IQueryable<T>`.

É importante observar que se um método de instância aplicável estiver disponível, isso sempre ser usado antes de procurar métodos de extensão, mas o compilador não emite um aviso se um método de extensão também corresponder a um método de instância existente. Para por exemplo, o .NET 4 tem um novo método Stream que também é chamado de CopyTo. Ele tem duas sobrecargas, uma das quais entra em conflito com o método de extensão que você acabou de criar. O resultado é que o novo método é escolhido em preferência ao método de extensão, portanto, se você compilar a listagem 10.4 com o .NET 4, acabará usando Stream.CopyTo em vez de Stream Util.CopyTo. Você ainda pode chamar o método StreamUtil estaticamente usando o método normal sintaxe de StreamUtil.CopyTo(entrada, saída), mas nunca será escolhido como um método de extensão. Neste caso, não há danos ao código existente: o novo método de instância tem o mesmo significado que o seu método de extensão, então não importa qual é usado. Em outros casos, pode haver diferenças sutis na semântica que podem ser difíceis para detectar até que o código seja quebrado.

Outro problema potencial com a forma como os métodos de extensão são disponibilizados para o código é que eles são muito abrangentes. Se houver duas classes na mesma namespace contendo métodos com o mesmo tipo estendido, não há como apenas usando os métodos de extensão de uma das classes. Da mesma forma, não há como importando um namespace para disponibilizar tipos usando apenas seus simples nomes, mas sem disponibilizar os métodos de extensão dentro desse namespace em o mesmo tempo. Você pode querer usar um namespace que contenha apenas classes estáticas com métodos de extensão para mitigar este problema, a menos que o resto da funcionalidade do namespace já depende fortemente dos métodos de extensão (como é o caso de System.Linq, por exemplo).

Um aspecto dos métodos de extensão pode ser bastante surpreendente quando você encontra pela primeira vez isso, mas também é útil em algumas situações. É tudo uma questão de referências nulas – vamos dar uma olhada.

#### 10.2.4 Chamando um método em uma referência nula

Qualquer pessoa que faça uma quantidade significativa de programação .NET certamente encontrará um NullReferenceException causada pela chamada de um método através de uma variável cujo valor muda

parece ser uma referência nula. Você não pode chamar métodos de instância em referências nulas em C# (embora o próprio IL suporte chamadas não virtuais), mas você *pode* chamar métodos de extensão com uma referência nula. Isso é demonstrado pela listagem a seguir. Observe que este não é um trecho, pois as classes aninhadas não podem conter métodos de extensão.

**Listagem 10.5 Método de extensão sendo chamado em uma referência nula**

```
usando o sistema;
classe estática pública NullUtil {

    public static bool IsNull(este objeto x) {
        retornar x == nulo;
    }

} teste de classe pública {

    static void Principal() {

        objeto y = nulo;
        Console.WriteLine(y.IsNull()); y = novo objeto();
        Console.WriteLine(y.IsNull());

    }
}
```

A saída da listagem 10.5 é True e depois False. Se IsNull fosse um método de instância normal, uma exceção teria sido lançada na segunda linha de Main; em vez disso, IsNull foi chamado com null como argumento. Antes do advento dos métodos de extensão, o C# não tinha como permitir que você escrevesse o formato y.IsNull() mais legível com segurança, exigindo NullUtil.IsNull(y) em vez disso.

Há um exemplo particularmente óbvio na estrutura em que esse aspecto do comportamento dos métodos de extensão pode ser útil: string.IsNullOrEmpty. C# 3 permite escrever um método de extensão que tenha a mesma assinatura (diferente do parâmetro extra para o tipo estendido) que um método estático existente no tipo estendido. Para evitar que você leia essa frase várias vezes, aqui está um exemplo - mesmo que a classe string tenha um método estático e sem parâmetros IsNullOrEmpty, você ainda pode criar e usar o seguinte método de extensão:

```
public static bool IsNullOrEmpty(esta string texto) {
    retornar string.IsNullOrEmpty (texto);
}
```

A princípio, parece estranho ser capaz de chamar IsNullOrEmpty em uma variável nula sem que uma exceção seja lançada, principalmente se você estiver familiarizado com ele como um método estático do .NET 2.0. Mas, na minha opinião, o código que usa o método de extensão é mais facilmente comprehensível. Por exemplo, se você ler a expressão if (name.IsNullOrEmpty()) em voz alta, ela dirá exatamente o que está fazendo.

Como sempre, experimente para ver o que funciona para você, mas esteja ciente da possibilidade de outras pessoas usarem essa técnica se você estiver depurando código. Não presumá que uma exceção será lançada em uma chamada de método, a menos que você tenha certeza de que não é um método de extensão. Além disso, pense cuidadosamente antes de reutilizar um nome existente para um método de extensão — o método de extensão anterior pode confundir os leitores que estão familiarizados apenas com o método estático da estrutura.

**VERIFICANDO A NULIDADE** Tenho certeza de que, como um desenvolvedor consciente, seus métodos de produção sempre verificam a validade de seus argumentos antes de prosseguir. Uma questão que surge naturalmente desse recurso peculiar dos métodos de extensão é qual exceção você deve lançar quando o primeiro argumento for nulo (assumindo que não é para ser assim). Deveria ser `ArgumentNullException`, como se fosse um argumento normal, ou deveria ser `NullReferenceException`, que é o que teria acontecido se o método de extensão tivesse sido um método de instância para começar? Eu recomendo o primeiro: ainda é um argumento, mesmo que a sintaxe do método de extensão não torne isso óbvio. Este é o caminho que a Microsoft seguiu para os métodos de extensão na estrutura, por isso também traz o benefício da consistência. Finalmente, tenha em mente que os métodos de extensão ainda podem ser chamados como métodos estáticos normais e, nessa situação, `ArgumentNullException` é claramente o resultado preferido.

Agora que você conhece a sintaxe e o comportamento dos métodos de extensão, podemos ver alguns exemplos daqueles fornecidos no .NET 3.5 como parte da estrutura.

### 10.3 Métodos de extensão no .NET 3.5

O maior uso de métodos de extensão no framework é para LINQ. Alguns provedores de LINQ têm alguns métodos de extensão para ajudá-los, mas há duas classes que se destacam, ambas aparecendo no namespace `System.Linq`: `Enumerable` e `Queryable`. Eles contêm muitos métodos de extensão; a maioria dos que estão em `Enumerable` operam em `IEnumerable<T>` e a maioria dos que estão em `Queryable` operam em `IQueryable<T>`. Veremos o propósito de `IQueryable<T>` no capítulo 12, mas por enquanto vamos nos concentrar em `Enumerable`.

#### 10.3.1 Primeiros passos com Enumerable

`Enumerable` contém muitos métodos, e o objetivo desta seção não é cobrir todos eles, mas dar a você uma ideia suficiente deles para que você se sinta confortável em experimentar. É uma alegria brincar com tudo o que está disponível no `Enumerable`, e definitivamente vale a pena iniciar o Visual Studio ou o LINQPad para seus experimentos (em vez de usar o Snippy), pois o IntelliSense é útil para esse tipo de atividade. O Apêndice A também fornece um rápido resumo do comportamento de todos os métodos do `Enumerable`.

Todos os exemplos completos nesta seção tratam de uma situação simples: começaremos com uma coleção de inteiros e a transformaremos de várias maneiras. As situações da vida real provavelmente serão um pouco mais complicadas, geralmente lidando com tipos relacionados a negócios, portanto, no final desta seção apresentarei alguns exemplos do lado da transformação

de coisas aplicadas a possíveis situações de negócios, com código-fonte completo disponível no site do livro. Mas é mais difícil brincar com esses exemplos do que com uma simples coleção de números.

Vale a pena considerar alguns projetos recentes nos quais você está trabalhando ao ler este capítulo; veja se você consegue pensar em situações em que poderia tornar seu código mais simples ou mais legível usando o tipo de operações descritas aqui.

Existem alguns métodos em `Enumerable` que não são métodos de extensão e usaremos um deles nos exemplos do restante do capítulo. O método `Range` usa dois parâmetros `int`: o número inicial e quantos resultados serão produzidos. O resultado é um `IEnumerable<int>` que retorna um número por vez da maneira óbvia.

Para demonstrar o método `Range` e criar uma estrutura para brincar, vamos imprimir os números de 0 a 9, conforme mostrado na listagem a seguir.

#### Listagem 10.6 Usando `Enumerable.Range` para imprimir os números de 0 a 9

```
var coleção = Enumerable.Range(0, 10);
foreach (elemento var na coleção)
{
    Console.WriteLine(elemento);
}
```

Nenhum método de extensão é chamado na listagem 10.6, apenas um método estático simples. E sim, ele realmente imprime apenas os números de 0 a 9 – nunca afirmei que esse código colocaria fogo no mundo.

**EXECUÇÃO DIFERIDA** O método `Range` não cria uma lista com os números apropriados — apenas os produz no momento apropriado. Em outras palavras, construir a instância enumerável não faz a maior parte do trabalho; ele prepara tudo para que os dados possam ser fornecidos just-in-time no momento apropriado. Isso é chamado de execução adiada – você viu esse tipo de comportamento quando examinamos os blocos iteradores no capítulo 6, mas verá muito mais sobre isso no próximo capítulo.

Praticamente a coisa mais simples que você pode fazer com uma sequência de números que já está em ordem é invertê-la. A listagem a seguir usa o método de extensão `Reverse` para fazer isso — ele retorna um `IEnumerable<T>` que produz os mesmos elementos da sequência original, mas na ordem inversa.

#### Listagem 10.7 Revertendo uma coleção com o método `Reverse`

```
var coleção = Enumerable.Range(0, 10)
    .Reverter();
foreach (elemento var na coleção) {
    Console.WriteLine(elemento);
}
```

### Eficiência: buffer versus streaming

Os métodos de extensão fornecidos pelo fluxo de estrutura ou dados de canal sempre que possível. Quando um iterador é solicitado a fornecer seu próximo elemento, ele geralmente pegará um elemento de o iterador ao qual está encadeado, processa esse elemento e então retorna algo apropriado, de preferência sem usar mais armazenamento. Transformações simples e filtros podem fazer isso facilmente e é uma maneira poderosa de processar dados com eficiência onde é possível, mas algumas operações, como inverter a ordem ou classificar, exigem todos os dados estejam disponíveis, para que todos sejam carregados na memória para processamento em massa. A diferença entre esta abordagem com buffer e a tubulação é semelhante à diferença entre a leitura de dados carregando um DataSet inteiro e o uso de um DataReader para processar um registro por vez. É importante considerar o que é necessário ao usar LINQ — uma única chamada de método pode ter implicações significativas no desempenho.

O streaming também é conhecido como avaliação preguiçosa e o buffer também é conhecido como avaliação ansiosa avaliação. Por exemplo, o método Reverse usa execução diferida (não faz nada até a primeira chamada para MoveNext), mas depois avalia avidamente sua fonte de dados. Pessoalmente, não gosto dos termos preguiçoso e ansioso, pois significam coisas diferentes para diferentes pessoas. pessoas (um tópico que discuto mais em meu blog “Quão preguiçoso você é?”: <http://mng.bz/3LLM>).

Previsivelmente, isso imprime 9, depois 8, depois 7 e assim por diante até 0. Você chamado Reverse (aparentemente) em um I Enumerable<int>, e o mesmo tipo foi devolvida. Este padrão de retornar um enumerável com base em outro é difundido em a classe Enumerável.

Vamos fazer algo mais aventureiro agora - usaremos uma expressão lambda para remova os números pares.

#### 10.3.2 Filtrando com Where e encadeando chamadas de método juntas

O método de extensão Where é uma maneira simples, mas poderosa de filtrar coleções. Isto aceita um predicado, que se aplica a cada um dos elementos da coleção original. Ele retorna um I Enumerable<T> e qualquer elemento que corresponda ao predicado é incluído na coleção resultante.

A Listagem 10.8 demonstra isso, aplicando o filtro ímpar/par à coleção de inteiros antes de revertê-lo. Você não precisa usar uma expressão lambda aqui; para Por exemplo, você pode usar um delegado criado anteriormente ou um método anônimo. Em neste caso (e em muitas outras situações da vida real), é simples colocar a lógica de filtragem expressões inline e lambda mantêm o código conciso.

#### Listagem 10.8 Usando o método Where com uma expressão lambda para encontrar números ímpares

```
var coleção = Enumerable.Range(0, 10)
    .Onde(x => x % 2 != 0)
    .Reverter();

foreach (elemento var na coleção)
{
    Console.WriteLine(elemento);
}
```

A Listagem 10.8 imprime os números 9, 7, 5, 3 e 1. Esperamos que você tenha notado um formação de padrões – você está encadeando as chamadas de método. A própria ideia de encadeamento não é novo. Por exemplo, `StringBuilder.Replace` sempre retorna a instância que você chama ativado, permitindo código como este:

```
construtor = construtor.Replace("<", "&lt;")
    .Replace(">", "&gt;")
    ...
```

Por outro lado, `String.Replace` retorna uma string, mas uma nova a cada vez - isso permite encadeamento, mas de uma maneira um pouco diferente. Ambos os padrões são úteis para conhecer; o O padrão “retornar a mesma referência” funciona bem para tipos mutáveis, enquanto “retornar uma nova instância que é uma cópia do original com algumas alterações” é necessária para tipos imutáveis.

**Encadeamento com métodos de instância como `String.Replace` e `StringBuilder.Replace`** foi simples, mas os métodos de extensão permitem chamadas de métodos *estáticos* para estar acorrentados juntos. *Esta é uma das principais razões pelas quais existem métodos de extensão.* Eles estão útil para outras classes de utilidade, mas seu verdadeiro poder é revelado nesta habilidade de encadear métodos estáticos de maneira natural. É por isso que os métodos de extensão aparecem principalmente em `Enumerable` e consultável em .NET: o LINQ é voltado para essa abordagem de processamento de dados, com informações viajando efetivamente por pipelines construídos a partir de operações individuais encadeadas.

**CONSIDERAÇÃO DE EFICIÊNCIA: CHAMADAS DE MÉTODO DE REORDENAMENTO PARA EVITAR DESPERDÍCIO** Estou não sou fã de micro-otimização sem uma boa causa, mas vale a pena dar uma olhada a ordem das chamadas de método na listagem 10.8. Você poderia ter adicionado o `Where` chamada após a chamada reversa e obteve os mesmos resultados, mas isso teria desperdiçou algum esforço - a chamada reversa teria que descobrir onde os números pares devem vir na sequência, mesmo que sejam descartados do resultado final. Neste caso, não fará muita diferença, mas pode têm um efeito significativo no desempenho em situações reais; se você pode reduzir a quantidade de trabalho desperdiçado sem comprometer a legibilidade, isso é uma boa coisa. Isso não significa que você deva sempre colocar filtros no início do pipeline; você precisa pensar cuidadosamente sobre qualquer reordenamento para ter certeza você obtém os resultados corretos.

Existem duas maneiras óbvias de escrever a primeira parte da listagem 10.8 sem usar o fato de que `Reverse` e `Where` são métodos de extensão. Uma é usar uma variável temporária, que mantém a estrutura intacta:

```
var coleção = Enumerable.Range(0, 10);
coleção      = Enumerable.Where(coleção, x => x % 2 != 0)
coleção      = Enumerable.Reverse(coleção);
```

Espero que você concorde que o significado do código é muito menos claro aqui do que na listagem 10.8.

Fica ainda pior com a outra opção, que é manter o estilo de instrução única:

```
var coleção = Enumerable.Reverse
    (Enumerable.Onde
        (Enumerable.Range(0, 10), x => x % 2 != 0));
```

A ordem de chamada do método parece estar invertida, porque a chamada do método mais interno (Range) será executada primeiro, depois as outras, com a execução saindo para fora. Mesmo com apenas três chamadas de método é feio — fica muito pior para consultas que envolvem mais operadores.

Antes de prosseguirmos, vamos pensar um pouco sobre o que o método Where faz.

#### 10.3.3 Interlúdio: não vimos o método Where antes?

Se o método Where parecer familiar, é porque você o implementou no capítulo 6. Tudo o que você precisa fazer é converter a listagem 6.9 em um método de extensão e alterar o tipo de delegado de `Predicate<T>` para `Func<T, bool>` e você terá uma implementação alternativa perfeitamente boa para `Enumerable.Where`:

```
public static IEnumerable<T> Where<T>(esta fonte IEnumerable<T>,
                                         Func<T, bool> predicado)
{
    if (fonte == nulo || predicado == nulo) {
        lançar novo ArgumentNullException();
    } return WhereImpl(fonte, predicado);
}

private static IEnumerable<T> WhereImpl<T>(IEnumerable<T> fonte,
                                         Func<T, bool> predicado)
{
    foreach (item T na fonte) {
        if (predicado(item)) {
            item de retorno de rendimento;
        }
    }
}
```

Você pode alterar a última parte da listagem 6.9 para torná-la mais parecida com LINQ também:

```
foreach (linha de string em LineReader.ReadLines("../FakeLinq.cs"))
    .Where(linha => linha.StartsWith("usando")))
{
    Console.WriteLine(linha);
}
```

Esta é efetivamente uma consulta LINQ sem usar o namespace `System.Linq`. Funcionaria perfeitamente no .NET 2.0 se você declarasse o delegado `Func` apropriado e `[ExtensionAttribute]`. Você pode até usar essa implementação para a cláusula `where` em uma expressão de consulta (ainda visando o .NET 2.0), como verá no próximo capítulo — mas não vamos nos precipitar.

Filtrar é uma das operações mais simples em uma consulta e outra é transformar ou *projetar* os resultados.

#### 10.3.4 Projeções utilizando o método Select e tipos anônimos

O método de projeção mais comumente usado em `Enumerable` é `Select`. Ele opera em um `IEnumerable<TSource>` e o projeta em um `IEnumerable<TResult>` por meio de um `Func<TSource, TResult>`, que é a transformação a ser usada em cada elemento, especificado como um delegado. É muito parecido com o método `ConvertAll` em `List<T>`, mas opera em qualquer coleção enumerável e usa execução adiada para realizar a projeção quando cada elemento é solicitado.

Quando apresentei os tipos anônimos, disse que eles eram úteis com expressões lambda e LINQ — aqui está um exemplo do tipo de coisa que você pode fazer com eles. Atualmente você tem números ímpares de 0 a 9 (em ordem inversa) — vamos criar um tipo que encapsule a raiz quadrada do número, bem como o número original. A listagem a seguir mostra tanto a projeção quanto uma forma ligeiramente modificada de escrever os resultados. Ajustei o espaço em branco apenas por questão de espaço na página impressa.

##### Listagem 10.9 Projeção usando uma expressão lambda e um tipo anônimo

```
var coleção = Enumerable.Range(0, 10)
    .Onde(x => x % 2 != 0)
    .Reverter()
    .Select(x => novo { Original = x, SquareRoot = Math.Sqrt(x) });

foreach (elemento var na coleção) {

    Console.WriteLine("sqrt({0})={1}",
        elemento.Original,
        elemento.SquareRoot);
}
```

Desta vez, o tipo de coleção não é `IEnumerable<int>` — é `IEnumerable <Something>`, onde `Something` é o tipo anônimo criado pelo compilador. Você não pode fornecer à variável de coleção um tipo explícito diferente do tipo ou objeto não genérico `IEnumerable`. A digitação implícita (com `var`) é o que permite usar as propriedades `Original` e `SquareRoot` ao escrever os resultados.

A saída da listagem 10.9 é a seguinte:

```
quadrado(9)=3
quadrado(7)=2,64575131106459
quadrado(5)=2,23606797749979
quadrado(3)=1,73205080756888
quadrado(1)=1
```

É claro que um método `Select` não precisa usar um tipo anônimo — você poderia ter selecionado apenas a raiz quadrada do número, descartando o original. Nesse caso, o resultado seria `IEnumerable<double>`. Como alternativa, você poderia ter escrito manualmente um tipo para encapsular um número inteiro e sua raiz quadrada — nesse caso, seria mais fácil usar um tipo anônimo.

Vejamos um último método para completar nossa cobertura de Enumerable para o momento: OrderBy.

### 10.3.5 Classificando usando o método OrderBy

A classificação é um requisito comum no processamento de dados e, no LINQ, isso geralmente é realizado usando os métodos OrderBy ou OrderByDescendente. A primeira chamada às vezes é seguida por ThenBy ou ThenByDecrescente se você precisar classificar por mais de uma propriedade dos dados. Essa capacidade de classificar múltiplas propriedades sempre esteve disponível da maneira mais difícil, usando uma comparação complicada, mas é muito mais claro poder apresentar uma série de comparações simples.

Para demonstrar isso, vamos fazer uma pequena alteração nas operações envolvidas. Você começará com os números inteiros -5 a 5 (inclusive, portanto há 11 elementos no total) e depois projetará para um tipo anônimo contendo o número original e seu quadrado (em vez de raiz quadrada). Finalmente, você classificará pelo quadrado e depois pelo número original. A listagem a seguir mostra tudo isso.

#### Listagem 10.10 Ordenando uma sequência por duas propriedades

```
var coleção = Enumerable.Range(-5, 11)
    .Select(x => novo {Original = x, Quadrado = x * x})
    .ThenBy(x => x.Original);

foreach (elemento var na coleção)
{
    Console.WriteLine(elemento);
}
```

Observe como, além da chamada para Enumerable.Range, o código é quase exatamente igual à descrição textual. A implementação ToString do tipo anônimo faz a formatação desta vez, e aqui estão os resultados:

```
{Original = 0, Quadrado = 0}
{Original = -1, Quadrado = 1}
{Original = 1, Quadrado = 1}
{Original = -2, Quadrado = 4}
{Original = 2, Quadrado = 4}
{Original = -3, Quadrado = 9}
{Original = 3, Quadrado = 9}
{Original = -4, Quadrado = 16}
{Original = 4, Quadrado = 16}
{Original = -5, Quadrado = 25}
{Original = 5, Quadrado = 25}
```

Como pretendido, a principal propriedade de classificação é Square, mas quando dois valores têm o mesmo quadrado, o número original negativo é sempre classificado antes do positivo. Escrever uma única comparação para fazer o mesmo tipo de coisa (em um caso geral – existem truques matemáticos para lidar com este exemplo específico) teria sido significativamente mais complicado, na medida em que você não gostaria de incluir o código embutido na expressão lambda.

Uma coisa a observar é que a ordem não altera uma coleção existente – ela retorna uma nova sequência que produz os mesmos dados que a sequência de entrada, exceto classificada. Compare isso com `List<T>.Sort` ou `Array.Sort`, que alteram o elemento ordem dentro da lista ou array. Os operadores LINQ pretendem ser livres de efeitos colaterais: eles não afetam suas contribuições e não fazem nenhuma outra alteração no ambiente, a menos que você esteja iterando por meio de uma sequência com estado natural (como a leitura de um fluxo de rede) ou um argumento delegado tem efeitos colaterais. Esta é uma abordagem de programação funcional e leva a um código mais legível, testável, combinável, previsível, seguro para threads e robusto.

Vimos apenas alguns dos muitos métodos de extensão disponíveis em `Enumerable`, mas espero que você possa apreciar o quão bem eles podem ser encadeados. Na próxima capítulo, você verá como isso pode ser expresso de uma maneira diferente usando a sintaxe extra fornecida pelo C# 3 (expressões de consulta) e veremos algumas outras operações que ainda não vimos. coberto aqui. Vale lembrar que você não precisa usar expressões de consulta— muitas vezes pode ser mais simples fazer algumas chamadas para métodos em `Enumerable`, usando métodos de extensão para encadear operações.

Agora que você viu como tudo isso se aplica ao exemplo da coleção de números, é hora de cumprir a promessa de mostrar a você algumas informações relacionadas a negócios exemplos.

#### 10.3.6 Exemplos de negócios envolvendo encadeamento

Muito do que fazemos como desenvolvedores envolve a movimentação de dados. Na verdade, para muitas aplicações, essa é a única coisa significativa que fazemos – a interface do usuário, os serviços web, o banco de dados e outros componentes geralmente existem apenas para levar dados de um lugar para outro, ou de uma forma para outra. Não deveria ser nenhuma surpresa que os métodos de extensão que vimos nesta seção são adequados para muitos problemas de negócios.

Vou apenas dar alguns exemplos aqui. Tenho certeza que você será capaz de imaginar como o C# 3 e a classe `Enumerable` pode ajudá-lo a resolver problemas que envolvem os requisitos do seu negócio de forma mais expressiva do que antes. Para cada exemplo, incluirei apenas uma amostra consulta - deve ser suficiente para ajudá-lo a entender o propósito do código, mas sem toda a bagagem. O código funcional completo está no site do livro.

##### AGREGAÇÃO: SOMA DOS SALÁRIOS

O primeiro exemplo envolve uma empresa composta por vários departamentos. Cada departamento tem um número de funcionários, cada um dos quais recebe um salário. Suponha que você queira relatório sobre o custo salarial total por departamento, com o departamento mais caro listado primeiro. A consulta é simplesmente a seguinte:

```
empresa.Departamentos
    .Select(dept => novo
    {
        nome do departamento,
        Custo = dept.Employees.Sum(pessoa => pessoa.Salário)
    })
    .OrderByDescendente(deptWithCost => deptWithCost.Cost);
```

Esta consulta usa um tipo anônimo para manter o nome do departamento (usando um inicializador de projeção) e a soma dos salários de todos os funcionários desse departamento.

O somatório de salários usa um método de extensão de soma autoexplicativo , novamente parte do Enumerável.

No resultado, o nome do departamento e o salário total podem ser recuperados como propriedades. Se você quisesse a referência original do departamento, bastaria alterar o tipo anônimo usado no método Select .

#### AGRUPAMENTO: CONTANDO BUGS ATRIBUÍDOS AOS DESENVOLVEDORES

Se você é um desenvolvedor profissional, tenho certeza de que já viu muitas ferramentas de gerenciamento de projetos que fornecem métricas diferentes. Se você tiver acesso aos dados brutos, o LINQ pode ajudá-lo a transformá-los praticamente da maneira que você escolher.

Como um exemplo simples, vejamos uma lista de desenvolvedores e quantos bugs eles atribuíram a eles no momento:

```
bugs.GroupBy(bug => bug.AssignedTo)
    .Select(lista => novo {Desenvolvedor = lista.Key, Contagem = lista.Count() })
    .OrderByDescendente(x => x.Count);
```

Esta consulta usa o método de extensão GroupBy , que agrupa a coleção original por uma projeção (o desenvolvedor designado para corrigir o bug, neste caso), resultando em um IGrouping< TKey, TElement >. Existem muitas sobrecargas de GroupBy, mas este exemplo usa a mais simples e depois seleciona apenas a chave (o nome do desenvolvedor) e a quantidade de bugs atribuídos a ele. Depois disso, você ordena o resultado para mostrar primeiro os desenvolvedores com mais bugs.

Um dos problemas ao observar a classe Enumerable pode ser descobrir exatamente o que está acontecendo; por exemplo, uma das sobrecargas de GroupBy possui quatro parâmetros de tipo e cinco parâmetros normais (três dos quais são delegados). Não entre em pânico — basta seguir os passos mostrados no capítulo 3, atribuindo diferentes tipos a diferentes parâmetros de tipo até ter um exemplo concreto de como seria o método.

Isso geralmente torna muito mais fácil entender o que está acontecendo.

Esses exemplos não são particularmente complicados, mas espero que você possa ver o poder de encadear chamadas de métodos, onde cada método pega uma coleção original e retorna outra de uma forma ou de outra, seja filtrando alguns valores, ordenando valores, transformando cada elemento, agregando alguns valores ou usando outras opções. Em muitos casos, o código resultante pode ser lido em voz alta e compreendido imediatamente e, em outras situações, ainda é geralmente muito mais simples do que o código equivalente seria nas versões anteriores do C#.

Usaremos o exemplo de rastreamento de defeitos como dados de amostra quando examinarmos as expressões de consulta no próximo capítulo. Agora que você viu alguns dos métodos de extensão fornecidos, vamos considerar como e quando faz sentido escrevê-los você mesmo.

#### 10.4 Ideias e diretrizes de uso

Assim como a tipagem implícita de variáveis locais, os métodos de extensão são controversos. Seria difícil afirmar que eles tornam o objetivo geral do código mais difícil de entender.

muitos casos, mas ao mesmo tempo obscurecem os detalhes de qual método está sendo chamado. Nas palavras de um dos professores da minha universidade: "Estou escondendo a verdade para lhe mostrar uma verdade maior." Se você acredita que o aspecto mais importante do código é o resultado, os métodos de extensão são ótimos. Se a implementação for mais importante para você, então chamar explicitamente um método estático é mais claro. Efetivamente, é a diferença entre o quê e o como.

Já vimos como usar métodos de extensão para classes utilitárias e métodos encadeamento, mas antes de discutirmos mais detalhadamente os prós e os contras, vale a pena destacar alguns aspectos que podem não ser óbvios.

##### 10.4.1 "Estendendo o mundo" e tornando as interfaces mais ricas

Wes Dyer, ex-desenvolvedor da equipe do compilador C#, tem um blog fantástico que cobre todos os tipos de assunto ([consulte http://blogs.msdn.com/b/wesdyer/](http://blogs.msdn.com/b/wesdyer/)). Um dos seus postagens sobre métodos de extensão chamaram minha atenção particularmente (veja <http://mng.bz/I4F2>). Chama-se "Extending the World" e fala sobre como os métodos de extensão pode tornar o código mais fácil de ler, adaptando efetivamente seu ambiente às suas necessidades:

Normalmente, para um determinado problema, um programador está acostumado a construir uma solução até que finalmente atenda aos requisitos. Agora, é possível estender o mundo para atender solução em vez de apenas construir até chegarmos a ela. Essa biblioteca não fornece o que você precisa, basta ampliar a biblioteca para atender às suas necessidades.

Isso tem implicações além das situações em que você usaria uma classe utilitária. Normalmente, os desenvolvedores só começam a criar classes utilitárias quando veem o mesmo tipo de código reproduzido em dezenas de lugares, mas estender uma biblioteca é uma questão de clareza de expressão, pois tanto quanto evitar duplicação. Os métodos de extensão podem fazer com que o código de chamada pareça a biblioteca é mais rica do que realmente é.

Você já viu isso com `IEnumerable<T>`, onde mesmo a implementação mais simples parece ter um amplo conjunto de operações disponíveis, como classificação, agrupamento, projeção e filtragem. Os benefícios não se limitam às interfaces – você também pode “estender o mundo” com enumerações, classes abstratas e assim por diante.

O .NET Framework também fornece um bom exemplo de outro uso para extensão métodos: interfaces fluentes.

##### 10.4.2 Interfaces fluentes

Costumava haver um programa de televisão no Reino Unido chamado Catchphrase. O ideia era que os competidores assistissem a uma tela onde uma animação mostraria algumas versões enigmáticas de uma frase ou ditado, que eles teriam que adivinhar. O anfitrião muitas vezes tenta ajudar instruindo-os: “Diga o que você vê”. Essa é basicamente a ideia por trás de interfaces fluentes - que se você ler o código literalmente, seu propósito saltará

a tela como se estivesse escrita em uma linguagem humana natural. O termo “interfaces fluentes” foi originalmente cunhado por Martin Fowler (veja seu blog em <http://mng.bz/3T9T>) e Eric Evans.

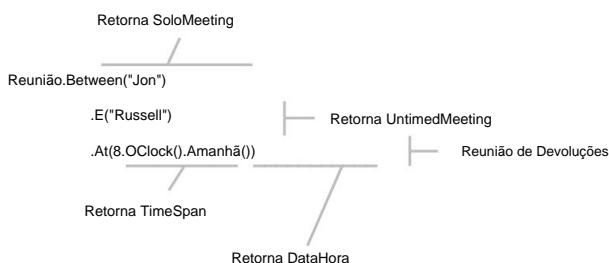
Se você estiver familiarizado com *linguagens específicas de domínio* (DSLs), você pode estar se perguntando o que as diferenças estão entre uma interface fluente e uma DSL.. Muito foi escrito sobre assunto, mas o consenso parece ser que uma DSL tem mais liberdade para criar sua própria sintaxe e gramática, enquanto uma interface fluente é limitada pela linguagem hospedeira (C#, no nosso caso).

Alguns bons exemplos de interfaces fluentes no framework são OrderBy e Métodos ThenBy : com um pouco de interpretação das expressões lambda, o código explica exatamente o que faz. No caso da listagem 10.10 anterior, você poderia ler “ordem pelo quadrado, depois pelo número original” sem muito trabalho. Declarações acabam sendo lidas como frases inteiras, em vez de frases nominais e verbais individuais.

Escrever interfaces fluentes pode exigir uma mudança de mentalidade. Os nomes dos métodos desafiam o forma verbal descritiva normal, com *And*, *Then* e *If* às vezes sendo métodos adequados em uma interface fluente. Os próprios métodos muitas vezes fazem pouco mais do que configurar o contexto para chamadas futuras, muitas vezes retornando um tipo cujo único propósito é atuar como uma ponte. entre chamadas. A Figura 10.2 ilustra como essa ponte funciona. Ele usa apenas duas extensões métodos (em int e TimeSpan), mas fazem toda a diferença na legibilidade.

A gramática do exemplo da figura 10.2 poderia ter muitas formas diferentes; você pode adicionar participantes adicionais a um UntimedMeeting ou criar um UnattendedMeeting em um determinado momento antes de especificar os participantes, por exemplo. Para Para obter mais orientações sobre DSLs, consulte *DSLs em Boo: linguagens específicas de domínio em .NET* por Ayende Rahien (Manning, 2010).

C# 3 suporta apenas *métodos de extensão* em vez de *propriedades de extensão*, o que restringe interfaces fluentes ligeiramente. Isso significa que você não pode ter expressões como 1.week.from.now ou 2 dias + 10 horas (ambos válidos no Groovy com um pacote apropriado—consulte o suporte de dados do Google do Groovy: <http://groovy.codehaus.org/Google+Data+Suporte>), mas com alguns parênteses supérfluos você pode obter resultados semelhantes. Inicialmente parece estranho chamar um método em um número (como 2.Dollars() ou 3.Meters()), mas é difícil negar que o significado é claro. Sem métodos de extensão, esse tipo de clareza não é possível quando você precisa agir em tipos como números que não estão sob seu controle.



**Figura 10.2 Separando uma expressão de interface fluente para criar uma reunião. A hora da reunião é especificada usando métodos de extensão para criar um TimeSpan de um int e um DateTime de um TimeSpan.**

No momento em que este artigo foi escrito, a comunidade de desenvolvimento ainda estava em dúvida sobre interfaces fluentes: elas são relativamente raras na maioria dos campos, embora muitas bibliotecas de simulação e testes unitários tenham pelo menos alguns aspectos fluentes. Certamente não são universalmente aplicáveis, mas nas situações certas podem transformar radicalmente a legibilidade do código de chamada. Por exemplo, com métodos de extensão apropriados da minha biblioteca Misc Util, posso iterar todos os dias em que estive vivo de uma forma legível:

```
foreach (DateTime dia em 19.June(1976).To(DateTime.Today)
    .Step(1.Dias()))
```

Embora os detalhes de implementação relacionados ao intervalo sejam complicados, os métodos de extensão que permitem 19.June(1976) e 1.Days() são extremamente simples. Este é um código específico da cultura, que você pode não querer expor em seu código de produção, mas pode tornar os testes de unidade muito mais agradáveis.

É claro que esses não são os únicos usos disponíveis para métodos de extensão. Eu os usei para validação de argumentos, implementando abordagens alternativas ao LINQ, adicionando meus próprios operadores ao LINQ to Objects, facilitando a construção de comparações compostas, adicionando mais funcionalidades relacionadas a sinalizadores às enumerações e muito mais. Fico constantemente surpreso ao ver como um recurso tão simples pode ter um impacto tão profundo na legibilidade quando usado de maneira adequada. A palavra-chave é “apropriadamente”, que é mais fácil de dizer do que descrever.

#### 10.4.3 Usando métodos de extensão de maneira sensata

Não estou em posição de ditar como você escreve seu código. Pode ser possível escrever testes para medir objetivamente a legibilidade para um desenvolvedor médio, mas isso só importa para aqueles que usarão e manterão seu código. Você precisa consultar as pessoas relevantes tanto quanto possível, apresentando diferentes opções e obtendo feedback apropriado. Os métodos de extensão tornam isso particularmente fácil em muitos casos, pois você pode demonstrar ambas as opções no código de trabalho simultaneamente – transformar um método em um método de extensão não impede que você o chame explicitamente da mesma maneira que antes.

A principal pergunta a ser feita é aquela a que me referi no início desta seção: o aspecto “o que ele faz” do código é mais importante do que o aspecto “como ele faz”?

Isso varia de acordo com a pessoa e a situação, mas aqui estão algumas diretrizes a serem lembradas:

- ÿ Todos na equipe de desenvolvimento devem estar cientes dos métodos de extensão e onde eles podem ser usados. Sempre que possível, evite surpreender os mantenedores de código. ÿ Ao colocar extensões em seu próprio namespace, você dificulta seu uso acidental. Mesmo que não seja óbvio ao ler o código, os desenvolvedores que o escrevem devem estar cientes do que estão fazendo. Use uma convenção para todo o projeto ou para toda a empresa para nomear o namespace. Você pode optar por dar um passo adiante e usar um único namespace para cada tipo estendido. Por exemplo, você poderia criar um namespace TypeExtensions para classes que estendem System.Type.

- ÿ Pense cuidadosamente antes de estender tipos amplamente usados, como números ou objetos, ou antes de escrever um método onde o tipo estendido seja um parâmetro de tipo. Algumas diretrizes chegam ao ponto de recomendar que você não faça isso; Acho que essas extensões têm seu lugar, mas deveriam realmente merecer seu lugar em sua biblioteca. Nessa situação, é ainda mais importante que o método de extensão seja interno ou esteja em seu próprio namespace; Eu não gostaria que o IntelliSense sugerisse o método de extensão June em todos os lugares onde eu usasse um número inteiro, por exemplo — apenas em classes que usassem pelo menos alguns métodos de extensão relacionados a data e hora.
- ÿ A decisão de escrever um método de extensão deve ser sempre consciente. Não deveria se tornar habitual. Nem todo método estático merece ser um método de extensão.
- ÿ Documente se o primeiro parâmetro (o valor no qual seu método parece ser chamado) pode ser nulo — se não for, verifique o valor no método e lance uma ArgumentNullException se necessário.
- ÿ Tenha cuidado para não usar um nome de método que já tenha um significado no tipo estendido. Se o tipo estendido for um tipo de estrutura ou vier de uma biblioteca de terceiros, verifique todos os nomes de seus métodos estendidos sempre que alterar as versões da biblioteca. Se você tiver sorte (como eu tive com Stream.CopyTo), o novo significado é igual ao antigo, mas mesmo assim, você pode querer descontinuar seu método de extensão.
- ÿ Questione seus instintos, mas reconheça que eles afetam sua produtividade. Assim como acontece com a digitação implícita, não faz sentido se forçar a usar um recurso que você instintivamente não gosta.
- ÿ Tente agrupar métodos de extensão em classes estáticas que lidam com o mesmo tipo estendido. Às vezes, classes relacionadas (como DateTime e TimeSpan) podem ser agrupadas de maneira sensata, mas evite agrupar métodos de extensão direcionados a tipos diferentes, como Stream e string dentro da mesma classe.
- ÿ Pense bem antes de adicionar métodos de extensão com o mesmo tipo estendido e mesmo nome em dois namespaces diferentes, especialmente se houver situações em que os diferentes métodos possam ser aplicáveis (eles têm o mesmo número de parâmetros). É razoável adicionar ou remover uma diretiva using para fazer com que um programa falhe na compilação, mas será desagradável se ele ainda for compilado, mas alterar o comportamento.

Poucas destas orientações são particularmente claras; até certo ponto, você terá que encontrar o melhor caminho para usar ou evitar métodos de extensão. É perfeitamente razoável nunca escrever seus próprios métodos de extensão e usar os relacionados ao LINQ para obter os ganhos de legibilidade disponíveis. Mas vale a pena pelo menos pensar no que é possível.

## 10.5 Resumo

O aspecto mecânico dos métodos de extensão é direto – o recurso é simples de descrever e demonstrar. É mais difícil falar sobre seus benefícios (e custos) de maneira definitiva – é um assunto delicado e pessoas diferentes tendem a ter opiniões diferentes sobre o valor fornecido.

Neste capítulo tentei mostrar um pouco de tudo. Inicialmente, analisamos o que o recurso alcança na linguagem e, em seguida, analisamos alguns dos recursos disponíveis por meio da estrutura. De certa forma, esta foi uma introdução relativamente suave ao LINQ; revisitaremos alguns dos métodos de extensão que você viu até agora e veremos alguns novos, quando nos aprofundarmos nas expressões de consulta no próximo capítulo.

Uma grande variedade de métodos está disponível na classe `Enumerable`, e apenas arranhamos a superfície neste capítulo. É divertido criar um cenário criado por você mesmo (seja hipotético ou em um projeto real) e navegar pelo MSDN para ver o que está disponível para ajudá-lo. Recomendo que você use algum tipo de projeto sandbox para brincar com os métodos de extensão fornecidos - parece mais uma diversão do que um trabalho, e é improvável que você queira se limitar apenas aos métodos necessários para atingir seu objetivo mais imediato . O Apêndice A contém uma lista dos operadores de consulta padrão do LINQ, que cobre muitos dos métodos em `Enumerable`.

Novos padrões e práticas continuam surgindo na engenharia de software, e as ideias de alguns sistemas muitas vezes se cruzam com outros. Essa é uma das coisas que mantém o desenvolvimento emocionante. Os métodos de extensão permitem que o código seja escrito de uma forma que antes não estava disponível em C#, criando interfaces fluentes e alterando o ambiente para se adequar ao seu código, e não o contrário. Essas são apenas as técnicas que examinamos neste capítulo – certamente haverá desenvolvimentos futuros interessantes usando os novos recursos do C#, sejam eles individualmente ou combinados.

A revolução obviamente não termina aqui. Para algumas chamadas, os métodos de extensão são adequados. No próximo capítulo, veremos as verdadeiras ferramentas poderosas: expressões de consulta e LINQ completo.

# 11

## *Expressões de consulta e LINQ to Objects*

### **Este capítulo cobre**

- ÿ Sequências de streaming de dados e execução adiada
- ÿ Operadores de consulta padrão e tradução de expressões de consulta
- ÿ Variáveis de intervalo e identificadores transparentes
- ÿ Projetar, filtrar e classificar
- ÿ Unir e agrupar
- ÿ Escolhendo qual sintaxe usar

Você já deve estar cansado de toda a hipérbole em torno do LINQ . Você já viu alguns exemplos no livro e quase certamente leu muito sobre isso na web. É aqui que separamos o mito da realidade:

- ÿ O LINQ não transforma a consulta mais complicada em uma única linha. ÿ LINQ não significa que você nunca mais precisará olhar para SQL bruto novamente. ÿ O LINQ não imbui você magicamente de genialidade arquitetônica.

Considerando tudo isso, o LINQ ainda é a melhor maneira de expressar consultas que já vi em um ambiente orientado a objetos. Não é uma solução milagrosa, mas é uma ferramenta muito poderosa para se ter em seu arsenal de desenvolvimento. Exploraremos dois aspectos distintos do LINQ: o suporte da estrutura e a tradução do compilador de expressões de consulta. Estes últimos podem parecer estranhos no início, mas tenho certeza que você aprenderá a amá-los.

As expressões de consulta são efetivamente pré-processadas pelo compilador em “normais” C# 3, que é então compilado de maneira comum. Esta é uma maneira simples de integrar consultas na linguagem sem alterar a especificação em mais de uma pequena seção. A maior parte deste capítulo é uma lista das traduções de pré-processamento realizadas pelo compilador, bem como os efeitos alcançados quando o resultado usa os métodos de extensão `Enumerable`.

Você não verá nenhum SQL ou XML aqui – isso espera por você no capítulo 12. Mas com este capítulo como base, você será capaz de entender o que os provedores LINQ mais interessantes fazem quando você os conhece. Chame-me de desmancha-prazeres, mas quero tirar um pouco da magia deles. Mesmo sem o ar de mistério, o LINQ ainda é muito legal.

Primeiro, vamos considerar a base do LINQ e como iremos explorá-lo.

## 11.1 Apresentando o LINQ Com

um tópico tão amplo quanto o LINQ, você precisa de uma certa quantidade de conhecimento antes de estar pronto para vê-lo em ação. Nesta seção, veremos alguns dos princípios fundamentais por trás do LINQ e o modelo de dados que usaremos nos exemplos deste capítulo e do próximo. Eu sei que você provavelmente estará ansioso para entrar no código, então serei bastante breve.

### 11.1.1 Conceitos fundamentais em LINQ

Um dos problemas com a redução da incompatibilidade de impedância entre dois modelos de dados é que isso geralmente envolve a criação de outro modelo para atuar como ponte. Esta seção descreve o modelo LINQ, começando com seu aspecto mais importante: sequências.

**SEQUÊNCIAS** Você já está familiarizado com o conceito de sequência: ela é encapsulada pelas interfaces `IEnumerable` e `IEnumerable<T>`, e examinamos isso mais de perto no capítulo 6, quando estudamos iteradores. Uma sequência é como uma esteira rolante de itens – você os busca, um de cada vez, até que não esteja mais interessado ou até que a sequência fique sem dados.

A principal diferença entre uma sequência e outras estruturas de dados de coleção, como listas e matrizes, é que quando você lê uma sequência, geralmente não sabe quantos itens a mais estão esperando e não tem acesso a itens arbitrários – apenas o atual. Na verdade, algumas sequências podem ser intermináveis; você poderia facilmente ter uma sequência infinita de números aleatórios, por exemplo. Listas e matrizes podem atuar como sequências, assim como `List<T>` implementa `IEnumerable<T>`, mas o inverso nem sempre é verdadeiro. Você não pode ter um array ou lista infinita, por exemplo.

As sequências são o pão com manteiga do LINQ. Ao ler uma expressão de consulta, você deve pensar nas sequências envolvidas; há sempre pelo menos uma sequência para

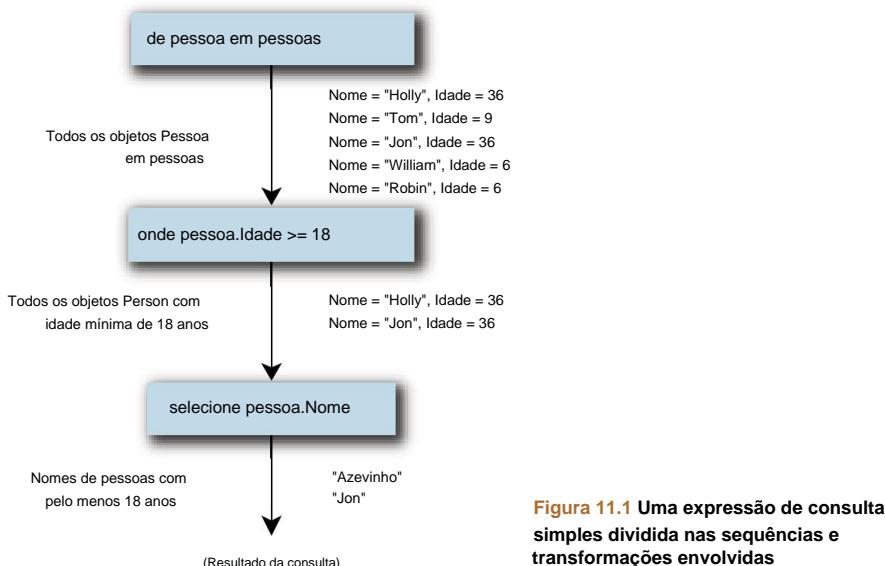
começa, e geralmente é transformado em outras sequências ao longo do caminho, possivelmente sendo unido a ainda mais sequências. Os exemplos de consulta LINQ na Web geralmente têm pouca explicação, mas quando você os separa observando cada sequência por vez, as coisas fazem muito mais sentido. Além de auxiliar na *leitura* do código, essa abordagem também pode ajudar muito na hora de *escrevê-lo*. Pensar em sequências pode ser complicado - às vezes é um salto mental - mas se você conseguir chegar lá, isso o ajudará imensamente quando estiver trabalhando com LINQ.

Como exemplo simples, vamos usar uma expressão de consulta executada em uma lista de pessoas. Aplicaremos primeiro um filtro e depois uma projeção, para terminarmos com uma sequência de nomes de adultos:

```
var adultNames = de pessoa em pessoas onde
    pessoa.Idade >= 18 selecione
        pessoa.Nome;
```

A Figura 11.1 mostra essa expressão de consulta graficamente, dividindo-a em etapas individuais.

Cada seta representa uma sequência – a descrição está no lado esquerdo e alguns dados de amostra estão à direita. Cada caixa é uma etapa na expressão de consulta. Inicialmente, você tem toda a família (como objetos Person); então, após a filtragem, a sequência contém apenas adultos (novamente, como objetos Person); e o resultado final tem os nomes desses adultos como strings. Cada vez, você pega uma sequência e aplica uma operação para produzir uma nova sequência. O resultado não são as strings Holly e Jon — em vez disso, é um `IEnumerable<string>`, que, quando solicitado por seus elementos um por um, produzirá primeiro Holly e depois Jon.



**Figura 11.1** Uma expressão de consulta simples dividida nas sequências e transformações envolvidas

Este exemplo foi simples no início, mas aplicaremos a mesma técnica posteriormente a expressões de consulta mais complicadas para entendê-las mais facilmente.

Algumas operações avançadas envolvem mais de uma sequência como entrada, mas ainda é muito menos preocupante do que tentar entender toda a consulta de uma só vez.

E por que as sequências são tão importantes? Eles são a base para um modelo de streaming para manipulação de dados – aquela que permite buscar e processar dados somente quando necessário.

#### EXECUÇÃO E STREAMING DIFERIDOS

Quando a expressão de consulta mostrada na figura 11.1 é criada, nenhum dado é processado. A lista original de pessoas não é acessada.<sup>1</sup> Em vez disso, uma representação da consulta é construída na memória. Instâncias de delegação são usadas para representar o teste de predicado para a idade adulta e a conversão de uma pessoa para o nome dessa pessoa. As rodas só começam a girar quando o `IEnumerable<string>` resultante é solicitado para seu primeiro elemento.

Este aspecto do LINQ é chamado *de execução adiada*. Quando o primeiro elemento do resultado é solicitado, a transformação `Select` solicita à transformação `Where` seu primeiro elemento. A transformação `Where` solicita à lista seu primeiro elemento, verifica se o predicado corresponde (o que acontece, neste caso) e retorna esse elemento de volta para `Select`. Isso, por sua vez, extrai o nome e o retorna como resultado.

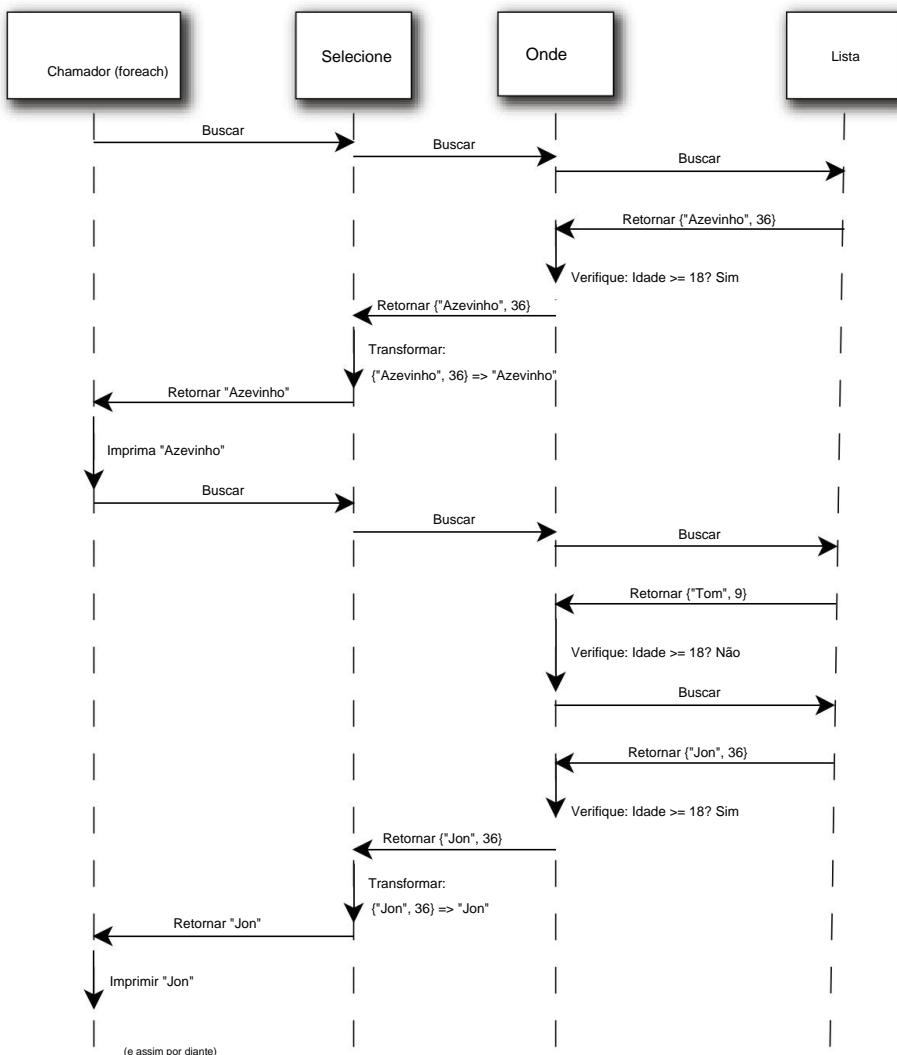
**NÃO VIMOS ISSO ANTES?** Você pode estar tendo uma sensação de déjà vu aqui, porque mencionei tudo isso no capítulo 10. Mas é um tópico tão importante que vale a pena abordar uma segunda vez, com mais detalhes.

Isso é complicado, mas um diagrama de sequência torna tudo muito mais claro. Recolherei as chamadas para `MoveNext` e `Current` para uma única operação de busca; isso torna o diagrama muito mais simples. Apenas lembre-se de que cada vez que a busca ocorre, ela também verifica efetivamente o final da sequência. A Figura 11.2 mostra os primeiros estágios do exemplo de expressão de consulta em operação, quando você imprime cada elemento do resultado usando um loop `foreach`.

Como você pode ver na figura 11.2, apenas um elemento de dados é processado por vez. Se você decidisse parar de imprimir a saída depois de Holly, nunca executaria nenhuma das operações nos outros elementos da sequência original. Embora vários estágios estejam envolvidos aqui, o processamento de dados em *streaming* como esse é eficiente e flexível. Independentemente da quantidade de dados de origem existentes, você não precisa saber sobre mais de um elemento em nenhum momento.

Este é o melhor cenário. Há momentos em que para buscar o primeiro resultado de uma consulta, você deve avaliar *todos* os dados da fonte. Já vimos um exemplo disso no capítulo anterior: o método `Enumerable.Reverse` precisa buscar todos os dados disponíveis para retornar o último elemento original como o primeiro elemento da sequência resultante. Isso faz do `Reverse` uma operação *de buffer*, que pode ter um efeito enorme na eficiência (ou mesmo na viabilidade) de sua operação geral.

<sup>1</sup> Os vários parâmetros envolvidos são verificados quanto à nulidade. É importante ter isso em mente se você implementar seus próprios operadores LINQ, como verá no capítulo 12.



**Figura 11.2** Diagrama de sequência da execução de uma expressão de consulta

Operação. Se você não puder ter todos os dados na memória de uma só vez, não poderá usar operações de buffer.

Assim como o streaming depende de qual operação você executa, algumas transformações ocorrem assim que você as chama, em vez de usar a execução adiada. Isso é chamado de *execução imediata*. De modo geral, as operações que retornam outra sequência (geralmente um `IEnumerable<T>` ou `IQueryable<T>`) usam execução adiada, enquanto as operações que retornam um único valor usam execução imediata.

As operações amplamente disponíveis no LINQ são conhecidas como operadores de consulta padrão — vamos dar uma breve olhada nelas agora.

#### OPERADORES DE CONSULTA PADRÃO

Os operadores de consulta padrão do LINQ são uma coleção de transformações cujos significados são bem entendido. Os provedores de LINQ são incentivados a implementar o maior número possível desses operadores, fazendo com que a implementação obedeça ao comportamento esperado. Isso é crucial para fornecer uma estrutura de consulta consistente em diversas fontes de dados. De

É claro que alguns provedores LINQ podem expor mais funcionalidades, e alguns dos operadores podem não mapear adequadamente para o domínio de destino do provedor, mas pelo menos o existe uma oportunidade para consistência.

**DETALHES ESPECÍFICOS DE IMPLEMENTAÇÃO DE OPERADORES PADRÃO** Só porque o operadores de consulta padrão têm significados gerais comuns, não significa eles funcionarão exatamente da mesma maneira para todas as implementações. Por exemplo, alguns provedores LINQ podem carregar os dados para uma consulta inteira assim que precisa do primeiro item - se você estiver acessando um serviço da web, isso pode ser perfeito senso. Da mesma forma, uma consulta que funciona no LINQ to Objects pode ter uma semântica sutilmente diferente no LINQ to SQL. Isso não significa que o LINQ falhou, apenas que você precisa considerar qual fonte de dados você está acessando ao escrever uma consulta. Ainda há uma enorme vantagem em ter um único conjunto de operadores de consulta e uma sintaxe de consulta consistente, mesmo que isso não seja uma panaceia.

C# 3 tem suporte para alguns dos operadores de consulta padrão integrados à linguagem por meio de expressões de consulta, mas você sempre pode optar por chamá-las manualmente se achar que torna o código mais claro. Você pode estar interessado em saber que o VB9 tem mais operadores presentes na linguagem; como sempre, há uma compensação entre a complexidade adicional de incluir um recurso na linguagem e os benefícios que esse recurso traz.

Pessoalmente, acho que a equipe C# fez um trabalho admirável; Sempre fui fã de um linguagem relativamente pequena com uma grande biblioteca por trás dela.

**SOBRECARGA DO OPERADOR** O termo operador é usado para descrever tanto a consulta operadores (métodos como Select e Where) e os operadores familiares como adição, igualdade e assim por diante. Normalmente deveria ser óbvio qual eu quero dizer a partir do contexto - se estou falando sobre LINQ, o operador quase sempre referem-se a um método usado como parte de uma consulta.

Você verá alguns desses operadores nos exemplos à medida que avançamos neste capítulo e o próximo, mas não pretendo fornecer um guia completo sobre eles aqui; este livro é principalmente sobre C#, não sobre todo o LINQ. Você não precisa conhecer todos os operadores em para ser produtivo no LINQ, mas é provável que sua experiência cresça com o tempo. O Apêndice A fornece uma breve descrição de cada um dos operadores de consulta padrão, e o MSDN fornece mais detalhes de cada sobrecarga específica. Quando você se deparar com um problema, verifique a lista: se parece que deveria haver um método integrado para ajudá-lo, provavelmente existe! Isso não é sempre é o caso - fundei o projeto de código aberto MoreLINQ para adicionar alguns operadores extras para LINQ to Objects (consulte <http://code.google.com/p/morelinq/>). Da mesma forma, o pacote Reactive Extensions (veja <http://mng.bz/R7ip>) tem acréscimos para o

pull do LINQ to Objects, bem como o modelo push que veremos mais tarde. Se os operadores padrão falharem, verifique ambos os projetos antes de construir sua própria solução. Porém, não é um desastre se você tiver que escrever seu próprio operador; Pode ser muito divertido. No capítulo 12 darei algumas dicas sobre esse assunto.

Tendo mencionado exemplos, é hora de apresentar o modelo de dados que a maior parte do restante do código de amostra deste capítulo usará.

### 11.1.2 Definindo o modelo de dados de amostra

Na seção 10.3.4 dei um breve exemplo de rastreamento de defeitos como um uso real para métodos de extensão e expressões lambda. Usaremos a mesma ideia para quase todo o código de exemplo neste capítulo – é um modelo bastante simples, mas que pode ser manipulado de muitas maneiras diferentes para fornecer informações úteis. O rastreamento de defeitos também é um domínio com o qual a maioria dos desenvolvedores profissionais está familiarizada, infelizmente.

Nosso cenário fictício é a SkeetySoft, uma pequena empresa de software com grandes ambições. Os fundadores decidiram criar um pacote de escritório, um media player e um aplicativo de mensagens instantâneas. Afinal, não existem grandes players nesses mercados, não é mesmo?

O departamento de desenvolvimento da SkeetySoft consiste em cinco pessoas: dois desenvolvedores (Deborah e Darren), dois testadores (Tara e Tim) e uma gerente (Mary). Atualmente há um único cliente: Colin. Os produtos mencionados acima são SkeetyOffice, SkeetyMediaPlayer e SkeetyTalk, respectivamente.<sup>2</sup> Veremos os defeitos registrados durante maio de 2013, usando o modelo de dados mostrado na Figura 11.3.

Como você pode ver, não há muitos dados sendo registrados aqui. Em particular, não há histórico real dos defeitos, mas há o suficiente aqui para permitir que você trabalhe com os recursos de expressão de consulta do C# 3.

Para os fins deste capítulo, todos os dados são armazenados na memória. Você tem uma classe chamada SampleData com propriedades AllDefects, AllUsers, AllProjects e AllSubscriptions, cada uma retornando um tipo apropriado de IEnumerable<T>. As propriedades Start e End retornam instâncias de DateTime para o início e o final de maio, respectivamente, e há classes aninhadas Users e Projects em SampleData para fornecer acesso fácil a um determinado usuário ou projeto. O único tipo que pode não ser imediatamente óbvio é NotificationSubscription; a ideia por trás disso é enviar um e-mail para o endereço especificado sempre que um defeito for criado ou alterado no projeto relevante.

Existem 41 defeitos nos dados de amostra, criados usando inicializadores de objeto C# 3. Todos do código está disponível no site do livro, junto com os dados de amostra.

Agora que as preliminares foram resolvidas, vamos começar com algumas dúvidas!

<sup>2</sup> O departamento de marketing da SkeetySoft não é particularmente criativo.

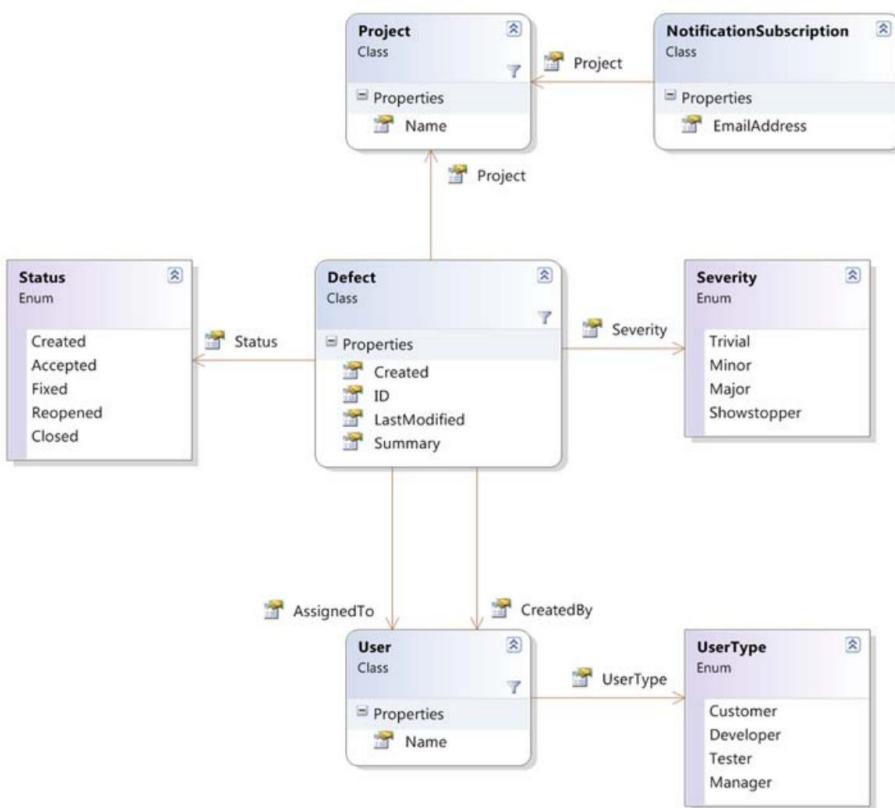


Figura 11.3 Diagrama de classes do modelo de dados de defeitos SkeetySoft

## 11.2 Começos simples: selecionando elementos

Já discutimos alguns conceitos gerais do LINQ — apresentarei os conceitos específicos do C# 3 à medida que surgirem no decorrer do capítulo. Começaremos com uma consulta simples (ainda mais simples do que a que você viu anteriormente) e avançaremos para algumas mais complicadas, não apenas construindo sua compreensão sobre o que o compilador C# 3 está fazendo, mas também ensinando como ler Código LINQ.

Todos os exemplos seguirão o padrão de definir uma consulta e depois imprimir os resultados no console. Não analisaremos consultas vinculadas a grades de dados ou algo assim — tudo isso é importante, mas não é diretamente relevante para o aprendizado do C# 3.

Você pode usar uma expressão simples que imprima todos os usuários como ponto de partida para examinar o que o compilador está fazendo nos bastidores e aprender sobre variáveis de intervalo.

### 11.2.1 Começando com uma fonte e terminando com uma seleção

Cada expressão de consulta em C# 3 começa da mesma maneira – informando a origem de uma sequência de dados:

do elemento na fonte

A parte do elemento é apenas um identificador, com um nome de tipo opcional antes dele. Na maioria das vezes você não precisará do nome do tipo e não teremos um para o primeiro exemplo.

A parte de origem é uma expressão normal. Muitas coisas diferentes podem acontecer depois da primeira cláusula, mas mais cedo ou mais tarde você sempre termina com uma cláusula select ou uma cláusula group .

Começaremos com uma cláusula select para manter as coisas simples e agradáveis. A sintaxe para um A cláusula select também é fácil:

selecione expressão

A cláusula select é conhecida como *projeção*.

Combinar os dois e usar uma expressão trivial resulta em uma consulta simples (e praticamente inútil), conforme mostrado na listagem a seguir.

#### Listagem 11.1 Consulta trivial para imprimir a lista de usuários

```
var query = do usuário em SampleData.AllUsers  
        selecionar o usuário;  
foreach (var usuário na consulta) {  
  
    Console.WriteLine(usuário);  
}
```

A expressão de consulta é a parte em negrito. Substituí ToString para cada uma das entidades no modelo, portanto os resultados da listagem 11.1 são os seguintes:

```
Usuário: Tim Trotter (testador)  
Usuário: Tara Tutu (testadora)  
Usuário: Deborah Denton (desenvolvedora)  
Usuário: Darren Dahlia (desenvolvedor)  
Usuário: Mary Malcop (Gerente)  
Usuário: Colin Carton (Cliente)
```

Você pode estar se perguntando o quanto útil isso é como exemplo; afinal, você poderia simplesmente usar SampleData.AllUsers diretamente na instrução foreach . Mas usaremos essa expressão de consulta – por mais trivial que seja – para introduzir dois novos conceitos. Primeiro veremos a natureza geral do processo de tradução que o compilador usa quando encontra uma expressão de consulta e depois discutiremos variáveis de intervalo.

### 11.2.2 Traduções do compilador como base para expressões de consulta

O suporte a expressões de consulta C# 3 é baseado no compilador que traduz expressões de consulta em código C# normal. Ele faz isso de uma maneira mecânica que não tenta entender o código, aplicar inferência de tipo, verificar a validade de chamadas de método ou realizar qualquer atividade normal de um compilador. Tudo isso é feito mais tarde, após a tradução. De muitas maneiras, esta primeira fase pode ser considerada uma etapa de pré-processador.

O compilador traduz a listagem 11.1 no código a seguir antes de fazer a compilação *real*.

**Listagem 11.2 A expressão de consulta da Listagem 11.1 traduzida em uma chamada de método**

```
var consulta = SampleData.AllUsers.Select(usuário => usuário);

foreach (var usuário na consulta) {

    Console.WriteLine(usuário);
}
```

O compilador C# 3 traduz a expressão de consulta *exatamente* neste código antes de compilá-la adequadamente. Em particular, ele não pressupõe que deva usar Enumerable .Select ou que List<T> conterá um método chamado Select. Ele apenas traduz o código e então permite que a próxima fase da compilação trate de encontrar um método apropriado - seja como um membro simples ou como um método de extensão.<sup>3</sup> O parâmetro pode ser um tipo delegado adequado ou um Expression<T> para um tipo apropriado T.

É aqui que é importante que as expressões lambda possam ser convertidas em instâncias delegadas e em árvores de expressão. Todos os exemplos neste capítulo usarão delegados, mas você verá como as árvores de expressão são usadas quando examinarmos os outros provedores LINQ no capítulo 12. Quando eu apresentar as assinaturas de alguns dos métodos chamados pelo compilador posteriormente, lembre-se de que esses são apenas aqueles chamados em LINQ to Objects - sempre que o parâmetro for do tipo delegado (o que a maioria deles é), o compilador usará uma expressão lambda como argumento e então tentará encontrar um método com uma assinatura adequada.

Também é importante lembrar que sempre que uma variável normal (como uma variável local dentro do método) aparecer dentro de uma expressão lambda após a tradução ter sido realizada, ela se tornará uma variável capturada da mesma forma que você viu no capítulo 5. Esse é um comportamento normal da expressão lambda, mas, a menos que você entenda quais variáveis serão capturadas, poderá facilmente ficar confuso com os resultados de suas consultas.

A especificação da linguagem fornece detalhes do *padrão de expressão de consulta*, que deve ser implementado para que todas as expressões de consulta funcionem, mas isso não é definido como uma interface como você poderia esperar. Isso faz muito sentido: permite que o LINQ seja aplicado a interfaces como IEnumerable<T> usando métodos de extensão. Este capítulo aborda cada elemento do padrão de expressão de consulta, um de cada vez. Se você quiser ver exatamente como a especificação da linguagem define cada tradução, consulte a seção 7.16 (“Expressões de Consulta”).

---

<sup>3</sup> É ainda mais geral do que isso: o compilador não exige que Select seja um método ou SampleData .AllUsers seja um acesso de propriedade. Contanto que o código traduzido seja compilado, tudo ficará bem. Em quase todos os casos sensatos, você acessará métodos padrão ou de extensão, mas tenho uma postagem no blog com algumas consultas particularmente estranhas com as quais o compilador está perfeitamente satisfeito (consulte <http://mng.bz/7E3i>). Não achei que consultas como essa fossem úteis na prática, mas gosto deste exemplo como uma forma de deixar claro o quão mecânico é o processo de tradução e como ele não se importa com o significado do código *traduzido*.

A Listagem 11.3 ilustra como funciona a tradução do compilador: ela fornece uma implementação fictícia de Select e Where, com Select como um método de instância normal e Where como um método de extensão. Nossa expressão de consulta simples original continha apenas uma cláusula select , mas esta inclui uma cláusula where para mostrar os dois tipos de métodos em uso. Esta é uma listagem completa e não um trecho, pois os métodos de extensão só podem ser declarados em classes estáticas de nível superior.

#### Listagem 11.3 Métodos de chamada de tradução do compilador em uma implementação LINQ fictícia

Extensões de classe estática

```

{
    public static Dummy<T> Onde<T>(este Dummy<T> fictício,
                                         Func<T,bool> predicado)
    {
        Console.WriteLine("Onde chamado"); manequim de
        retorno;
    }
}

class Dummy<T> {

    public Dummy<U> Select<U>(Func<T,U> seletor) {
        Console.WriteLine("Selecionar chamado"); return new
        Dummy<U>();
    }
}

class TraduçãoExemplo {

    static void Principal() {

        var fonte = new Dummy<string>(); var query = do fictício
        na fonte
            onde dummy.ToString() == "Ignorado" selecione "Qualquer
            coisa";
    }
}

```

**A** Declara onde  
Método de extensão B

**C** Declara Selecionar  
Método de instância C

**D** Cria fonte a ser  
consultada

**E** Chama métodos por  
meio de uma consulta  
Expressão E

Quando você executa a listagem 11.3, ela imprime Onde chamado e, em seguida, Selecionar chamado, exatamente como seria de esperar, porque a expressão de consulta foi traduzida neste código:

```

var consulta = source.Where(dummy => dummy.ToString() == "Ignorado")
                     .Select(dummy => "Qualquer coisa");

```

Claro, você não está fazendo nenhuma consulta ou transformação aqui, mas mostra como o compilador está traduzindo a expressão de consulta. Se você está confuso sobre por que a expressão lambda na chamada Select retorna "Anything" em vez de apenas fictício, é porque uma projeção de fictício (que é uma projeção sem fazer nada) seria removida pelo compilador neste caso específico. Veremos isso na seção 11.3.2, mas no momento a ideia importante é o tipo geral de tradução envolvida. Você só precisa aprender quais traduções o compilador C# usará e então poderá fazer qualquer consulta

expressão, converta-a no formato que não usa expressões de consulta e, em seguida, procure no que está fazendo desse ponto de vista.

Observe que você não implementa `IEnumerable<T>` em `Dummy<T>`. A tradução das expressões de consulta ao código normal não depende disso, mas na prática a maioria Os provedores LINQ expõem os dados como `IEnumerable<T>` ou `IQueryable<T>` (que veremos no capítulo 12). O fato de a tradução não depender de nenhum tipo específico, mas apenas dos nomes dos métodos e parâmetros, é uma espécie de forma de digitação em tempo de compilação. Isso é semelhante ao modo como os inicializadores de coleção apresentados no capítulo 8 encontram um método público chamado `Add` usando resolução de sobrecarga normal.

em vez de usar uma interface contendo um método `Add` com uma assinatura específica.

As expressões de consulta levam essa ideia um passo além – a tradução ocorre no início do processo de compilação para permitir que o compilador escolha métodos de instância ou métodos de extensão. Você poderia até considerar a tradução como o trabalho de um mecanismo de pré-processamento separado.

Você pode pensar que estou falando muito sobre isso, mas tudo faz parte da remoção da névoa isso às vezes envolve o LINQ. Se você reescrever uma expressão de consulta como uma série de métodos chamadas, fazendo efetivamente o que o compilador teria feito, você não alterará o desempenho e sua consulta não se comportará de maneira diferente. São apenas duas maneiras diferentes de representando o mesmo código.

**POR QUE FROM...WHERE...SELECT EM VEZ DE SELECT...FROM...WHERE?** Muitos desenvolvedores acham a ordem das cláusulas nas expressões de consulta confusa para começar com. Parece exatamente com SQL – exceto de trás para frente. Se você olhar novamente para a tradução em métodos, verá a principal razão por trás disso. A expressão de consulta é processada na mesma ordem em que foi escrita: você começa com uma fonte em cláusula `from`, filtre-a na cláusula `where` e projete-a na cláusula cláusula de seleção . Outra maneira de ver isso é considerar os diagramas ao longo deste capítulo. Os dados fluem de cima para baixo e as caixas aparecem no diagrama na mesma ordem que suas cláusulas correspondentes aparecem na expressão de consulta. Depois de superar qualquer desconforto inicial devido ao desconhecimento, você pode achar essa abordagem atraente – eu sei que sim. Você pode até mesmo faça a pergunta equivalente sobre SQL.

Agora você sabe que está envolvida uma tradução em nível de origem, mas há outra questão crucial conceito para entender antes de prosseguirmos.

### 11.2.3 Variáveis de amplitude e projeções não triviais

Vejamos a expressão de consulta original deste capítulo com mais detalhes. Nós não temos examinou o identificador na cláusula `from` ou a expressão na cláusula `select` .

A Figura 11.4 mostra a expressão de consulta novamente, com cada parte rotulada para explicar sua propósito.

As palavras-chave contextuais são fáceis de explicar – elas especificam ao compilador o que você quero fazer com os dados. Da mesma forma, a expressão de origem é uma expressão C# normal— uma propriedade neste caso, mas poderia facilmente ter sido uma chamada de método ou uma variável.

As partes complicadas são a declaração da variável de intervalo e a expressão de projeção. Variáveis de intervalo não são como qualquer outro tipo de variável. De certa forma, elas não são variáveis! Eles estão disponíveis apenas em expressões de consulta e estão efetivamente presentes para propagar o contexto de uma expressão para outra. Eles representam um elemento de uma sequência específica por vez e são usados na tradução do compilador para permitir que outras expressões sejam facilmente transformadas em expressões lambda.

Você já viu que a expressão de consulta original foi transformada em

```
SampleData.AllUsers.Select(usuário => usuário)
```

O lado esquerdo da expressão lambda – a parte que fornece o nome do parâmetro – vem da declaração da variável de intervalo. O lado direito vem da cláusula select. A tradução é tão simples quanto isso (neste caso). Tudo funciona bem porque o mesmo nome é usado em ambos os lados.

Suponha que você tenha escrito a expressão de consulta assim:

do usuário em SampleData.AllUsers selecione a pessoa

Nesse caso, a versão traduzida teria sido

```
SampleData.AllUsers.Select(usuário => pessoa)
```

Nesse ponto, o compilador teria reclamado porque não saberia a que pessoa se referia.

Agora que você sabe como o processo é simples, fica mais fácil entender uma expressão de consulta que possui uma projeção um pouco mais complicada. A listagem a seguir imprime apenas os nomes de nossos usuários.

#### Listagem 11.4 Consulta selecionando apenas os nomes dos usuários

```
IEnumerable<string> consulta = do usuário em SampleData.AllUsers selecione user.Name;

foreach (nome da string na consulta) {
    Console.WriteLine(nome);
}
```

Desta vez você está usando user.Name como projeção e o resultado é uma sequência de strings, não de objetos User. (Usei uma variável digitada explicitamente para enfatizar esse ponto.) A tradução da expressão de consulta segue as mesmas regras de antes e se torna

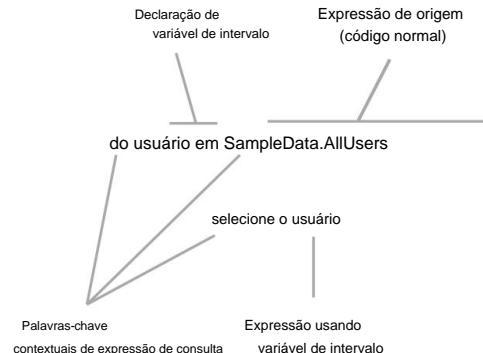


Figura 11.4 Uma expressão de consulta simples dividida em suas partes constituintes

```
SampleData.AllUsers.Select(usuário => usuário.Nome)
```

O compilador permite isso, porque o método de extensão Select escolhido de Enumerable tem esta assinatura:<sup>4</sup>

```
estático IEnumerable<TResult> Select<TSource,TResult>
    (esta fonte IEnumerable<TSource>,
     Seletor Func<TSource,TResult>)
```

A inferência de tipo descrita no capítulo 9 entra em ação, convertendo a expressão lambda em Func<TSource,TResult>. Primeiro infere que TSource é User devido ao tipo SampleData.AllUsers. Nesse ponto, ele conhece o tipo de parâmetro da expressão lambda, para que possa resolver user.Name como uma expressão de acesso de propriedade que retorna o tipo string, inferindo assim que TResult é string. É por isso que as expressões lambda permitem parâmetros digitados implicitamente e por que existem regras de inferência de tipos tão complicadas; estas são as engrenagens e pistões do motor LINQ .

**POR QUE VOCÊ PRECISA SABER TUDO ISSO?** Você quase pode ignorar o que está acontecendo com variáveis de intervalo na maior parte do tempo. Você pode ter visto muitas, muitas dúvidas e entendido o que elas alcançam, sem nunca saber o que está acontecendo nos bastidores. Tudo bem quando as coisas estão funcionando (como costumam acontecer com exemplos em tutoriais), mas quando as coisas dão errado, vale a pena conhecer os detalhes. Se você tiver uma expressão de consulta que não será compilada porque o compilador está reclamando que não conhece um identificador específico, você deve examinar as variáveis de intervalo envolvidas.

Até agora, vimos apenas variáveis de intervalo digitadas implicitamente. O que acontece quando você inclui um tipo na declaração? A resposta está nos operadores de consulta padrão Cast e OfType .

#### 11.2.4 Cast, OfType e variáveis de intervalo digitadas explicitamente

Na maioria das vezes, variáveis de intervalo podem ser digitadas implicitamente; é provável que você esteja trabalhando com coleções genéricas nas quais o tipo especificado é tudo o que você precisa. E se não fosse esse o caso? E se você tivesse um ArrayList, ou talvez um objeto[] no qual desejasse realizar uma consulta? Seria uma pena se o LINQ não pudesse ser aplicado nessas situações. Felizmente, existem dois operadores de consulta padrão que vêm em socorro: Cast e OfType. Somente Cast é suportado diretamente pela sintaxe da expressão de consulta, mas veremos ambos nesta seção.

Os dois operadores são semelhantes: ambos pegam uma sequência arbitrária não digitada (são métodos de extensão no tipo IEnumerable não genérico ) e retornam uma sequência fortemente tipada. Cast faz isso convertendo cada elemento para o tipo de destino (e falhando em qualquer elemento que não seja do tipo correto), e OfType faz um teste primeiro, ignorando quaisquer elementos do tipo errado.

---

<sup>4</sup> Para permitir que todas as assinaturas dos métodos deste capítulo caibam na página impressa, omiti o modificador público . Na realidade, eles são todos públicos.

A listagem a seguir demonstra esses dois operadores, usados como métodos de extensão simples de Enumerable. Para variar, não usaremos o sistema de defeitos SkeetySoft para dados de amostra – afinal, tudo isso é fortemente tipado! Em vez disso, usaremos dois objetos ArrayList .

#### Listagem 11.5 Usando Cast e OfType para trabalhar com coleções de tipo fraco

```
Lista ArrayList = new ArrayList { "Primeiro", "Segundo", "Terceiro" }; IEnumerable<string> strings =
list.Cast<string>(); foreach (item de string em strings)

{
    Console.WriteLine(item);
}

lista = new ArrayList { 1, "não é um int", 2, 3 }; IEnumerable<int> ints =
list.OfType<int>(); foreach (item int em ints)

{
    Console.WriteLine(item);
}
```

A primeira lista contém apenas strings, portanto é seguro usar Cast<string> para obter uma sequência de strings. A segunda lista tem conteúdo misto, então para buscar apenas os números inteiros dela você usa OfType<int>. Se você tivesse usado Cast<int> na segunda lista, uma exceção teria sido lançada quando você tentasse converter “not an int” em int. Observe que isso só teria acontecido depois que você retornasse 1 – ambos os operadores transmitem seus dados, convertendo elementos à medida que os buscam.

#### APENAS CONVERSÕES DE IDENTIDADE, REFERÊNCIA E UNBOXING O

comportamento do Cast mudou sutilmente no .NET 3.5 SP1. No .NET 3.5 original, ele realizaria mais conversões, portanto, usar Cast<int> em um List<short> converteria cada short em um int à medida que fosse obtido. No .NET 3.5 service pack 1 e em todas as versões posteriores, isso gerará uma exceção. Se você quiser qualquer conversão diferente de uma conversão de referência ou de unboxing (ou a conversão de identidade sem operação), use uma projeção Select . OfType também executa essas conversões, mas não lança uma exceção se elas falharem.

Quando você introduz uma variável de intervalo com um tipo explícito, o compilador usa uma chamada para Cast para garantir que a sequência usada pelo restante da expressão de consulta seja do tipo apropriado. A listagem a seguir mostra isso, com uma projeção usando o método Substring para provar que a sequência gerada pela cláusula from é uma sequência de strings.

#### Listagem 11.6 Usando uma variável de intervalo digitada explicitamente para chamar Cast automaticamente

```
lista ArrayList = new ArrayList { "Primeiro", "Segundo", "Terceiro" }; var strings = da entrada de string
na lista
    selecione entrada.Substring(0, 3);
foreach (string começa em strings)
```

```
{
    Console.WriteLine(iniciar);
}
```

A saída da listagem 11.6 é Fir, Sec, Thi, mas o que é mais interessante é a expressão de consulta traduzida:

```
list.Cast<string>().Select(entrada => entrada.Substring(0,3));
```

Sem a conversão, você não seria capaz de chamar `Select`, porque o método de extensão é definido apenas para `IEnumerable<T>` em vez de `IEnumerable`. Mesmo quando você estiver usando uma coleção fortemente tipada, você ainda pode querer usar uma variável de intervalo explicitamente digitada. Por exemplo, você pode ter uma coleção definida como `List<ISomeInterface>` mas sabe que todos os elementos são instâncias de `MyImplementation`. Usar uma variável de intervalo com um tipo explícito de `MyImplementation` permite acessar todos os membros de `MyImplementation` sem inserir manualmente conversões em todo o código.

Cobrimos muitos aspectos conceituais importantes até agora, embora não obtiveram resultados impressionantes. Para recapitular brevemente os pontos mais importantes:

- ÿ LINQ é baseado em sequências de dados, que são transmitidas sempre que possível. ÿ
- Criar uma consulta geralmente não a executa; a maioria das operações usa diferido execução.
- ÿ Expressões de consulta em C# 3 envolvem uma fase de pré-processamento que converte a expressão em C# normal, que é então compilado adequadamente com todas as regras normais de inferência de tipo, sobrecarga, expressões lambda e assim por diante. ÿ As variáveis declaradas nas expressões de consulta não agem como qualquer outra coisa; são variáveis de intervalo que permitem fazer referência a dados de forma consistente na expressão de consulta.

Eu sei que há muitas informações um tanto abstratas para serem assimiladas.

preocupe-se se você está começando a se perguntar se o LINQ vale todo esse trabalho. Eu prometo a você que é. Com grande parte do trabalho de base resolvido, podemos começar a fazer coisas genuinamente úteis – como filtrar dados e depois ordená-los.

### 11.3 Filtrando e ordenando uma sequência

Você pode se surpreender ao saber que filtrar e ordenar são duas das operações mais simples de explicar em termos de traduções do compilador. Isso ocorre porque eles sempre retornam uma sequência com o mesmo tipo de elemento de sua entrada, o que significa que você não precisa se preocupar com a introdução de novas variáveis de intervalo. Também ajuda o fato de você ter visto os métodos de extensão correspondentes no capítulo 10.

#### 11.3.1 Filtrando usando uma cláusula where

É extremamente fácil entender a cláusula `where`. A sintaxe é apenas onde expressão de filtro

O compilador traduz isso em uma chamada ao método Where com uma expressão lambda, que usa a variável de intervalo apropriada como parâmetro e a expressão de filtro como corpo. A expressão de filtro é aplicada como um predicado a cada elemento do fluxo de dados de entrada, e somente aqueles que retornam verdadeiro estão presentes no resultado.

seqüência.

O uso de múltiplas cláusulas where resulta em múltiplas chamadas Where encadeadas – apenas os elementos que correspondem a *todos* os predicados fazem parte da sequência resultante. A listagem a seguir demonstra uma expressão de consulta que localiza todos os defeitos abertos atribuídos a Tim.

#### Listagem 11.7 Expressão de consulta usando múltiplas cláusulas where

```
Usuário tim = SampleData.Users.TesterTim;

var query = do defeito em SampleData.AllDefects onde defect.Status != Status.Closed onde
    defect.AssignedTo == tim select defect.Summary;

foreach (var resumo na consulta)
{
    Console.WriteLine(resumo);
}
```

A expressão de consulta na listagem 11.7 é traduzida assim:

```
SampleData.AllDefects.Where (defeito => defeito.Status != Status.Closed)
    .Where(defeito => defeito.AssignedTo == tim)
    .Select(defeito => defeito.Resumo)
```

A saída da listagem 11.7 é a seguinte:

```
A instalação é lenta
As legendas só funcionam em galês
O botão Play aponta para o lado errado
Webcam me faz parecer careca
A rede está saturada ao reproduzir arquivo WAV
```

Claro, você poderia escrever uma única cláusula where que combinasse as duas condições como uma alternativa ao uso de múltiplas cláusulas where. Em alguns casos, isso pode melhorar o desempenho, mas vale a pena ter em mente também a legibilidade da expressão da consulta, e isso provavelmente será bastante subjetivo. Minha inclinação pessoal é combinar condições que estão logicamente relacionadas, mas manter outras separadas. Neste caso, ambas as partes da expressão tratam diretamente de um defeito (pois isso é tudo que nossa sequência contém), então seria razoável combiná-las. Como antes, vale a pena tentar as duas formas para ver qual fica mais clara.

Dentro de instantes você começará a aplicar algumas regras de ordenação à consulta, mas primeiro vamos deve observar um pequeno detalhe relacionado à cláusula select.

### 11.3.2 Expressões de consulta degeneradas

Embora tenhamos uma tradução bastante simples para trabalhar, vamos revisitar um ponto que abordei anteriormente na seção 11.2.2, quando apresentei pela primeira vez as traduções do compilador. Até agora, todas as nossas expressões de consulta traduzidas incluíram uma chamada para Select. O que acontece se

a cláusula select não faz nada, retornando efetivamente a mesma sequência fornecida?

A resposta é que o compilador remove essa chamada para Select, mas somente se houver outras operações sendo executadas na expressão de consulta.

Por exemplo, a seguinte expressão de consulta apenas seleciona todos os defeitos no sistema:

do defeito em SampleData.AllDefects selecione o defeito

Isso é conhecido como *expressão de consulta degenerada*. O compilador gera deliberadamente uma chamada para Select mesmo que pareça não fazer nada:

```
SampleData.AllDefects.Select(defeito => defeito)
```

Porém , há uma grande diferença entre isso e usar SampleData.AllDefects como uma expressão simples. Os itens retornados pelas duas sequências são iguais, mas o resultado do método Select é apenas a sequência de itens, não a fonte em si. O resultado de uma expressão de consulta nunca é o mesmo objeto que os dados de origem, a menos que o provedor LINQ tenha sido mal codificado. Isso pode ser importante do ponto de vista da integridade dos dados: um provedor pode retornar um objeto de resultado mutável, sabendo que as alterações na sequência de dados retornada não afetarão o mestre, mesmo diante de uma consulta degenerada.

Quando outras operações estão envolvidas, não há necessidade de o compilador manter cláusulas de seleção não operacionais . Por exemplo, suponha que você altere a expressão de consulta na listagem 11.7 para selecionar todo o defeito em vez de apenas o nome:

```
do defeito em SampleData.AllDefects onde defeito.Status!
= Status.Closed
onde defeito.AssignedTo == SampleData.Users.TesterTim selecione defeito
```

Agora você não precisa da chamada final para Select, então o código traduzido é apenas este:

```
SampleData.AllDefects.Where (defeito => defeito.Status! = Status.Closed)
    .Where(defeito => defeito.AssignedTo == tim)
```

Essas regras raramente atrapalham quando você escreve expressões de consulta, mas podem causar confusão se você descompilar o código com uma ferramenta como o Reflector — pode ser surpreendente ver a chamada Select desaparecer sem motivo aparente .

Com esse conhecimento em mãos, é hora de melhorar a consulta para que você saiba o que Tim deve trabalhar a seguir.

### 11.3.3 Pedidos usando uma cláusula orderby

Não é incomum que desenvolvedores e testadores sejam solicitados a trabalhar nos defeitos mais críticos antes de resolverem os mais triviais. Você pode usar uma consulta simples para informar a Tim a ordem em que ele deve resolver os defeitos em aberto atribuídos a ele. A listagem a seguir faz exatamente isso usando uma cláusula orderby , imprimindo todos os detalhes dos defeitos em ordem decrescente de prioridade.

### Listagem 11.8 Classificando pela gravidade de um defeito, de alta para baixa prioridade

```
Usuário tim = SampleData.Users.TesterTim;

var query = do defeito em SampleData.AllDefects onde defect.Status != Status.Closed

    onde defeito.AssignedTo == tim orderby defeito.Severity
    descendente selecione defeito;

foreach (var defeito na consulta) {

    Console.WriteLine("{0}: {1}", defeito.Severidade, defeito.Summary);
}
```

O resultado da listagem 11.8 mostra que você classificou os resultados adequadamente:

Showstopper: Webcam me faz parecer careca  
 Principal: as legendas só funcionam em galês  
 Principal: o botão Play aponta para o lado errado  
 Menor: a rede está saturada ao reproduzir arquivo WAV  
 Trivial: a instalação é lenta

Você tem dois defeitos principais. Em que ordem eles devem ser abordados? Atualmente, nenhuma ordem clara está envolvida.

Vamos alterar a consulta para que, após classificar por gravidade em ordem decrescente, você classifique pela hora da última modificação em ordem *crescente*. Isso significa que a Tim testará os defeitos que foram corrigidos há muito tempo, antes daqueles resolvidos mais recentemente. Isso requer apenas uma expressão extra na cláusula `orderby`, conforme mostrado na listagem a seguir.

### Listagem 11.9 Ordenação por gravidade e hora da última modificação

```
Usuário tim = SampleData.Users.TesterTim;

var query = do defeito em SampleData.AllDefects onde defect.Status != Status.Closed

    onde defeito.AssignedTo == tim orderby defeito.Severidade
    descendente, defeito.LastModified selecione defeito;

foreach (var defeito na consulta) {

    Console.WriteLine("{0}: {1} ({2:d})", defeito.Severidade, defeito.Summary,
        defeito.LastModified);
}
```

Os resultados da listagem 11.9 são mostrados aqui. Observe como a ordem dos dois defeitos principais foi invertida:

Showstopper: Webcam me deixa careca (27/05/2013)  
 Major: Botão Play aponta para o lado errado (**17/05/2013**)  
 Principal: legendas só funcionam em galês (**23/05/2013**)  
 Menor: a rede fica saturada ao reproduzir arquivo WAV (31/05/2013)  
 Trivial: A instalação é lenta (15/05/2013)

Esta é a aparência da expressão de consulta, mas o que o compilador faz? Ele simplesmente chama os métodos OrderBy e ThenBy (ou OrderByDescendente/ThenByDescendente para ordens decrescentes). Sua expressão de consulta é traduzida assim:

```
SampleData.AllDefects.Where (defeito => defeito.Status! = Status.Closed)
    .Where(defeito => defeito.AssignedTo == tim)
    .OrderByDecrescente(defeito => defeito.Severidade)
    .ThenBy(defeito => defeito.LastModified)
```

Agora que você viu um exemplo, podemos dar uma olhada na sintaxe geral das cláusulas orderby. Eles são basicamente a palavra-chave contextual orderby seguida por uma ou mais ordenações. Uma ordenação é apenas uma expressão (que pode usar variáveis de intervalo) opcionalmente seguida de crescente ou decrescente, que têm significados óbvios. (A ordem padrão é crescente.) A tradução para a ordem primária é uma chamada para OrderBy ou OrderByDescendente, seguida por tantas chamadas para ThenBy ou ThenBy-Descendente quantas forem as ordenações subsequentes.

A diferença entre OrderBy e ThenBy é simples: OrderBy assume que tem controle primário sobre o pedido, enquanto ThenBy entende que é subserviente a um ou mais pedidos anteriores. Para LINQ to Objects, ThenBy é definido apenas como um método de extensão para IOrderedEnumerable<T>, que é o tipo retornado por OrderBy (e pelo próprio ThenBy, para permitir encadeamento adicional).

É importante observar que, embora você possa usar várias cláusulas orderby, cada uma começará com sua própria cláusula OrderBy ou OrderByDescendente, o que significa que a última vencerá efetivamente. Ainda estou para ver uma situação em que isso seja útil, a menos que você faça outra coisa na consulta entre as cláusulas orderby; você quase sempre deve usar uma única cláusula contendo vários pedidos.

Conforme observado no capítulo 10, aplicar uma ordenação requer que todos os dados sejam carregados (pelo menos para LINQ to Objects) — você não pode ordenar uma sequência infinita, por exemplo. Esperamos que a razão para isso seja óbvia: você não sabe qual elemento deve vir no início da sequência resultante até ver todos os elementos.

Estamos na metade do aprendizado sobre expressões de consulta e você pode se surpreender por ainda não termos analisado nenhuma junção. Obviamente eles são importantes no LINQ, assim como são importantes no SQL, mas também são complicados. Prometo que chegaremos a eles no devido tempo, mas para introduzir apenas um novo conceito de cada vez, desviaremosmos primeiro das cláusulas let. Dessa forma, podemos discutir identificadores transparentes antes de fazermos junções.

## 11.4 Cláusulas Let e identificadores transparentes

A maioria dos demais operadores que ainda precisamos examinar envolve *identificadores transparentes*. Assim como as variáveis de intervalo, você pode se dar bem sem entender os identificadores transparentes se quiser apenas ter uma compreensão bastante superficial das expressões de consulta. Porém, se você comprou este livro, provavelmente desejará conhecer C# em um nível mais profundo, o que (entre outras coisas) permitirá que você veja os erros de compilação de frente e saiba do que eles estão falando.

Você não precisa saber *tudo* sobre identificadores transparentes, mas vou te ensinar o suficiente para que, se você encontrar um na especificação da linguagem, não tenha vontade de correr e se esconder. Você também entenderá por que eles são necessários – e é aí que um exemplo será útil. A cláusula let é a transformação mais simples disponível que utiliza identificadores transparentes.

### 11.4.1 Apresentando um cálculo intermediário com let

Uma cláusula let introduz uma nova variável de intervalo com um valor que pode ser baseado em outras variáveis de intervalo. A sintaxe é tão fácil quanto uma torta:

deixe identificador = expressão

Para explicar este operador em termos que não utilizam quaisquer outros operadores complicados, recorrerei a um exemplo muito artificial. Suspenda sua descrença e imagine que encontrar o comprimento de uma corda é uma operação cara. Agora imagine que você tem um requisito de sistema completamente bizarro para ordenar seus usuários pelo comprimento de seus nomes e depois exibir o nome e seu comprimento. Sim, eu sei que é improvável. A listagem a seguir mostra uma maneira de fazer isso sem uma cláusula let .

#### Listagem 11.10 Classificando pelo comprimento dos nomes de usuários sem uma cláusula let

```
var query = do usuário em SampleData.AllUsers
    ordenar por usuário.Nome.Comprimento
    selecione usuário.Nome;

foreach (nome da var na consulta)
{
    Console.WriteLine("{0}: {1}", nome.Comprimento, nome);
}
```

Isso funciona bem, mas usa a temida propriedade Length duas vezes – uma para classificar os usuários e outra no lado da exibição. Certamente nem mesmo o supercomputador mais rápido conseguiria encontrar *duas vezes o comprimento de seis cordas!* Não, você precisa evitar esse cálculo redundante.

Você pode fazer isso com a cláusula let , que avalia uma expressão e a introduz como uma nova variável de intervalo. O código a seguir obtém o mesmo resultado da listagem 11.10, mas usa a propriedade Length apenas uma vez por usuário.

#### Listagem 11.11 Usando uma cláusula let para remover cálculos redundantes

```
var query = do usuário em SampleData.AllUsers
    deixe comprimento = user.Name.Length
    ordenar por comprimento
    selecione novo { Nome = usuário.Nome, Comprimento = comprimento };

foreach (entrada var na consulta) {

    Console.WriteLine("{0}: {1}", entrada.Comprimento, entrada.Nome);
}
```

A Listagem 11.11 introduz uma nova variável de intervalo chamada `length`, que contém o comprimento do nome do usuário (para o usuário atual na sequência original). Você então usa essa nova variável de intervalo para classificação e projeção no final. Você já identificou o problema? Você precisa usar duas variáveis de intervalo, mas a expressão lambda passada para `Select` leva apenas um parâmetro! É aqui que os identificadores transparentes entram em cena.

#### 11.4.2 Identificadores transparentes

Na listagem 11.11 você tem duas variáveis de intervalo envolvidas na projeção final, mas o método `Select` atua apenas em uma única sequência.

Como você pode combinar as variáveis de intervalo?

A resposta é criar um tipo anônimo que contenha ambas as variáveis e, em seguida, aplicar uma tradução inteligente para fazer parecer que você realmente possui dois parâmetros para as cláusulas `select` e `orderby`. A Figura 11.5 mostra as sequências envolvidas.

A cláusula `let` atinge seus objetivos usando outra chamada para `Select`, criando um tipo anônimo para a sequência resultante e criando efetivamente um novo intervalo

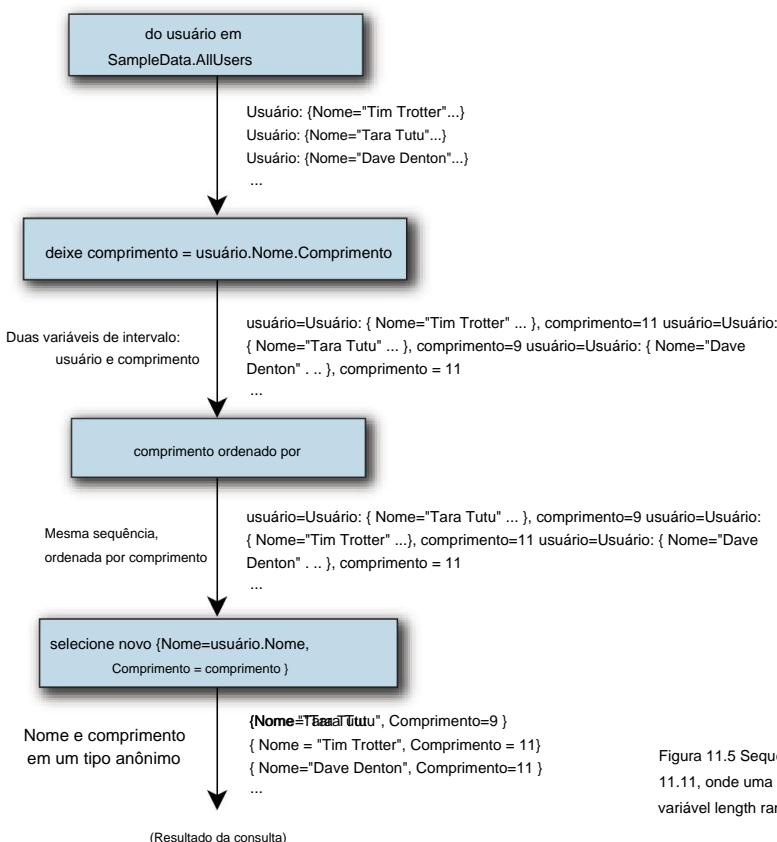


Figura 11.5 Sequências envolvidas na listagem 11.11, onde uma cláusula `let` introduz a variável `length range`

variável cujo nome nunca pode ser visto ou usado no código-fonte. A expressão de consulta da listagem 11.11 é traduzido para algo assim:

```
SampleData.AllUsers
    .Select(usuário => novo { usuário, comprimento = usuário.Nome.Comprimento })
    .OrderBy(z => z.comprimento)
    .Select(z => new { Nome = z.user.Name, Comprimento = z.length })
```

Cada parte da consulta foi ajustada adequadamente: onde a consulta original expressão referenciou usuário ou comprimento diretamente, se a referência ocorrer após let cláusula, ela será substituída por z.user ou z.length. A escolha de z como nome aqui é arbitrária – está tudo oculto pelo compilador.

**TIPOS ANÔNIMOS SÃO UM DETALHE DE IMPLEMENTAÇÃO** A rigor, cabe à implementação do compilador C# para decidir como agrupar diferentes variáveis de intervalo para fazer identificadores transparentes funcionarem. A Microsoft A implementação usa tipos anônimos, e a especificação também mostra as traduções nesses termos, então segui a tendência. Mesmo que outro compilador escolha uma abordagem diferente, isso não deverá afetar os resultados.

Se você consultar a especificação da linguagem sobre cláusulas let (seção 7.16.2.4), você veja que a tradução que ela descreve é de uma expressão de consulta para outra. Ele usa um asterisco (\*) para representar o identificador transparente introduzido. O identificador transparente é então apagado como etapa final da tradução. Não usarei essa notação neste capítulo, já que é difícil de entender e desnecessário no nível de detalhe que vamos em. Esperançosamente, com este pano de fundo, a especificação não será tão impenetrável como poderia ser de outra forma, caso você precise consultá-lo.

A boa notícia é que agora podemos ver o restante das traduções que compõem Suporte a expressões de consulta do C# 3. Não entrarei em detalhes de cada identificador transparente introduzido, mas mencionarei as situações em que eles ocorrem. Vejamos o suporte para junções primeiro.

## 11.5 Junções

Se você já leu alguma coisa sobre SQL, provavelmente tem uma ideia do que é uma junção de banco de dados. é. São necessárias duas tabelas (ou visualizações, ou funções com valor de tabela, e assim por diante) e cria um resultado combinando um conjunto de linhas com outro conjunto de linhas. Uma junção LINQ é semelhante, exceto que funciona em sequências. Três tipos de junção estão disponíveis, embora nem todos eles usam a palavra-chave join na expressão de consulta. Começaremos com a junção que é mais próximo de uma junção interna SQL.

### 11.5.1 Junções internas usando cláusulas de junção

As junções internas envolvem duas sequências. Uma expressão de seletor de chave é aplicada a cada elemento da primeira sequência, e outro seletor de chave (que pode ser totalmente diferente) é aplicado. aplicado a cada elemento da segunda sequência. O resultado da junção é uma sequência de todos os pares de elementos onde a chave do primeiro elemento é igual à chave do segundo elemento.

## CONFLITO DE TERMINOLOGIA! SEQUÊNCIAS INTERNAS E EXTERNAS A

documentação do MSDN para o método Join usado para avaliar junções internas chama as sequências envolvidas interna e externamente, e os parâmetros reais do método também são baseados nesses nomes. Isso não tem nada a ver com junções internas e junções externas – é apenas uma forma de diferenciar as sequências. Você pode pensar neles como o primeiro e o segundo, à esquerda e à direita, Bert e Ernie – qualquer coisa que você quiser e que o ajude. Usei esquerda e direita neste capítulo, para que fique claro qual é qual nos diagramas. Normalmente, *exterior* corresponde à *esquerda* e *interior* corresponde à *direita*.

As duas sequências podem ser o que você quiser; a sequência certa pode até ser igual à sequência esquerda, se isso for útil. (Imagine encontrar pares de pessoas que nasceram no mesmo dia, por exemplo.) A única coisa que importa é que as duas expressões do seletor de chave devem resultar no mesmo tipo de chave.<sup>5</sup> Você não pode unir uma

sequência de pessoas para uma sequência de cidades dizendo que a data de nascimento da pessoa é igual à população da cidade – não faz sentido. Mas uma possibilidade importante é usar um tipo anônimo para a chave; isso funciona porque os tipos anônimos implementam igualdade e hashing adequadamente. Se você precisar criar efetivamente uma chave com várias colunas, os tipos anônimos são a melhor opção. Isto também se aplica às operações de agrupamento que veremos mais tarde.

A sintaxe para uma junção interna parece mais complicada do que é:

[consulta selecionando a sequência esquerda] junta a  
variável do intervalo direito na sequência direita  
no seletor de tecla esquerda é igual ao seletor de tecla direita

Ver igual como uma palavra-chave contextual em vez de usar símbolos pode ser desconcertante, mas torna mais fácil distinguir o seletor de tecla esquerdo do seletor de tecla direito. Freqüentemente (mas nem sempre), pelo menos um dos seletores de chave é trivial, que apenas seleciona o elemento exato daquela sequência. A palavra-chave contextual é usada pelo compilador para separar os seletores de chave em diferentes expressões lambda. A capacidade do processador de consulta de obter as chaves para cada valor (em cada lado da junção) é importante tanto para o desempenho no LINQ to Objects quanto para a viabilidade de traduzir a consulta em outros formatos, como SQL.

Vejamos um exemplo do nosso sistema de defeitos. Suponha que você acabou de adicionar o recurso de notificação e deseja enviar o primeiro lote de e-mails para todos os defeitos existentes. Você precisa juntar a lista de notificações à lista de defeitos onde seus projetos correspondem. A listagem a seguir executa exatamente essa junção.

### Listagem 11.12 Unindo os defeitos e assinaturas de notificação com base no projeto

```
var query = do defeito em SampleData.AllDefects junta-se à assinatura em
    SampleData.AllSubscriptions em Defect.Project é igual a subscription.Project
```

<sup>5</sup> Também é válido que existam dois tipos de chaves envolvidos, com uma conversão implícita de um para o outro. Um dos tipos deve ser uma escolha melhor que o outro, da mesma forma que o compilador infere o tipo de um array digitado implicitamente. Na minha experiência, raramente é necessário considerar esse detalhe conscientemente.

```

seleciona novo {defect.Summary, subscription.EmailAddress};

foreach (entrada var na consulta) {

    Console.WriteLine("{0}: {1}", entrada.EmailAddress, entrada.Summary);
}

```

A Listagem 11.12 mostrará cada um dos defeitos do media player duas vezes – uma vez para mediabugs@skeetysoft.com e outra para theboss@skeetysoft.com (porque o chefe realmente se importa com o projeto do media player).

Neste caso específico, você poderia facilmente ter feito a junção ao contrário, invertendo as sequências esquerda e direita. O resultado teria sido as mesmas entradas, mas em uma ordem diferente. A implementação em LINQ to Objects retorna entradas tais que todos os pares que usam o primeiro elemento da sequência à esquerda são retornados (na ordem da sequência à direita), depois todos os pares que usam o segundo elemento da sequência à esquerda e assim por diante. A sequência direita é armazenada em buffer, mas a sequência esquerda é transmitida, portanto, se você quiser juntar uma sequência enorme a uma sequência minúscula, vale a pena usar a sequência minúscula como a sequência correta, se possível. A operação ainda está adiada: ela espera até você solicitar o primeiro par antes de ler qualquer dado de qualquer sequência. Nesse ponto, ele lê toda a sequência correta para construir uma pesquisa das chaves até os valores que produzem essas chaves. Depois disso, ele não precisa ler a sequência certa novamente e pode começar a iterar na sequência esquerda, produzindo pares apropriadamente.

Um erro que pode atrapalhar você é colocar os seletores de chave ao contrário. No seletor de tecla esquerda, apenas a variável de intervalo de sequência esquerda está no escopo; no seletor de chave direito, apenas a variável de intervalo certa está no escopo. Se você inverter as sequências esquerda e direita, terá que inverter também os seletores de teclas esquerda e direita. Felizmente, o compilador sabe que este é um erro comum e sugere o curso de ação apropriado.

Apenas para tornar mais óbvio o que está acontecendo, a figura 11.6 mostra as sequências como eles são processados.

Freqüentemente, você desejará filtrar a sequência, e filtrar antes que a junção ocorra é mais eficiente do que filtrá-la depois. Nesta fase, a expressão de consulta é mais simples se a sequência da esquerda for a que requer filtragem. Por exemplo, se você quiser mostrar apenas defeitos fechados, poderá usar esta expressão de consulta:

```

do defeito em SampleData.AllDefects onde defect.Status
== Status.Closed junta-se à assinatura em
SampleData.AllSubscriptions em defect.Project é igual a subscription.Project
select new {defect.Summary, subscription.EmailAddress }

```

Você pode realizar a mesma consulta com as sequências invertidas, mas é mais confuso:

```

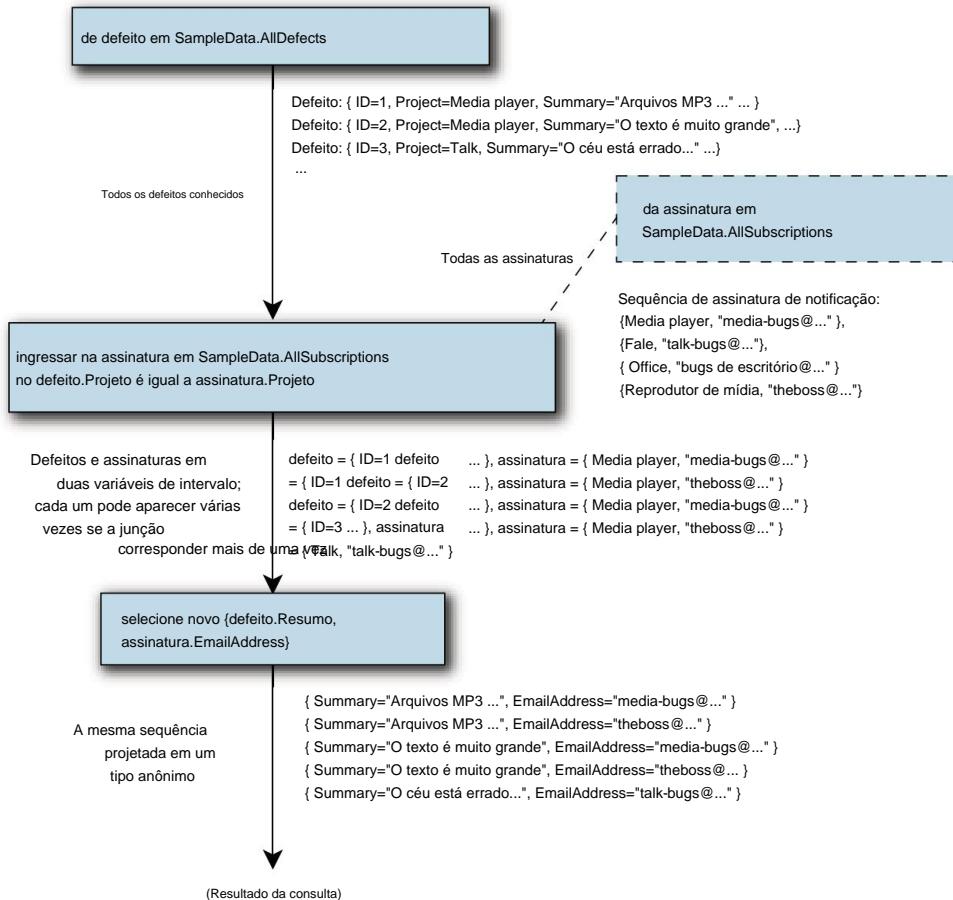
da assinatura em SampleData.AllSubscriptions junta-se ao defeito em (do
defeito em SampleData.AllDefects onde defect.Status == Status.Closed seleciona
o defeito)

```

```

na assinatura.Project é igual a defeito.Project selecione novo
{defeito.Summary, assinatura.EmailAddress}

```



**Figura 11.6** A junção da listagem 11.12 em forma gráfica, mostrando duas sequências diferentes (defeitos e assinaturas) usadas como fontes de dados

Observe como você pode usar uma expressão de consulta dentro de outra — a especificação da linguagem descreve muitas das traduções do compilador nesses termos. Expressões de consulta aninhadas são úteis, mas também prejudicam a legibilidade; muitas vezes vale a pena procurar uma alternativa, ou usando uma variável para a sequência à direita para deixar o código mais claro.

**AS JUNÇÕES INTERNAS SÃO ÚTEIS NO LINQ TO OBJECTS?** Junções internas são usadas o tempo todo em SQL. Eles são efetivamente a maneira como você navega de uma entidade para outra relacionada, geralmente unindo uma chave estrangeira em uma tabela à chave primária em outro. No modelo orientado a objetos, você tende a navegar de um objeto para outro através de referências. Por exemplo, recuperar o resumo de um defeito e o nome do usuário designado para trabalhar nele exigiria uma junção no SQL—em C# você geralmente usa apenas uma cadeia de propriedades. Se você tivesse uma associação reversa do Project com a lista de objetos NotificationSubscription associados a ele no modelo, não precisaria da junção para obter o resultado.

objetivo deste exemplo também. Isso não quer dizer que as junções internas às vezes não sejam úteis em modelos orientados a objetos, mas elas não ocorrem naturalmente como frequentemente como em modelos relacionais.

As junções internas são traduzidas pelo compilador em chamadas para o método `Join`, assim:

```
leftSequence.Join(rightSequence,
    leftKeySelector,
    rightKeySelector,
    seletor de resultado)
```

A assinatura da sobrecarga usada para LINQ to Objects é a seguinte (esta é a natureza da assinatura real, com os nomes reais dos parâmetros - daí as referências *internas* e *externas*):

```
estático IEnumerable<TResult> Join<TOuter,TInner,TKey,TResult> (
    este IEnumerable<TOuter> externo,
    IEnumerable<TInner> interno,
    Func<TOuter,TKey> outerKeySelector,
    Func<TInner,TKey> innerKeySelector,
    Func<TOuter,TInner,TResult> resultSelector
)
```

Os três primeiros parâmetros são autoexplicativos quando você se lembra de *tratar* e *exterior* como *direita* e *esquerda*, respectivamente, mas o último é mais interessante. É uma projeção de dois elementos (um da sequência esquerda e outro da direita sequência) em um único elemento da sequência resultante.

Quando a junção é seguida por algo diferente de uma cláusula `select`, o compilador C# 3 introduz um identificador transparente para tornar as variáveis de intervalo usadas em ambas as sequências disponíveis para cláusulas posteriores e cria um tipo anônimo e um mapeamento simples para usar para o parâmetro `resultSelector`.

Mas se a próxima parte da expressão de consulta for uma cláusula `select`, a projeção de uma cláusula `select` é usada diretamente como o parâmetro `resultSelector` - não faz sentido na criação de um par e depois chamar `Select` quando você puder fazer a transformação em uma etapa. Você ainda pode *pensar* nisso como uma etapa de "junção" seguida por uma etapa de "seleção", apesar do dois sendo esmagados em uma única chamada de método. Isso leva a uma mentalidade mais consistente no modelo na minha opinião, e um que é mais fácil de raciocinar. A menos que você esteja olhando para o código gerado, simplesmente ignore a otimização que o compilador está realizando para você.

A boa notícia é que, tendo aprendido sobre `inner joins`, você encontrará o próximo tipo de adesão muito mais fácil de abordar.

### 11.5.2 Junções de grupo com cláusulas `join...into`

Você viu que a sequência de resultados de uma cláusula de junção normal consiste em pares de elementos, um de cada sequência de entrada. Uma *associação de grupo* é semelhante em termos de expressão de consulta, mas tem um resultado significativamente diferente. Cada elemento de um grupo O resultado da junção consiste em um elemento da sequência esquerda (usando sua variável de intervalo original) e uma sequência de todos os elementos correspondentes da sequência direita, expostos como uma nova variável de intervalo especificada pelo identificador que vem depois da cláusula `join`.

Vamos mudar o exemplo anterior para usar uma associação de grupo. A listagem a seguir mostra novamente todos os defeitos e as notificações necessárias para cada um deles, mas os divide por defeito. Preste atenção especial em como os resultados são exibidos com um loop foreach aninhado.

#### Listagem 11.13 Unindo defeitos e assinaturas com uma junção de grupo

```
var query = from defeito in SampleData.AllDefects
            join assinatura in
                SampleData.AllSubscriptions
                on defeito.Projeto equals assinatura.Projeto
            into groupedSubscriptions
            select new { Defeito = defeito,
                        Assinaturas = groupedSubscriptions
                                      .SelectMany(s => s.Subscriptions)
                                      .ToList()
            };

foreach (var entrada in query) {
    Console.WriteLine(entrada.Defeito.Summary);
    foreach (var assinatura in entrada.Assinaturas) {
        Console.WriteLine("{0}", assinatura.EmailAddress);
    }
}
```

A propriedade Assinaturas de cada entrada é a sequência incorporada de assinaturas que correspondem ao defeito daquela entrada. A Figura 11.7 mostra como as duas sequências iniciais são combinadas.

Uma diferença importante entre uma junção interna e uma junção de grupo - e entre uma junção de grupo e um agrupamento normal - é que uma junção de grupo tem uma correspondência um-para-um entre a sequência esquerda e a sequência resultante, mesmo que alguns dos elementos da sequência esquerda não correspondem a nenhum elemento da sequência direita. Isso pode ser importante e às vezes é usado para simular uma *junção externa esquerda* do SQL. A sequência incorporada está vazia quando o elemento esquerdo não corresponde a nenhum elemento direito. Tal como acontece com uma junção interna, uma junção de grupo armazena em buffer a sequência direita, mas transmite a sequência esquerda.

A Listagem 11.14 mostra um exemplo disso, contando o número de defeitos criados em cada dia de maio. Ele usa um tipo `DateTimeRange` para gerar uma sequência de datas em maio como sequência à esquerda e uma projeção que chama `Count()` na sequência incorporada no resultado da junção do grupo.<sup>6</sup>

#### Listagem 11.14 Contando o número de defeitos levantados em cada dia de maio

```
var datas = new DateTimeRange(SampleData.Start, SampleData.End);

var query = from data in datas
            join defeito in
                SampleData.AllDefects
                on data equals defeito.Created.Date
            group new { Data = data, Contagem = ingressou.Count() }
            by data into grupos
            select grupos;
```

<sup>6</sup> Esta é uma implementação simples para fins de exemplo - não uma gama completa de uso geral.

```

foreach (var agrupado na consulta)
{
    Console.WriteLine("{0:d}: {1}", agrupado.Date, agrupado.Count);
}

```

O método Count() usa execução imediata, iterando por todos os elementos do a sequência em que é chamado - mas você está chamando-o apenas na parte de projeção do expressão de consulta, para que se torne parte de uma expressão lambda. Isso significa que você ainda tem execução diferida; nada é avaliado até você iniciar o loop foreach.

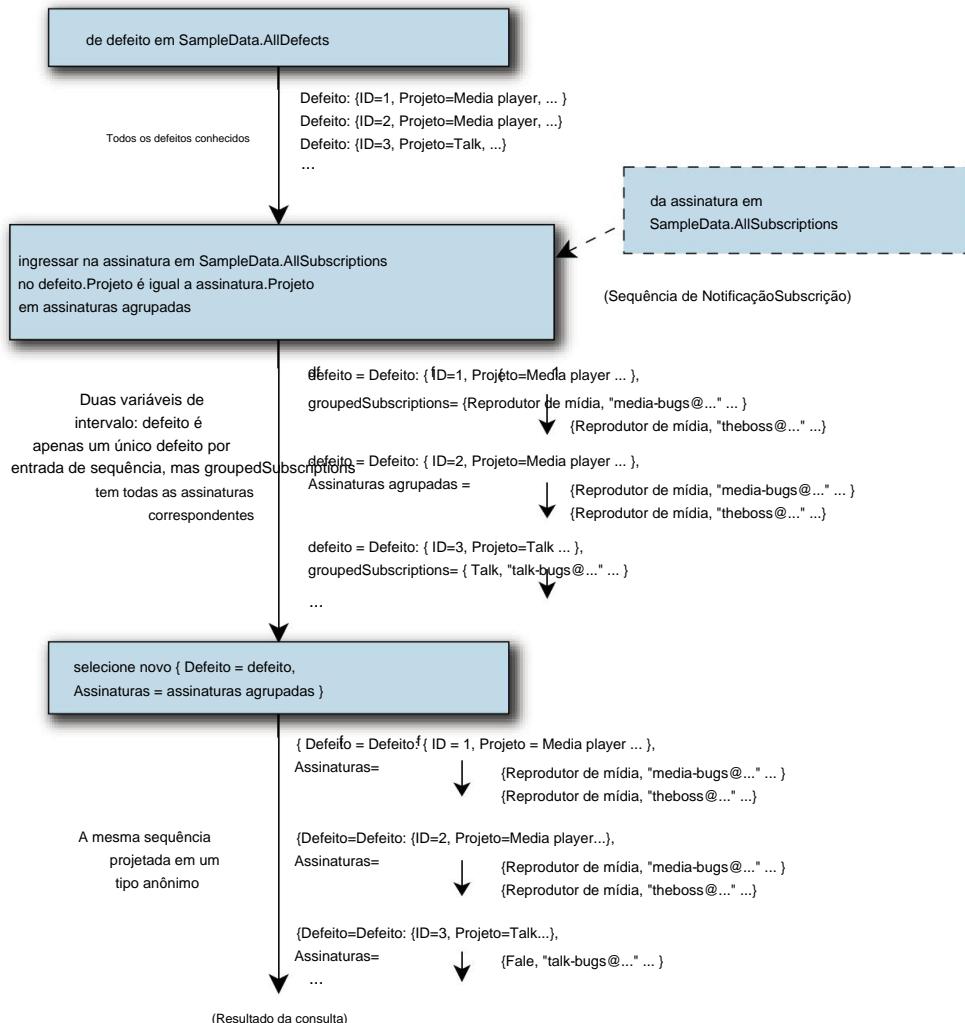


Figura 11.7 Sequências envolvidas na junção de grupo da listagem 11.13. As setas curtas indicam sequências incorporadas nas entradas de resultado. Na saída, algumas entradas contêm vários endereços de e-mail para o mesmo defeito.

Aqui está a primeira parte dos resultados da listagem 11.14, mostrando o número de defeitos criados a cada dia na primeira semana de maio:

```
01/05/2013: 1
02/05/2013: 0
03/05/2013: 2
04/05/2013: 1
05/05/2013: 0
06/05/2013: 1
07/05/2013: 1
```

A tradução do compilador envolvida para uma junção de grupo é simplesmente uma chamada ao método GroupJoin da mesma forma que uma junção interna se traduz em uma chamada para Join. Aqui está a assinatura para Enumerable.GroupJoin:

```
estático IEnumerable<TResult> GroupJoin<TOuter,TInner,TKey,TResult>( este IEnumerable<TOuter>
    externo,
    IEnumerable<TInner> interno,
    Func<TOuter,TKey> outerKeySelector,
    Func<TInner,TKey> innerKeySelector,
    Func<TOuter,IEnumerable<TInner>,TResult> resultSelector
)
```

Isso é exatamente o mesmo que para junções internas, exceto que o parâmetro resultSelector precisa funcionar com uma sequência de elementos à direita, e não apenas com um único. Tal como acontece com as junções internas, se uma junção de grupo for seguida por uma cláusula select , a projeção será usada como o seletor de resultados da chamada GroupJoin ; caso contrário, é introduzido um identificador transparente. Neste caso você tem uma cláusula select imediatamente após a adesão ao grupo, então a consulta traduzida fica assim:

```
datas.GroupJoin(SampleData.AllDefects, data => data, defeito
    =>
    defeito.Created.Date, (data, ingresso) => novo
    { Data = data, Contagem = ingresso.Count() })
```

O tipo final de junção é conhecido como *junção cruzada*, mas não é tão simples quanto pode parecer inicialmente.

### 11.5.3 Junções cruzadas e sequências de nívelamento usando múltiplas cláusulas from

Até agora, todas as nossas junções foram *equijuntas* – uma correspondência foi realizada entre os elementos das sequências esquerda e direita. As junções cruzadas não realizam nenhuma correspondência entre as sequências; o resultado contém todos os pares possíveis de elementos. Isto é conseguido simplesmente usando duas (ou mais) cláusulas from . Por uma questão de sanidade, consideraremos apenas duas cláusulas from no momento - quando houver mais, apenas execute mentalmente uma junção cruzada nas duas primeiras cláusulas from e, em seguida, faça uma junção cruzada da sequência resultante com a próxima cláusula from , e assim por diante. Cada cláusula extra from adiciona sua própria variável de intervalo por meio de um identificador transparente.

A listagem a seguir mostra uma junção cruzada simples (mas inútil) em ação, produzindo uma sequência onde cada entrada consiste em um usuário e um projeto. Escolhi deliberadamente duas sequências iniciais completamente não relacionadas para mostrar que nenhuma correspondência é realizada.

### Listagem 11.15 Junção cruzada de usuários em projetos

```
var query = do usuário em SampleData.AllUsers do projeto em
    SampleData.AllProjects selecione novo { Usuário = usuário,
    Projeto = projeto };
foreach (par var na consulta) {

    Console.WriteLine("{0}/{1}",
        par.Usuário.Nome,
        par.Projeto.Nome);
}
```

A saída da listagem 11.15 começa assim:

```
Tim Trotter/Skeety Media Player
Tim Trotter/Skeety Talk
Tim Trotter/Skeety Office
Tara Tutu/Skeety Media Player
Tara Tutu/Skeety Talk
Tara Tutu/Escrítorio Skeety
```

A Figura 11.8 mostra as sequências envolvidas para obter esse resultado.

Se você estiver familiarizado com SQL, provavelmente já se sente confortável até agora – parece um produto cartesiano obtido de uma consulta especificando múltiplas tabelas. Porém, mais potência está disponível quando você deseja: a sequência certa pode depender do valor atual da sequência esquerda. Em outras palavras, cada elemento da sequência esquerda é usado para gerar uma sequência direita e, em seguida, esse elemento esquerdo é emparelhado com cada elemento da nova sequência. Quando for esse o caso, não é uma junção cruzada no sentido normal do termo. Em vez disso, está efetivamente achatando uma sequência de sequências em uma única sequência. A tradução da expressão de consulta é a mesma, quer você esteja usando ou não uma junção cruzada verdadeira, portanto, você precisa entender o cenário mais complicado para entender o processo de tradução.

Antes de mergulharmos nos detalhes, vamos ver o efeito que isso produz. A seguinte listagem mostra um exemplo simples, usando sequências de inteiros.

### Listagem 11.16 Cross join onde a sequência direita depende do elemento esquerdo

```
var query = da esquerda em Enumerable.Range(1, 4)
    da direita em Enumerable.Range(11, left) selecione new { Left =
    left, Right = right };

foreach (par var na consulta) {

    Console.WriteLine("Esquerda={0}; Direita={1}", par.Esquerda,
        par.Direita);
}
```

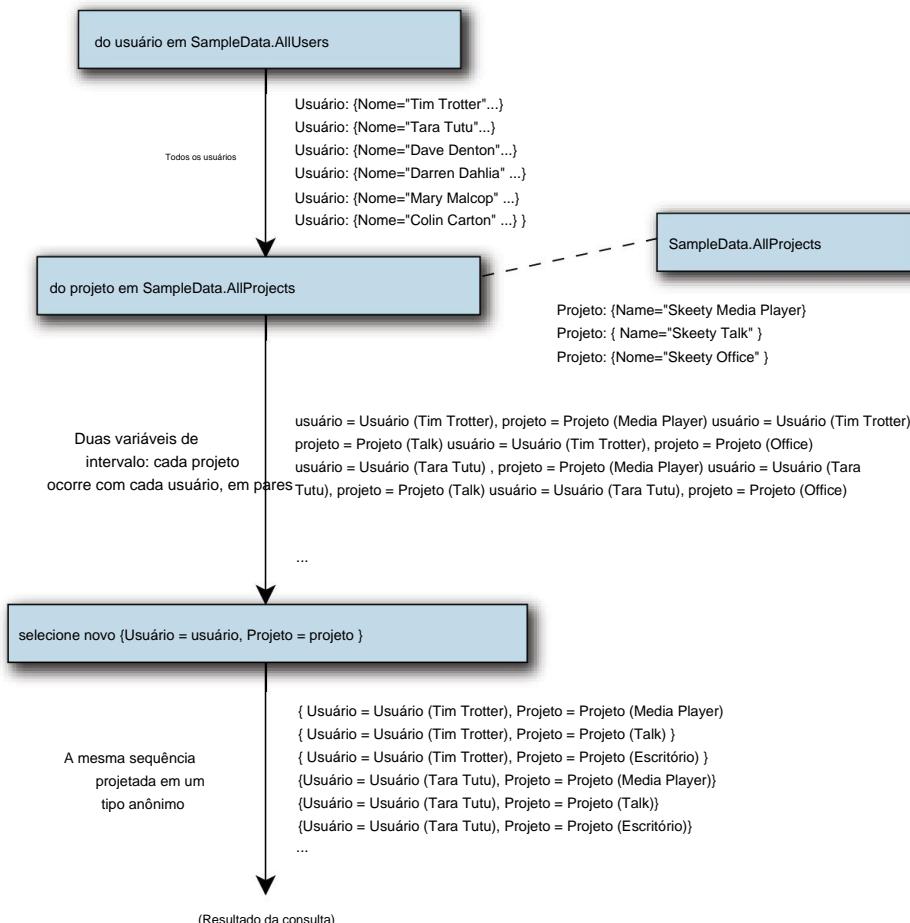


Figura 11.8 Sequências da Listagem 11.15, união cruzada de usuários e projetos. Todas as combinações possíveis são retornadas nos resultados.

A Listagem 11.16 começa com um intervalo simples de números inteiros, de 1 a 4. Para cada um desses números inteiros, você cria outro intervalo, começando em 11 e tendo tantos elementos quanto o número inteiro original. Ao usar múltiplas cláusulas from, a sequência esquerda é unida a cada uma das sequências direitas geradas, resultando nesta saída:

```

Esquerda=1; Direita=11
Esquerda=2; Direita=11
Esquerda=2; Direita=12
Esquerda=3; Direita=11
Esquerda=3; Direita=12
Esquerda=3; Direita=13
Esquerda=4; Direita=11
Esquerda=4; Direita=12
Esquerda=4; Direita=13
Esquerda=4; Direita=14
  
```

O método que o compilador chama para gerar essa sequência é `SelectMany`. É necessária uma única sequência de entrada (a sequência `esquerda` em nossa terminologia), um delegado para *gerar* outra sequência a partir de qualquer elemento da sequência esquerda e um delegado para gerar um elemento de resultado dado um elemento de cada uma das sequências. Aqui está a assinatura de `Enumerable.SelectMany`:

```
static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    esta fonte IEnumerable<TSource>,
    Func<TSource, IEnumerable<TCollection>> coleçãoSelector,
    Func<TSource, TCollection, TResult> resultSelector
)
```

Tal como acontece com as outras junções, se a parte da expressão de consulta após a junção for uma cláusula `select`, essa projeção será usada como argumento final; caso contrário, um identificador transparente será introduzido para disponibilizar as variáveis de intervalo das sequências esquerda e direita posteriormente na consulta.

Só para tornar tudo isso um pouco mais concreto, aqui está a expressão de consulta de listagem 11.16 como código-fonte traduzido:

```
Enumerable.Range(1, 4)
    .SelectMany(esquerda => Enumerable.Range(11, esquerda), (esquerda,
        direita) => novo {Esquerda = esquerda, Direita = direita})
```

Um recurso interessante do `SelectMany` é que a execução é completamente transmitida — ele só precisa processar um elemento de cada sequência por vez, porque usa uma sequência direita recém-gerada para cada elemento diferente da sequência esquerda.

Compare isso com junções internas e junções de grupo: ambas carregam a sequência correta completamente antes de começar a retornar qualquer resultado.

O comportamento de nivelamento de `SelectMany` pode ser muito útil. Considere uma situação em que você deseja processar muitos arquivos de log, uma linha por vez. Você pode processar uma sequência contínua de linhas quase sem nenhum trabalho. O pseudocódigo a seguir é preenchido mais detalhadamente no código-fonte para download, mas o significado geral e a utilidade devem ser claros:

```
var query = do arquivo em Directory.GetFiles(logDirectory, "*.log") da linha em ReadLines(file) let entry = new
    LogEntry(line) where entry.Type ==
        EntryType.Error selecione a entrada;
```

Em apenas cinco linhas de código, você pode recuperar, analisar e filtrar uma coleção inteira de arquivos de log, retornando uma sequência de entradas que representam erros. O mais importante é que você não precisa carregar nem mesmo um único arquivo de log completo na memória de uma só vez, muito menos todos os arquivos — todos os dados são transmitidos.

Tendo abordado as junções, os últimos itens que precisamos examinar são um pouco mais fáceis de entender. Veremos como agrupar elementos por uma chave e continuar uma expressão de consulta após uma cláusula `group ... by ou select`.

## 11.6 Agrupamentos e continuações

Um requisito comum é agrupar uma sequência de elementos por uma de suas propriedades. O LINQ facilita isso com a cláusula `group ... by`. Além de descrever esse tipo final de cláusula nesta seção, também revisitaremos o `select` para ver um recurso chamado `continuações` de consulta que pode ser aplicado tanto a agrupamentos quanto a projeções. Vamos começar com um agrupamento simples.

### 11.6.1 Agrupamento com a cláusula `group...by`

O agrupamento é bastante intuitivo e o LINQ o torna simples. Para agrupar uma sequência em uma expressão de consulta, tudo que você precisa fazer é usar a cláusula `group ... by`, com esta sintaxe:

projeção de grupo por agrupamento

Esta cláusula vem no final de uma expressão de consulta da mesma forma que uma cláusula `select`. As semelhanças entre essas cláusulas não param por aí: a expressão de projeção é o mesmo tipo de projeção que as cláusulas selecionadas usam. O resultado é um pouco diferente, no entanto.

A expressão de agrupamento determina pelo que a sequência é agrupada – é o seletor de chave da operação de agrupamento. O resultado geral é uma sequência onde cada elemento é um grupo. Cada grupo é uma sequência de elementos projetados que também possui uma propriedade `Key`, que é a chave desse grupo; essa combinação é encapsulada na interface `IGrouping< TKey, TElement >`, que estende `IEnumerable< TElement >`. Novamente, se quiser agrupar por vários valores, você pode usar um tipo anônimo para a chave.

Vejamos um exemplo simples do sistema de defeitos SkeetySoft: agrupar defeitos por seu responsável atual. A listagem a seguir faz isso com a forma mais simples de projeção, de modo que a sequência resultante tenha o destinatário como chave e uma sequência de defeitos embutida em cada entrada.

#### Listagem 11.17 Agrupando defeitos por cessionário – projeção trivial

```
var query = do defeito em SampleData.AllDefects
    onde defeito.AssignedTo! = Nulo
    agrupar defeito por defeito.AssignedTo;

foreach (entrada var na consulta)
{
    Console.WriteLine(entrada.Chave.Nome);
    foreach (var defeito na entrada)
    {
        Console.WriteLine("({0})(1)",
            defeito.Severidade, defeito.Resumo);
    }
    Console.WriteLine();
}
```

A Listagem 11.17 pode ser útil em um relatório diário de construção, para ver rapidamente quais defeitos cada pessoa precisa observar. Ele filtra todos os defeitos que não precisam de mais atenção **B** e depois agrupa usando a propriedade `AssignedTo`. Embora desta vez você esteja usando apenas uma propriedade, a expressão de agrupamento pode ser qualquer coisa que você quiser — ela é aplicada a cada entrada na sequência recebida e a sequência é agrupada com base no resultado da expressão. Observe que o agrupamento não pode transmitir os resultados; aplica a seleção de chave e projeção a cada elemento na entrada e armazena em buffer as sequências agrupadas de elementos projetados. Mas mesmo que não seja transmitido, a execução ainda será adiada até você começar a recuperar os resultados.

A projeção aplicada no agrupamento **C** é trivial — apenas seleciona o elemento original. À medida que você percorre a sequência resultante, cada entrada possui uma propriedade `Key`, que é do tipo `Usuário D`, e cada entrada também implementa `IEnumerable<Defect>`, que é a sequência de defeitos atribuídos a esse usuário `E`.

Os resultados da listagem 11.17 começam assim:

Darren Dahlia

- (Showstopper) Sistema de travamento de arquivos MP3
- (Maior) Não é possível reproduzir arquivos com mais de 200 bytes
- (Principal) DivX está instável no Pentium 100
- (Trivial) A interface do usuário deveria ser mais caramelada

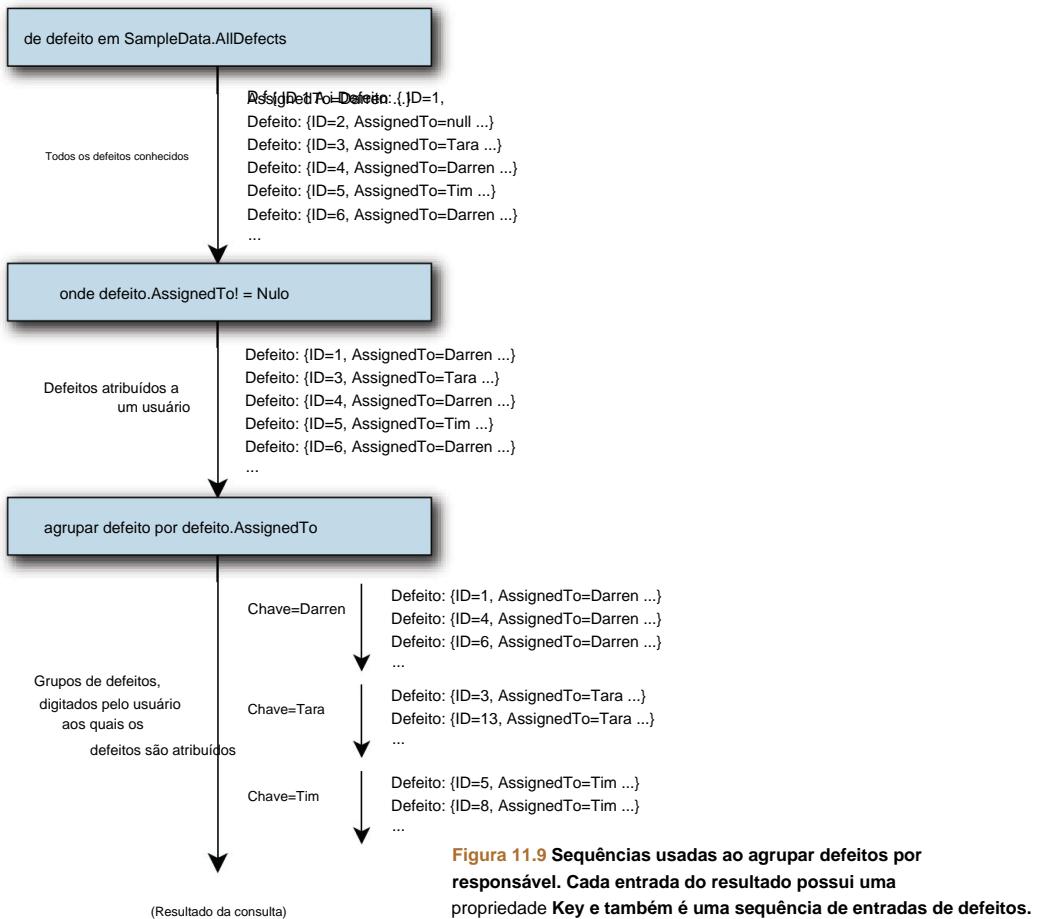
Depois que todos os defeitos de Darren forem devolvidos, você verá os de Tara, depois os de Tim e assim por diante. A implementação mantém efetivamente uma lista dos destinatários vistos até o momento e adiciona uma nova sempre que necessário. A Figura 11.9 mostra as sequências geradas ao longo da expressão de consulta, o que pode tornar essa ordenação mais clara.

Dentro da subsequência de cada entrada, a ordem dos defeitos é a mesma da sequência de defeitos original. Se você se preocupa ativamente com a ordem, considere declará-la explicitamente na expressão de consulta, para torná-la mais legível.

Se você executar a listagem 11.17, verá que Mary Malcop não aparece na saída, porque ela não tem nenhum defeito atribuído a ela. Se você quisesse produzir uma lista completa de usuários e defeitos atribuídos a cada um deles, você precisaria usar uma junção de grupo como a da listagem 11.14.

O compilador sempre usa um método chamado `GroupBy` para agrupar cláusulas. Quando a projeção em uma cláusula de agrupamento é trivial — quando cada entrada na sequência original é mapeada diretamente para exatamente o mesmo objeto em uma subsequência — o compilador usa uma chamada de método simples que requer apenas a expressão de agrupamento, para que ele saiba como mapear cada elemento para uma chave. Por exemplo, a expressão de consulta na listagem 11.17 é traduzida assim:

```
SampleData.AllDefects.Where (defeito => defeito.AssignedTo! = Nulo)
    .GroupBy(defeito => defeito.AssignedTo)
```



Quando a projeção não é trivial, utiliza-se uma versão um pouco mais complicada. A listagem a seguir fornece um exemplo de projeção em que você captura apenas o resumo de cada defeito, e não o próprio objeto Defeito .

#### Listagem 11.18 Agrupando defeitos por cessionário – a projeção retém apenas o resumo

```

var consulta = do defeito em SampleData.AllDefects onde defeito.AssignedTo =
    grupo nulo defeito.Summary por defeito.AssignedTo;

foreach (entrada var na consulta) {

    Console.WriteLine(entrada.Chave.Nome); foreach (var
    resumo na entrada) {

        Console.WriteLine("{0}", resumo);
    }
    Console.WriteLine();
}
  
```

Destaquei as diferenças entre as listagens 11.18 e 11.17 em negrito. Como cada defeito é projetado apenas em seu resumo, a sequência incorporada em cada entrada é apenas um `IEnumerable<string>`. Neste caso o compilador utiliza uma sobrecarga de `GroupBy` com outro parâmetro para representar a projeção. A expressão de consulta na listagem 11.18 é traduzida na seguinte expressão:

```
SampleData.AllDefects.Where (defeito => defeito.AssignedTo! = Nulo)
    .GroupBy(defeito => defeito.AssignedTo, defeito =>
        defeito.Resumo)
```

As cláusulas de agrupamento são relativamente simples, mas úteis. Mesmo no sistema de rastreamento de defeitos, você poderia facilmente imaginar querer agrupar os defeitos por projeto, criador, gravidade ou status, bem como pelo responsável usado nesses exemplos.

Até agora, você finalizou cada expressão de consulta com uma cláusula `select` ou `group ... by`, e esse foi o fim da expressão. Mas há momentos em que você desejará fazer mais com os resultados, e é aí que as continuações de consulta são usadas.

#### 11.6.2 Continuações de consulta

As continuações de consulta fornecem uma maneira de usar o resultado de uma expressão de consulta como a sequência inicial de outra. Elas se aplicam às cláusulas `group ... by` e `select`, e a sintaxe é a mesma para ambas - você simplesmente usa a palavra-chave contextual `into` e fornece o nome de uma nova variável de intervalo. Essa variável de intervalo pode então ser usada na próxima parte da expressão de consulta.

A especificação explica isso em termos de tradução de uma expressão de consulta para outro, mudando

primeira consulta no identificador segundo  
corpo de consulta

em

do identificador em (primeira consulta) segundo  
corpo de consulta

Um exemplo tornará isso mais claro. Voltemos ao agrupamento de defeitos por cessionário, mas desta vez imagine que você deseja apenas a contagem dos defeitos atribuídos a cada pessoa.

Você não pode fazer isso com a projeção na cláusula de agrupamento, porque isso se aplica apenas a cada defeito individual. Você deseja projetar cada grupo, que contém um responsável e a sequência de seus defeitos, em um único elemento que consiste no responsável e na contagem de defeitos no grupo. Isso pode ser alcançado com o código a seguir.

#### Listagem 11.19 Continuando um agrupamento com outra projeção

```
var query = from defeito in SampleData.AllDefects where defect.AssignedTo!
    = null agrupar defeito por defeito.AssignedTo em
    agrupado selecione novo { Assignee = grouped.Key,
    Contagem = agrupada.Contagem() };

foreach (entrada var na consulta)
```

```
{
    Console.WriteLine("{0}: {1}",
        entrada.Assinee.Nome, entrada.Count);
}
```

As alterações na expressão de consulta são destacadas em negrito. Você pode usar a variável de intervalo agrupado na segunda parte da consulta, mas a variável de intervalo de defeito não está mais disponível – você pode considerá-la fora do escopo. Essa projeção simplesmente cria um tipo anônimo com propriedades `Assinee` e `Count`, usando a chave de cada grupo como destinatário e contando a sequência de defeitos associados a cada grupo.

Os resultados da listagem 11.19 são os seguintes:

```
Darren Dahlia: 14
Tutu de Tara: 5
Tim Trotter: 5
Débora Denton: 9
Colin Caixa: 2
```

Seguindo a especificação, a expressão de consulta da listagem 11.19 é traduzida nesta:

```
de agrupado em (de defeito em SampleData.AllDefects
    onde defeito.AssignedTo!= grupo nulo defeito por
    defeito.AssignedTo)
seleciona novo {Destinatário = grouped.Key, Count = grouped.Count() }
```

O restante das traduções são então realizadas, resultando no seguinte código:

```
SampleData.AllDefects .Where(defeito
    => defeito.AssignedTo != nulo)
    .GroupBy(defeito => defeito.AssignedTo)
    .Select(agrupado => novo { Cessionário = agrupado.Chave,
        Contagem = agrupado.Contagem() })
```

Uma forma alternativa de entender as continuações é pensar em duas afirmações separadas. Isso não é tão preciso em termos da tradução real do compilador, mas acho que facilita ver o que está acontecendo. Neste caso, a expressão de consulta (e a atribuição à variável de consulta) pode ser considerada como as duas declarações a seguir:

```
var tmp = do defeito em SampleData.AllDefects
    onde defeito.AssignedTo!= grupo nulo defeito por
    defeito.AssignedTo;

var query = from agrupado em tmp
    selecione novo {Destinatário = agrupado.Chave,
        Contagem = agrupada.Contagem() };
```

Claro, se você achar isso mais fácil de ler, não há nada que o impeça de dividir a expressão original nesta forma em seu código-fonte. Nada será avaliado até que você comece a tentar percorrer os resultados da consulta de qualquer maneira, devido à execução adiada.

**JOIN...INTO NÃO É UMA CONTINUAÇÃO** É fácil cair na armadilha de pensar que onde quer que você veja a palavra-chave contextual, você terá uma continuação. Isso não é verdade para junções - a cláusula `join ... into` (que é usada para junções de grupo)

não forma uma continuação. A diferença importante é que com uma junção de grupo, todas as variáveis de intervalo anteriores (exceto aquela usada para nomear o lado direito da junção) ainda podem ser usadas. Compare isso com as consultas que estamos analisando nesta seção, onde a continuação limpa a lousa; a única variável de intervalo disponível posteriormente é aquela declarada pela continuação.

Vamos estender este exemplo para ver como múltiplas continuações podem ser usadas. Os resultados estão atualmente desordenados – vamos mudar isso para que você possa ver quem tem mais defeitos atribuídos a eles primeiro. Você poderia usar uma cláusula `let` após a primeira continuação, mas a listagem a seguir mostra uma alternativa com uma segunda continuação após a expressão atual.

#### Listagem 11.20 Consultar continuações de expressão do grupo e selecionar

```
var query = from defeito in SampleData.AllDefects onde defect.AssignedTo!
    = null agrupar defeito por defeito.AssignedTo em
    agrupado selecione novo { Assignee = grouped.Key,
        Count = grouped.Count() } no resultado ordenado por
        result.Count descendente selecione o resultado;

foreach (entrada var na consulta) {

    Console.WriteLine("{0}: {1}",
        entrada.Assignee.Nome,
        entrada.Count);
}
```

As alterações entre as listagens 11.19 e 11.20 estão destacadas em negrito. Você não precisava alterar nenhum código de saída, porque tinha o mesmo tipo de sequência - bastava aplicar uma ordem a ela.

Desta vez, a expressão de consulta traduzida é a seguinte:

```
SampleData.AllDefects .Where(defeito
    => defeito.AssignedTo != nulo)
    .GroupBy(defeito => defeito.AssignedTo)
    .Select(agrupado => novo { Cessionário = agrupado.Chave,
        Contagem = agrupado.Contagem() })
    .OrderByDescendente(resultado => resultado.Contagem);
```

Por pura coincidência, isso é notavelmente semelhante à primeira consulta de rastreamento de defeitos que examinamos, na seção 10.3.6. A cláusula `select` final efetivamente não faz nada, então o compilador C# a ignora. Porém, ela é obrigatória na expressão de consulta, pois todas as expressões de consulta devem terminar com uma cláusula `select` ou `group...by`. Não há nada que impeça de usar uma projeção diferente ou realizar outras operações com a consulta continuada – junções, agrupamentos adicionais e assim por diante. Fique de olho na legibilidade da expressão de consulta à medida que ela cresce.

Falando em legibilidade, há opções a serem consideradas ao escrever consultas LINQ.

## 11.7 Escolhendo entre expressões de consulta e notação de ponto

Como você viu ao longo deste capítulo, as expressões de consulta são traduzidas em expressões normais C# antes de ser compilado posteriormente. Não existe um nome oficial para uma chamada para o LINQ operadores de consulta escritos usando C# normal em vez de uma expressão de consulta, mas muitos os desenvolvedores agora se referem a isso como *notação de ponto*.<sup>7</sup> Toda expressão de consulta pode ser escrita em notação de ponto, mas o inverso não é verdadeiro: muitos operadores LINQ não têm uma expressão de consulta equivalente em C#. A grande questão é esta: quando você deve usar qual sintaxe?

### 11.7.1 Operações que requerem notação de ponto

A situação mais óbvia em que você é迫使ado a usar a notação de ponto é quando você está chamando um método como Reverse ou ToDictionary que não está representado na consulta sintaxe de expressão. Mas mesmo quando você usa um operador de consulta compatível com expressões de consulta, é bem possível que a sobrecarga que você deseja esteja indisponível.

Por exemplo, Enumerable.Where tem uma sobrecarga onde o índice no par sequência é fornecida como outro argumento para o delegado. Em tal situação, você poderia usar um código como o seguinte para obter todos os outros itens de uma sequência:

```
sequência.Where((item, índice) => índice% 2 == 0)
```

Há uma sobrecarga semelhante para Select, então se você quiser chegar ao original índice em uma sequência após o pedido, você poderia fazer algo assim:

```
sequencia.Select((Item, Índice) => new { Item, Índice })
    .OrderBy(x => x.Item.Nome)
```

Este exemplo mostra outra opção que você pode considerar: se for usar um parâmetro de expressão lambda diretamente em um tipo anônimo, você poderia contrariar a convenção normal de iniciar o nome do parâmetro com uma letra minúscula e, em seguida, usar um inicializador de projeção para evitar escrever novo { Item = item, Índice = índice }, que pode ser distraindo. Claro, você pode ignorar a convenção sobre nomes de propriedades, e faça com que seu tipo anônimo tenha propriedades começando com uma letra minúscula (item e índice, por exemplo). Tudo isso depende inteiramente de você e vale a pena experimentar. Embora a consistência seja geralmente importante, não importa muito aqui, uma vez que o impacto da inconsistência está confinado ao método em questão; você não está expondo esses nomes em sua API pública ou no restante de sua classe.

Muitos dos operadores de consulta também suportam comparações personalizadas – ordenação e juntando-se são os exemplos mais óbvios. É pouco provável que estes sejam necessários com frequência, em minha experiência, mas ocasionalmente são inestimáveis. Por exemplo, se você deseja realizar uma junção no nome de uma pessoa sem distinção entre maiúsculas e minúsculas, você pode especificar StringComparison.OrdinalIgnoreCase (ou um comparador específico de cultura) como o argumento final para uma chamada de ingresso . Novamente, se você sentir que um operador *quase* faz o que você quer, mas não bastante, verifique a documentação para outras sobrecargas.

---

<sup>7</sup> Esse é o termo que usarei de agora em diante, mas se você ouvir outras pessoas falando sobre *notação fluente*, provavelmente eles querem dizer a mesma coisa.

Quando você é forçado a usar a notação de ponto, a decisão de usá-la é fácil, mas o que sobre casos em que uma expressão de consulta poderia ser usada?

#### 11.7.2 Expressões de consulta onde a notação de ponto pode ser mais simples

Alguns desenvolvedores usam expressões de consulta em todos os lugares onde podem; Pessoalmente, vejo o que a consulta está fazendo e decido qual abordagem será mais legível.

Por exemplo, pegue esta expressão de consulta, que é semelhante a outra no início deste capítulo:

```
var adultos = de pessoa em pessoas
    onde person.Age >= 18 selecione
        pessoa;
```

São três linhas de código com muita bagagem, embora tudo o que faça seja filtrar.

Neste caso eu usaria a notação de ponto:

```
var adultos = pessoas.Where(pessoa => pessoa.Idade >= 18);
```

Acho isso mais claro: cada parte menciona algo em que você realmente está interessado.

Outra área em que o uso da notação de ponto em uma expressão de consulta pode fornecer mais clareza é quando você é forçado a usá-la em parte da consulta de qualquer maneira. Por exemplo, suponha que você vá usar o método de extensão `ToList()` para obter uma lista de nomes de adultos. (Estou realizando uma projeção também, neste caso, para que seja uma comparação mais equilibrada.) Aqui está a expressão de consulta:

```
var adultNames = (de pessoa em pessoas onde pessoa.Idade
    >= 18 selecione
        pessoa.Nome).ToList();
```

Aqui está o equivalente em notação de ponto:

```
var adultNames = pessoas.Where(pessoa => pessoa.Idade >= 18)
    .Select(pessoa => pessoa.Nome)
    .Listar();
```

Algo sobre a necessidade de parênteses em torno da expressão de consulta no primeiro caso faz com que pareça mais feio para mim. Este é um caso de escolha pessoal – esta seção está apenas aumentando sua consciência de que existe uma escolha e que você pode escolher. Se você for usar o LINQ de forma significativa, você realmente deve se sentir confortável com ambas as notações, e não há mal nenhum em mudar de estilo com base na consulta em questão. Como você viu, o código gerado é absolutamente equivalente. Nada disso quer dizer que eu não goste de expressões de consulta, é claro.

#### 11.7.3 Onde as expressões de consulta brilham

Tendo explicado onde você pode achar a notação de ponto benéfica, devo salientar que quando se trata de qualquer operação em que a expressão de consulta usaria identificadores transparentes — particularmente junções — a notação de ponto começa a sofrer em termos de legibilidade.

A beleza dos identificadores transparentes é que eles são transparentes — tão transparentes que você não conseguevê-los quando precisa apenas olhar a expressão da consulta. Mesmo um

a simples cláusula let pode ser suficiente para mudar a decisão em favor das expressões de consulta; introduzir um novo tipo anônimo apenas para propagar o contexto por meio da consulta torna-se irritante rapidamente.

A outra área em que as expressões de consulta vencem é em situações em que seriam necessárias múltiplas expressões lambda, ou mesmo múltiplas chamadas de método. Novamente, isso inclui junções, onde você deve especificar o seletor de chave para cada lado da junção, bem como o seletor de resultados. Por exemplo, aqui está uma versão reduzida de uma consulta anterior onde introduzi junções internas:

```
do defeito em SampleData.AllDefects junte-se à
assinatura em SampleData.AllSubscriptions em defeito.Projeto é igual a
    assinatura.Projeto selecione novo {defeito.Summary,
    assinatura.EmailAddress}
```

Em um IDE, seria razoável colocar toda a cláusula join em uma linha, resultando em um código bastante fácil de ler. O equivalente em notação de ponto é bastante horrível:

```
SampleData.AllDefects.Join(SampleData.AllSubscriptions, defeito => defeito.Projeto,
    assinatura => assinatura.Projeto,
    (defeito, assinatura) => novo {defeito.Summary,
        assinatura.EmailAddress })
```

O último argumento poderia caber em uma linha em um IDE, mas ainda é muito feio porque as expressões lambda não têm muito contexto; você não pode dizer imediatamente qual argumento significa o quê. Argumentos nomeados em C# 4 podem ajudar nisso, mas adicionam ainda mais volume à consulta.

Ordenações complexas podem ser igualmente desagradáveis na notação de pontos. Considere qual você prefere ler - esta cláusula orderby

ordenar por item. Classificação decrescente, item. Preço, item. Nome  
ou três chamadas de método:

```
.OrderByDescendente(item => item.Classificação)
.ThenBy(item => item.Preço)
.ThenBy(item => item.Nome)
```

Alterar a prioridade dessas ordenações é simples na expressão de consulta – basta trocá-las. Na notação de ponto, também pode ser necessário mudar de OrderBy para ThenBy ou vice-versa.

Para reiterar, não estou tentando impor minhas preferências pessoais em seu código. Eu simplesmente quero que você saiba o que está disponível e pense nas escolhas que você faz. É claro que esse é apenas um aspecto da escrita de código legível, mas é uma área totalmente nova a ser considerada em C#.

## 11.8 Resumo

Neste capítulo, vimos como o LINQ to Objects e o C# 3 interagem, focando na maneira como as expressões de consulta são primeiramente traduzidas em código que não envolve consulta.

expressões e então são compiladas da maneira usual. Você viu como todas as expressões de consulta formam uma série de sequências, aplicando uma transformação de alguma descrição em cada passo. Em muitos casos, essas sequências são avaliadas usando execução adiada, buscar dados somente quando for necessário pela primeira vez.

Comparadas com todos os outros recursos do C# 3, as expressões de consulta parecem um tanto alien – mais parecido com SQL do que com o C# com o qual você está acostumado. Uma das razões pelas quais eles parecem tão estranhos é que eles são declarativos em vez de imperativos - uma consulta fala sobre os recursos do resultado final, e não as etapas exatas necessárias para alcançá-lo. Isso anda de mãos dadas com uma forma de pensar mais funcional. Pode demorar um pouco para clicar e não é adequado para cada situação, mas onde a sintaxe declarativa for apropriada, ela pode melhorar enormemente legibilidade, além de tornar o código mais fácil de testar e paralelizar.

Não se iluda pensando que o LINQ só deve ser usado com bancos de dados. Simples a manipulação de coleções na memória é comum, e você viu como ela é bem suportada por expressões de consulta e métodos de extensão em `Enumerable`.

Na verdade, você já viu todos os recursos introduzidos no C# 3! Nós não temos já olhou para qualquer outro provedor LINQ, mas você tem uma compreensão mais clara do que o compilador fará isso quando você solicitar que ele lide com XML e SQL. O compilador em si não sabe a diferença entre LINQ to Objects, LINQ to SQL ou qualquer um dos outros fornecedores; apenas segue cegamente as mesmas regras.

No próximo capítulo você verá como essas regras formam a peça final do quebra-cabeça do LINQ quando convertem expressões lambda em árvores de expressão para que as diversas cláusulas das expressões de consulta possam ser executadas em diferentes plataformas. Você também vai veja alguns outros exemplos do que o LINQ pode fazer.

# 12

## LINQ além das coleções

Este capítulo cobre

- ÿ LINQ para SQL
- ÿ Consultas IQueryables e de árvore de expressão
- ÿ LINQ para XML
- ÿ LINQ paralelo
- ÿ Extensões reativas para .NET
- ÿ Escrevendo seus próprios operadores

Suponha que um alienígena o visite e peça que você descreva a cultura. Como você poderia capturar a diversidade da cultura humana em um curto espaço de tempo? Você pode decidir passar esse tempo mostrando-lhe a cultura em vez de descrevê-la de forma abstrata: uma visita a um clube de jazz de Nova Orleans, uma ópera no La Scala, a galeria do Louvre em Paris, uma peça de Shakespeare em Stratford-upon-Avon e breve.

Esse alienígena saberia tudo sobre cultura depois? Poderia compor uma música, escrever um livro, dançar um balé, criar uma escultura? Absolutamente não. Mas ele espera sair totalmente com um sentido de cultura – a sua riqueza e variedade, a sua capacidade de iluminar a vida das pessoas.

Assim é com este capítulo. Você já viu todos os recursos do C# 3, mas sem ver mais do LINQ, não terá contexto suficiente para realmente apreciá-los.

Quando a primeira edição deste livro foi publicada, não havia muitas tecnologias LINQ disponíveis, mas agora há uma abundância delas, tanto da Microsoft quanto de terceiros.

Isso por si só não me surpreendeu, mas fiquei fascinado ao ver as diferentes naturezas destas tecnologias.

Veremos várias maneiras pelas quais o LINQ se manifesta, com um exemplo de cada uma. Optei por demonstrar as principais tecnologias da Microsoft, porque são as mais típicas. Isso não significa que terceiros não sejam bem-vindos no ecossistema LINQ: há vários projetos, tanto comerciais quanto de código aberto, que fornecem acesso a diversas fontes de dados e criam recursos extras além dos provedores existentes.

Em contraste com o resto deste livro, iremos apenas dar uma olhada superficial em cada um dos tópicos aqui — o objetivo não é aprender os detalhes, mas mergulhar no espírito do LINQ. Para investigar mais a fundo qualquer uma dessas tecnologias, recomendo que você compre um livro dedicado ao assunto ou leia atentamente a documentação relevante. Resisti à tentação de dizer “Há mais no LINQ to [xxx] do que isso” no final de cada seção, mas por favor, leia-o como lido. Cada tecnologia tem muitos recursos além da consulta, mas concentrei-me aqui nas áreas diretamente relacionadas ao LINQ.

Vamos começar com o provedor que geralmente recebeu mais atenção quando o LINQ foi introduzido pela primeira vez: LINQ to SQL.

## 12.1 Consultando um banco de dados com LINQ to SQL

Tenho certeza de que agora você já absorveu a mensagem de que o LINQ to SQL converte expressões de consulta em SQL, que é então executado no banco de dados. É mais do que isso — é uma solução ORM completa — mas vou me concentrar no lado da consulta do LINQ to SQL em vez de entrar no tratamento de simultaneidade e nos outros detalhes com os quais um ORM precisa lidar. Mostrarei apenas o suficiente para que você mesmo possa experimentar — o banco de dados e o código estão disponíveis no site do livro (<http://csharpinprofundidade.com>). O banco de dados está no formato SQL Server 2005 para facilitar a utilização, mesmo se você não tiver a versão mais recente do SQL Server instalada, embora obviamente a Microsoft tenha garantido que o LINQ to SQL também funcione em versões mais recentes.

**POR QUE LINQ TO SQL EM VEZ DO ENTITY FRAMEWORK?** Falando em “versões mais recentes”, você pode estar se perguntando por que escolhi demonstrar o LINQ to SQL em vez do Entity Framework, que agora é a solução preferida da Microsoft (e que também suporta LINQ). A resposta é meramente simplicidade; o Entity Framework é sem dúvida mais poderoso que o LINQ to SQL de várias maneiras, mas requer conceitos extras que ocupariam muito espaço para serem explicados aqui. Estou tentando dar uma ideia da consistência (e da inconsistência ocasional) que o LINQ fornece, e isso é tão aplicável ao LINQ to SQL quanto ao Entity Framework.

Antes de começar a escrever qualquer consulta, você precisa de um banco de dados e de um modelo para representá-la no código.

### 12.1.1 Primeiros passos: o banco de dados e o modelo

LINQ to SQL precisa de metadados sobre o banco de dados para saber quais classes correspondem quais tabelas de banco de dados e assim por diante. Há várias maneiras de representar esses metadados e esta seção usará o designer LINQ to SQL integrado ao Visual Studio. Você pode projetar as entidades primeiro e solicitar ao LINQ para criar o banco de dados ou projetar seu banco de dados e deixar o Visual Studio definir a aparência das entidades. Pessoalmente, eu favorecemos a segunda abordagem, mas há prós e contras em ambas as formas.

#### CRIANDO O ESQUEMA DE BANCO DE DADOS

O mapeamento das classes do capítulo 11 para as tabelas do banco de dados é direto. Cada tabela tem uma coluna de ID de número inteiro com incremento automático com um nome apropriado: ProjectID, DefectID e assim por diante. As referências entre tabelas simplesmente usam o mesmo nome, então a tabela Defect possui uma coluna ProjectID , por exemplo, com uma chave estrangeira limitação.

Existem algumas exceções a este conjunto simples de regras:

- ÿ User é uma palavra reservada em T-SQL, então a classe User é mapeada para DefectUser mesa.
- ÿ As enumerações (status, gravidade e tipo de usuário) não possuem tabelas. Seus valores são mapeados para colunas tinyint nas tabelas Defect e DefectUser .
- ÿ A tabela Defect possui dois links para a tabela DefectUser : um para o usuário que criou o defeito e um para o destinatário atual. Estes são representados com as colunas CriadoById e AssignedToUserId , respectivamente.

#### CRIANDO AS CLASSES DE ENTIDADE

Depois que suas tabelas forem criadas, será fácil gerar as classes de entidade no Visual Studio. Basta abrir o Server Explorer (View > Server Explorer) e adicionar uma fonte de dados ao o banco de dados SkeetySoftDefects (clique com o botão direito em Conexões de dados e selecione Adicionar conexão). Você deverá conseguir ver quatro tabelas: Defect, DefectUser, Project e Assinatura de notificação.

Você pode então adicionar um novo item do tipo “classes LINQ to SQL ” ao projeto. Esse name será a base para uma classe gerada que representa o modelo geral do banco de dados; Usei o nome DefectModel, que leva a uma classe chamada DefectModelData-Context. O designer será aberto quando você criar o novo item.

Você pode então arrastar as quatro tabelas do Server Explorer para o designer e isso descobrir todas as associações. Depois disso, você pode reorganizar o diagrama e ajustar diversas propriedades das entidades. Aqui está uma lista do que mudei:

- ÿ Renomeei a propriedade DefectID para ID para corresponder ao modelo anterior.
- ÿ Renomeei DefectUser para User (portanto, embora a tabela ainda seja chamada de DefectUser, irá gerar uma classe chamada User, como antes).
- ÿ Alterei o tipo das propriedades Severidade, Status e UserType para seus enum equivalentes (tendo copiado essas enumerações para o projeto).

↳ Renomeei as propriedades pai e filho usadas para as associações entre Defect e DefectUser — o designer adivinhou nomes adequados para o outro associações, mas teve problemas aqui porque havia duas associações entre o mesmo par de tabelas. Nomeei os relacionamentos AssignedTo/Assigned Defects e CriadoBy/CreatedDefects.

A Figura 12.1 mostra o diagrama do designer após todas essas alterações. Como você pode ver, é se parece muito com o diagrama de classes da Figura 11.3, mas sem as enumerações.

Se você olhar no código C# gerado pelo designer (DefectModel.designer.cs), você encontrará cinco classes parciais: uma para cada uma das entidades e a classe DefectModelData Context que mencionei anteriormente. O fato de serem parciais é útil; neste caso eu adicionou construtores extras para corresponder aos das classes originais na memória, então o código do capítulo 11 para criar os dados de amostra pode ser reutilizado sem muito esforço extra trabalhar. Por uma questão de brevidade, não inclui o código de inserção aqui, mas se você olhar em PopulateDatabase.cs no código-fonte, você poderá segui-lo facilmente suficiente. Claro, você não precisa executar isso sozinho - o banco de dados para download é já povoado.

Agora que você tem um esquema em SQL, um modelo de entidade em C# e alguns dados de amostra, é hora de fazer perguntas.

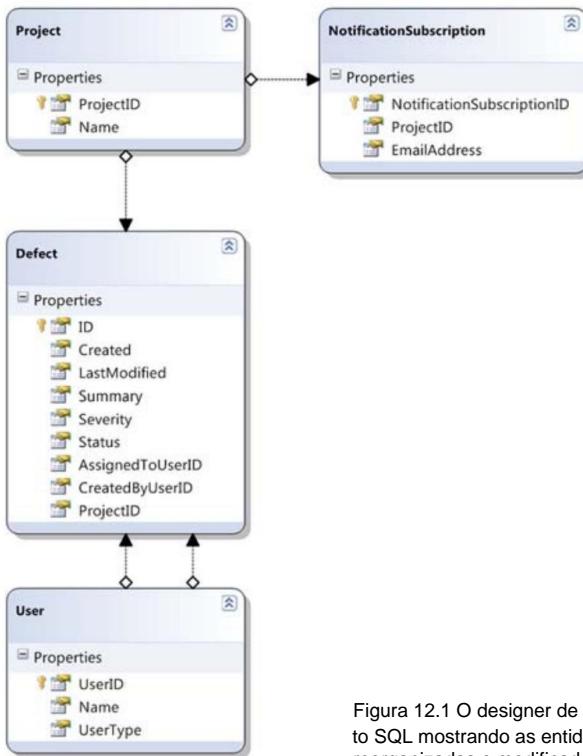


Figura 12.1 O designer de classes LINQ to SQL mostrando as entidades reorganizadas e modificadas

## 12.1.2 Consultas iniciais

Tenho certeza que você adivinhou o que está por vir, mas espero que isso não o torne menos impressionante. Executaremos expressões de consulta na fonte de dados, observando o LINQ to SQL converter a consulta em SQL instantaneamente. Para fins de familiaridade, usaremos algumas das mesmas consultas que executamos nas coleções na memória no capítulo 11.

### PRIMEIRA CONSULTA: ENCONTRANDO

**DEFEITOS ATRIBUÍDOS A TIM** Irei pular os exemplos triviais do início do capítulo 11 e começarei com a consulta da Listagem 11.7 que verifica defeitos abertos atribuídos a Tim. Aqui está a parte de consulta da listagem 11.7, para fins de comparação:

```
Usuário tim = SampleData.Users.TesterTim;

var query = do defeito em SampleData.AllDefects onde defect.Status !=  
    Status.Closed  
    onde defeito.AssignedTo == tim selecione  
        defeito.Summary;
```

O equivalente completo do LINQ to SQL da listagem 11.7 é o seguinte.

Listagem 12.1 Consultando o banco de dados para encontrar todos os defeitos abertos de Tim

```
usando (var context = new DefectModelDataContext()) {  
  
    contexto.Log = Console.Out;  
  
    Usuário tim = contexto.Usuários  
        .Where(usuário => usuário.Nome == "Tim Trotter")  
        .Solteiro();  
  
    var consulta = de defeito em context.Defects onde defeito.Status !=  
        Status.Closed onde defeito.AssignedTo == tim select  
            defeito.Summary;  
  
    foreach (var resumo na consulta) {  
  
        Console.WriteLine(resumo);  
    }  
}
```

A Listagem 12.1 requer uma certa explicação, porque é tudo novo. Primeiro, você cria um novo *contexto de dados* para trabalhar com B. Os contextos de dados são bastante multifuncionais, assumindo a responsabilidade pelo gerenciamento de conexões e transações, tradução de consultas, rastreamento de alterações em entidades e tratamento de identidade. Para os propósitos deste capítulo, você pode considerar um contexto de dados como seu ponto de contato com o banco de dados. Não mostro os recursos mais avançados aqui, mas você aproveita um recurso útil aqui: você diz ao contexto de dados para gravar todos os comandos SQL que ele executa no console C. As propriedades relacionadas ao modelo usadas no código para esta seção (Defeitos, Usuários e assim por diante) são todos do tipo `Table<T>` para o tipo de entidade relevante. Eles atuam como fontes de dados para suas consultas.

Você não pode usar SampleData.Users.TesterTim para identificar Tim na consulta principal porque esse objeto não conhece o ID da linha relevante na tabela DefectUser .

Em vez disso, você usa uma consulta separada para carregar a entidade de usuário D de Tim. Usei a notação de ponto para isso, mas uma expressão de consulta também teria funcionado. O método Single retorna apenas um único resultado de uma consulta, lançando uma exceção se não houver exatamente um elemento. Em uma situação da vida real, você pode ter a entidade como produto de outras operações, como fazer login, e se não tiver a entidade completa, poderá ter seu ID, que pode ser usado igualmente bem dentro a consulta principal. Como alternativa neste caso, você poderia alterar a consulta de defeitos abertos para filtrar com base no nome do responsável.

No entanto, isso não estaria de acordo com o espírito da consulta original.

Na expressão de consulta E, a única diferença entre a consulta na memória e a consulta LINQ to SQL é a fonte de dados – em vez de usar SampleData.All-Defects, você usa context.Defects. Os resultados finais são os mesmos (embora a ordem não seja garantida), mas o trabalho foi feito no banco de dados.

Como você solicitou ao contexto de dados para registrar o SQL gerado, você pode ver exatamente o que está acontecendo ao executar o código. A saída do console mostra ambas as consultas executadas no banco de dados, juntamente com os valores dos parâmetros de consulta:<sup>1</sup>

```
SELEÇÃO [t0].[UserID], [t0].[Nome], [t0].[UserType]
FROM [dbo].[DefectUser] AS [t0]
WHERE [t0].[Nome] = @p0 -- @p0:
String de entrada (Tamanho = 11; Prec = 0; Escala = 0) [Tim Trotter]

SELEÇÃO [t0].[Resumo]
FROM [dbo].[Defeito] AS [t0]
WHERE ([t0].[AssignedToUserID] = @p0) AND ([t0].[Status] <> @p1) - @p0: Entrada Int32 (Tamanho = 0;
Prec = 0; Escala = 0) [2] -- @p1: Entrada Int32 (Tamanho = 0; Prec = 0; Escala = 0) [4]
```

Observe como a primeira consulta busca todas as propriedades do usuário porque você está preenchendo uma entidade inteira, mas a segunda consulta busca apenas o resumo, pois isso é tudo que você precisa. LINQ to SQL também converteu as duas cláusulas where separadas na segunda consulta em um único filtro no banco de dados.

LINQ to SQL é capaz de traduzir uma ampla variedade de expressões. Vamos tentar um consulta um pouco mais complicada do capítulo 11, apenas para ver qual SQL é gerado.

#### **GERAÇÃO SQL PARA UMA CONSULTA MAIS COMPLEXA:**

**UMA CLÁUSULA LET** A próxima consulta mostra o que acontece quando você introduz uma espécie de variável temporária com uma cláusula let . No capítulo 11, consideramos uma situação bizarra, se você se lembra: fingir que calcular o comprimento de uma corda demorava muito tempo. Novamente, a expressão de consulta aqui é exatamente a mesma da listagem 11.11, com exceção da fonte de dados. A listagem a seguir mostra o código LINQ to SQL .

---

<sup>1</sup> A saída de log adicional é gerada mostrando alguns detalhes do contexto de dados, que omiti aqui para evitar distração do SQL. A saída do console também contém os resumos impressos pelo loop foreach , de curso.

**Listagem 12.2 Usando uma cláusula let no LINQ to SQL**

```
usando (var context = new DefectModelDataContext())
{
    contexto.Log = Console.Out;

    var query = do usuário em context.Users let length =
        user.Name.Length orderby length select new
        { Name = user.Name,
          Length = length };

    foreach (entrada var na consulta) {

        Console.WriteLine("{0}: {1}", entrada.Comprimento, entrada.Nome);
    }
}
```

O SQL gerado está próximo do espírito das sequências que vimos na figura 11.5. A sequência mais interna (a primeira nesse diagrama) é a lista de usuários; que é transformado em uma sequência de pares nome/comprimento (como a seleção aninhada) e, em seguida, a projeção não operacional é aplicada, com uma ordenação por comprimento:

```
SELEÇÃO [t1].[Nome], [t1].[valor]
FROM
    ( SELECT LEN([t0].[Nome]) AS [valor], [t0].[Nome]
      FROM [dbo].[DefectUser] AS [t0]
    ) COMO [t1]
ORDENAR POR [t1].[valor]
```

Este é um bom exemplo de onde o SQL gerado é mais prolixo do que deveria ser.

Embora não seja possível fazer referência aos elementos da sequência de saída final ao executar uma ordenação na expressão de consulta, você pode fazer isso no SQL. Esta consulta mais simples teria funcionado bem:

```
SELECT LEN([t0].[Nome]) AS [valor], [t0].[Nome]
FROM [dbo].[DefectUser] AS [t0]
ORDENAR POR [valor]
```

Obviamente, o que importa é o que o otimizador de consultas faz no banco de dados: o plano de execução exibido no SQL Server Management Studio Express é o mesmo para ambas as consultas, portanto, não parece que você está perdendo.

O conjunto final de consultas LINQ to SQL que veremos são todas junções.

### 12.1.3 Consultas envolvendo junções

Tentaremos junções internas e junções de grupo, usando os exemplos de associação de assinaturas de notificação em projetos. Suspeito que você já esteja acostumado com o exercício — o padrão do código é o mesmo para cada consulta, então daqui em diante mostrarei apenas a expressão da consulta e o SQL gerado, a menos que algo mais esteja acontecendo .

#### JUNÇÕES EXPLÍCITAS: DEFEITOS DE CORRESPONDÊNCIA COM ASSINATURAS DE NOTIFICAÇÃO

A primeira consulta é o tipo mais simples de junção – uma equijoin interna usando uma cláusula de junção LINQ :

```
// Expressão de consulta (modificada da listagem 11.12) do defeito em
context.Defects
ingressar na assinatura em contexto.NotificaçãoSubscrições em defeito.Projeto é igual a
assinatura.Projeto selecione novo {defeito.Summary, assinatura.EmailAddress}

-- SQL SELECT [t0].
[Resumo], [t1].[EmailAddress] gerado
FROM [dbo].[Defeito] AS [t0]
INNER JOIN [dbo].[NotificationSubscription] AS [t1]
ON [t0].[ProjectID] = [t1].[ProjectID]
```

Não é novidade que ele usa uma junção interna no SQL. Seria fácil adivinhar o SQL gerado neste caso. Que tal participar de um grupo? É aqui que as coisas ficam um pouco mais agitadas:

```
// Expressão de consulta (modificada da listagem 11.13) do defeito em
context.Defects
ingressar na assinatura no contexto.NotificaçãoSubscrições com defeito.Projeto é igual a
assinatura.Projeto em assinaturas agrupadas

selecione novo { Defeito = defeito, Assinaturas = assinaturas agrupadas }

-- SQL gerado SELECT
[t0].[DefectID] AS [ID], [t0].[Criado], [t0].[LastModified], [t0].[Resumo], [t0].
[Gravidade], [t0].[Status], [t0].[AssignedToUserID], [t0].[CreatedByUserID], [t0].[ProjectID],
[t1].[NotificationSubscriptionID], [t1].[ProjectID] AS [ProjectID2] ,
[t1].[Endereço de e-mail],
```

```
(SELECIONE CONTAGEM(*)
DE [dbo].[NotificationSubscription] AS [t2]
WHERE [t0].[ProjectID] = [t2].[ProjectID]) AS [contagem]
FROM [dbo].[Defeito] AS [t0]
LEFT OUTER JOIN [dbo].[NotificationSubscription] AS [t1]
ON [t0].[ProjectID] = [t1].[ProjectID]
ORDER BY [t0].[DefectID], [t1].[NotificationSubscriptionID]
```

Essa é uma grande mudança na quantidade de SQL gerada! Há duas coisas importantes a serem observadas. Primeiro, ele usa uma junção externa esquerda em vez de uma junção interna, então você ainda veria um defeito mesmo se não houvesse ninguém assinando seu projeto. Se você deseja uma junção externa esquerda, mas sem o agrupamento, a maneira convencional de expressar isso é usar uma junção de grupo e, em seguida, uma cláusula from extra, usando o método de extensão DefaultIfEmpty na sequência incorporada. Parece estranho, mas funciona bem.

A segunda coisa estranha sobre a consulta anterior é que ela calcula a contagem para cada grupo no banco de dados. Este é efetivamente um truque executado pelo LINQ to SQL para garantir que todo o processamento possa ser feito no servidor. Uma implementação ingênuo teria que realizar o agrupamento na memória após buscar todos os resultados. Em alguns casos, o provedor pode fazer truques para evitar a necessidade da contagem, simplesmente detectando quando o ID do agrupamento muda, mas há problemas com essa abordagem para algumas consultas. É possível que uma implementação posterior do LINQ to SQL seja capaz de mudar os cursos de ação dependendo da consulta exata.

Você não precisa escrever explicitamente uma junção na expressão de consulta para ver uma no SQL. Nossas consultas finais mostrarão junções criadas implicitamente por meio de expressões de acesso de propriedade.

#### JUNÇÕES IMPLÍCITAS: MOSTRANDO RESUMOS DE DEFEITOS E NOMEIS DE PROJETOS

Vejamos um exemplo simples. Suponha que você queira listar cada defeito, mostrando seu resumo e o nome do projeto do qual faz parte. A expressão de consulta é apenas uma questão de projeção:

```
// Consulta a expressão do
defeito em context.Defects
seleciona novo {defect.Summary, ProjectName = defect.Project.Name }

-- SQL SELECT [t0].
[Resumo], [t1].[Nome] gerado
FROM [dbo].[Defeito] AS [t0]
INNER JOIN [dbo].[Projeto] AS [t1]
ON [t1].[ProjectID] = [t0].[ProjectID]
```

Observe como você navega do defeito para o projeto por meio de uma propriedade: o LINQ to SQL converteu essa navegação em uma junção interna. Ele pode usar uma junção interna aqui porque o esquema tem uma restrição não anulável na coluna ProjectID da tabela Defect – cada defeito tem um projeto. Porém, nem todo defeito tem um destinatário, porque o campo AssignedToUserID é anulável; portanto, se você usar o destinatário em uma projeção, uma junção externa esquerda será gerada:

```
// Consulta a expressão do
defeito em context.Defects
seleciona novo {defect.Summary, Assignee = defect.AssignedTo.Name }

-- SQL SELECT [t0].
[Resumo], [t1].[Nome] gerado
FROM [dbo].[Defeito] AS [t0]
LEFT OUTER JOIN [dbo].[DefectUser] AS [t1]
ON [t1].[UserID] = [t0].[AssignedToUserID]
```

É claro que se você navegar por mais propriedades, as junções ficarão cada vez mais complicadas. Não vou entrar em detalhes aqui — o importante é que o LINQ to SQL precisa fazer muitas análises da expressão de consulta para descobrir qual SQL é necessário. Para realizar essa análise, é claramente necessário analisar a consulta que você especificou.

Vamos nos afastar especificamente do LINQ to SQL e pensar em termos gerais sobre o que os provedores de LINQ desse tipo precisam fazer. Isso se aplicará a qualquer provedor que precise examinar a consulta, em vez de apenas receber um delegado. Finalmente, é hora de ver por que as árvores de expressão foram adicionadas como um recurso do C# 3.

## 12.2 Traduções usando IQueryble e IQueryProvider

Nesta seção, mostrarei o básico de como o LINQ to SQL consegue converter suas expressões de consulta em SQL. Este é o ponto de partida para implementar o seu próprio

Provedor LINQ, se desejar. (Por favor, não subestime as dificuldades técnicas envolvidas em fazer isso - mas se você gosta de desafios, implementar um provedor LINQ é certamente interessante.) Esta é a seção mais teórica do capítulo, mas é útil para ter algumas dicas sobre como o LINQ decide se deve usar o processamento na memória, um banco de dados ou algum outro mecanismo de consulta.

Em todas as expressões de consulta que vimos no LINQ to SQL, a origem foi um Tabela<T>. Mas se você olhar para Table<T>, verá que não tem um método Where, ou Selecione ou Junte-se ou qualquer um dos outros operadores de consulta padrão. Em vez disso, ele usa o mesmo truque que o LINQ to Objects faz; assim como a fonte no LINQ to Objects sempre implementa IEnumerable<T> (possivelmente após uma chamada para Cast ou OfType) e então usa os métodos de extensão em Enumerable, então Table<T> implementa IQueryable<T> e em seguida, usa os métodos de extensão em Queryable. Você verá como o LINQ constrói um árvore de expressão e permite que um provedor a execute no momento apropriado.

Vamos começar examinando em que consiste IQueryable<T>.

#### 12.2.1 Apresentando IQueryable<T> e interfaces relacionadas

Se você procurar IQueryable<T> na documentação e ver quais membros ele contém diretamente (em vez de herdar), você poderá ficar desapontado. Não há nenhum. Em vez disso, ele herda de IEnumerable<T> e do IQueryable não genérico, que em turn herda do IEnumerable não genérico. Então IQueryable é onde o novo e membros interessantes são, certo? Bem, quase. Na verdade, IQueryable possui apenas três vínculos de propriedades: QueryProvider, ElementType e Expression. A propriedade QueryProvider é do tipo IQueryProvider — mais uma nova interface a ser considerada.

Perdido? Talvez a figura 12.2 ajude – é um diagrama de classes de todas as interfaces envolvidas diretamente.

A maneira mais fácil de pensar em IQueryable é que ele representa uma consulta que produz uma sequência de resultados quando você a executa. Os detalhes da consulta em termos LINQ são mantidos em uma árvore de expressão, conforme retornado pela propriedade Expression do IConsultável. A consulta é executada iterando através do IQueryable (em outros palavras, chamando o método GetEnumerator e, em seguida, MoveNext no resultado) ou chamando o método Execute em um IQueryProvider, passando uma árvore de expressão.

Agora que você tem pelo menos alguma noção da finalidade do IQueryable, o que é o IQuery Provider?  
Você pode fazer mais com uma consulta do que apenas executá-la; você também pode usá-lo para construir uma consulta maior, que é o objetivo dos operadores de consulta padrão no LINQ.<sup>2</sup> Construir criar uma consulta, você precisará usar o método CreateQuery no IQueryProvider relevante.<sup>3</sup>

Pense em uma fonte de dados como uma consulta simples (SELECT \* FROM SomeTable em SQL, por instância) — chamar Where, Select, OrderBy e métodos semelhantes resulta em um resultado diferente

<sup>2</sup> Bem, aqueles que continuam adiando a execução, como Where e Join. Você verá o que acontece com o agregações como Count daqui a pouco.

<sup>3</sup> Tanto Execute quanto CreateQuery possuem sobrecargas genéricas e não genéricas. As versões não genéricas tornam mais fácil criar consultas dinamicamente no código. As expressões de consulta em tempo de compilação usam a versão genérica.

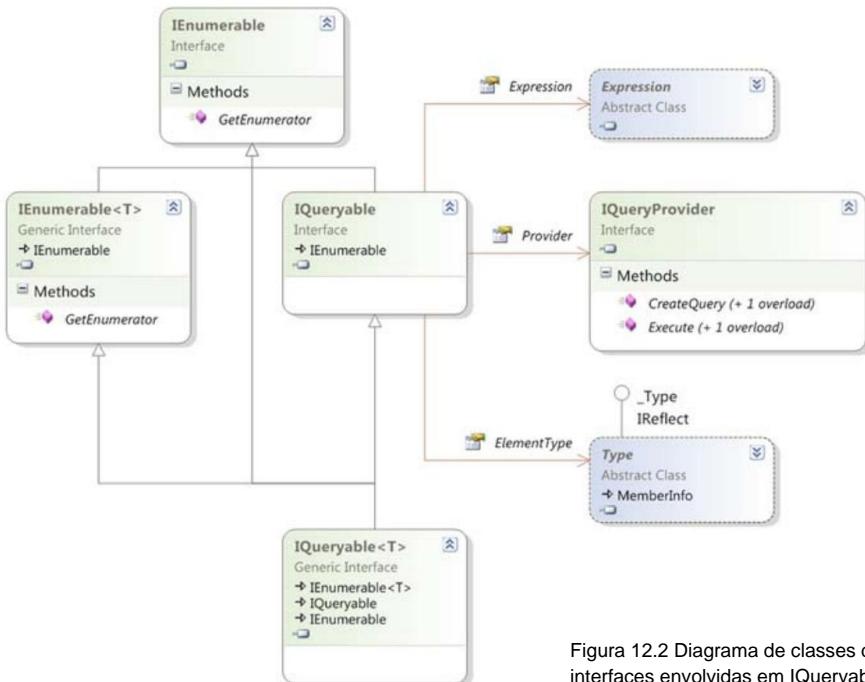


Figura 12.2 Diagrama de classes das interfaces envolvidas em `IQueryable<T>`

consulta baseada na primeira. Dada qualquer consulta `IQueryable`, você pode criar uma nova consulta executando as seguintes etapas:

- 1 Peça à consulta existente sua árvore de expressão de consulta (usando o método `Expression` propriedade).
- 2 Construa uma nova árvore de expressão que contenha a expressão original e a funcionalidade extra desejada (um filtro, projeção ou ordenação, por exemplo).
- 3 Peça à consulta existente seu provedor de consulta (usando a propriedade `Provider`).
- 4 Chame `CreateQuery` no provedor, passando a nova árvore de expressão.

Dessas etapas, a única complicada é criar a nova árvore de expressão. Felizmente, existem vários métodos de extensão na classe estática `Queryable` que podem fazer tudo isso para você. Chega de teoria – vamos começar a implementar as interfaces para que possamos ver tudo isso em ação.

### 12.2.2 Fingindo: implementações de interface para registrar chamadas

Antes que você fique muito animado, você não construirá um provedor de consultas completo neste capítulo. Mas se você entender tudo nesta seção, você estará em uma posição muito melhor para construir uma se precisar — e possivelmente mais importante, você entenderá o que está acontecendo quando você emitir consultas LINQ to SQL. A maior parte do trabalho árduo dos provedores de consulta ocorre no ponto de execução, onde eles precisam analisar uma árvore de expressão e convertê-la no formato apropriado para a plataforma de destino.

Vamos nos concentrar no trabalho que acontece antes disso – como o LINQ se prepara para executar uma consulta.

Veremos implementações de `IQueryable` e `IQueryProvider` e tentaremos executar algumas consultas neles. A parte interessante não são os resultados – as consultas não farão nada de útil – mas a série de chamadas feitas até o ponto de execução.

Vamos nos concentrar em dois tipos: `FakeQueryProvider` e `FakeQuery`. A implementação de cada método de interface escreve a expressão atual envolvida, usando um método de registro simples (não mostrado aqui).

Vejamos primeiro o `FakeQuery`, conforme mostrado na listagem a seguir.

**Listagem 12.3 Uma implementação simples de `IQueryable` que registra chamadas de método**

```
classe FakeQuery<T> : IQueryable<T>
{
    Expressão pública Expressão {obter; conjunto privado; } provedor público
    IQueryProvider { get; conjunto privado; } public Type ElementType { get; conjunto
    privado; }

    FakeQuery interno (provedor IQueryProvider,
                       Expressão expressão)
    {
        Expressão = expressão;
        Provedor = provedor;
        ElementType = tipo de(T);
    }

    FakeQuery interno(): this(new FakeQueryProvider(), null) {

        Expressão = Expressão.Constant(este);
    }

    público IEnumerator<T> GetEnumerator()
    {
        Logger.Log(este, Expressão); retornar
        Enumerable.Empty<T>().GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator() {

        Logger.Log(este, Expressão); retornar
        Enumerable.Empty<T>().GetEnumerator();
    }

    string de substituição pública ToString() {
        retornar "FakeQuery";
    }
}
```

- A Declara automático simples Propriedades B
- B Expressão C Usa esta consulta como inicial
- C D Retorna sequência de resultados vazia
- E Substitui ToString para fins de registro

Os membros das propriedades de `IQueryable` são implementados em `FakeQuery` com propriedades automáticas B, que são definidas pelos construtores. Existem dois construtores: um sem parâmetros que é usado pelo programa principal para criar uma fonte simples para a consulta e outro que é chamado por `FakeQueryProvider` com a expressão de consulta atual.

O uso de Expression.Constant(this) como expressão de origem inicial **C** é apenas uma forma de mostrar que a consulta representa inicialmente o objeto original. (Imagine uma implementação representando uma tabela, por exemplo – até que você aplique qualquer operador de consulta, a consulta retornará apenas a tabela inteira.) Quando a expressão constante é registrada, ela usa o método ToString substituído, e é por isso que você fornece um descrição curta e constante e. Isso torna a expressão final muito mais limpa do que seria sem a substituição. Quando você é solicitado a iterar os resultados da consulta, você sempre retorna uma sequência vazia **D** para facilitar a vida. As implementações de produção analisariam a expressão aqui ou (mais provavelmente) chamariam Execute em seu provedor de consulta e retornariam o resultado.

Como você pode ver, não há muita coisa acontecendo no FakeQuery, e a listagem a seguir mostra esse FakeQueryProvider também é simples.

#### Listagem 12.4 Uma implementação de IQueryProvider que usa FakeQuery

```
classe FakeQueryProvider: IQueryProvider {
    public IQueryables CreateQuery<T>(expressão de expressão)
    {
        Logger.Log(isto, expressão); return new
        FakeQuery<T>(isto, expressão);
    }

    public IQueryables CreateQuery(Expression expressão) {
        Digite queryType = typeof(FakeQuery<>).MakeGenericType(expression.Type); object[] construtorArgs = new object[]
        { this, expressão }; return (IQueryables)Activator.CreateInstance(queryType, construtorArgs);
    }

    public T Execute<T>(Expressão expressão) {
        Logger.Log(isto, expressão); retornar padrão(T);
    }

    objeto público Executar(Expressão expressão) {
        Logger.Log(isto, expressão); retornar nulo;
    }
}
```

Há ainda menos a dizer sobre a implementação de FakeQueryProvider do que sobre FakeQuery<T>. Os métodos CreateQuery não realizam nenhum processamento real, mas atuam como métodos de fábrica para a consulta. A única parte complicada é que a sobrecarga não genérica ainda precisa fornecer o argumento de tipo correto para FakeQuery<T> com base na propriedade Type da expressão fornecida. As sobrecargas do método Execute retornam resultados vazios após registrar a chamada. É aqui que *normalmente* seriam feitas muitas análises , juntamente com a chamada real para o serviço web, banco de dados ou outra plataforma de destino.

Mesmo que você não tenha feito nenhum trabalho real, coisas interessantes começam a acontecer quando você começa a usar `FakeQuery` como fonte em uma expressão de consulta. Já deixei escapar que você é capaz de escrever expressões de consulta sem escrever explicitamente métodos para lidar com os operadores de consulta padrão: trata-se de métodos de extensão — desta vez, os da classe `Queryable`.

#### 12.2.3 Unindo expressões: os métodos de extensão `Queryable`

Assim como o tipo `Enumerable` contém métodos de extensão em `IEnumerable<T>` para implementar os operadores de consulta padrão LINQ, o tipo `Queryable` contém métodos de extensão em `IQueryable<T>`. Existem duas grandes diferenças entre as implementações em `Enumerable` e aquelas em `Queryable`.

Primeiro, todos os métodos `Enumerable` usam delegados como parâmetros - o método `Select` usa `Func<TSource, TResult>`, por exemplo. Isso é bom para manipulação na memória, mas para provedores LINQ que executam a consulta em outro lugar, você precisa de um formato que possa examinar mais de perto: árvores de expressão. Por exemplo, a sobrecarga correspondente de `Select` em `Queryable` leva um parâmetro do tipo `Expression<Func<TSource, TResult>>`. O compilador não se importa - após a tradução da consulta, ele tem uma expressão lambda que precisa passar como argumento para o método, e as expressões lambda podem ser convertidas em instâncias de delegação ou em árvores de expressão.

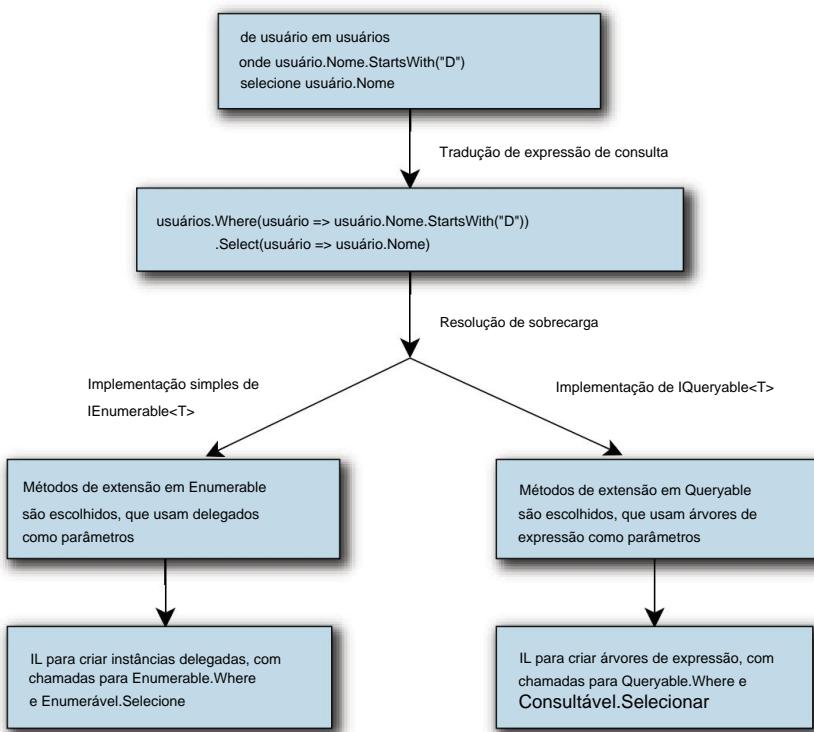
É assim que o LINQ to SQL pode funcionar perfeitamente. Os quatro elementos principais envolvidos são todos novos recursos do C# 3: expressões lambda, a tradução de expressões de consulta em expressões normais que usam expressões lambda, métodos de extensão e árvores de expressão. Sem todos os quatro, haveria problemas. Se as expressões de consulta sempre fossem traduzidas em delegados, por exemplo, elas não poderiam ser usadas com um provedor como LINQ to SQL, que requer árvores de expressão. A Figura 12.3 mostra dois caminhos possíveis percorridos pelas expressões de consulta; eles diferem apenas nas interfaces que sua fonte de dados implementa.

Observe como na Figura 12.3 as partes iniciais do processo de compilação são independentes da fonte de dados. A mesma expressão de consulta é usada e é traduzida exatamente da mesma maneira. Somente quando o compilador analisa a consulta traduzida para encontrar os métodos `Select` e `Where` apropriados para usar é que a fonte de dados é realmente importante.

Nesse ponto, as expressões lambda podem ser convertidas em instâncias de delegação ou em árvores de expressão, potencialmente fornecendo implementações radicalmente diferentes: normalmente na memória para o caminho esquerdo e SQL sendo executado em um banco de dados no caminho certo.

Apenas para enfatizar um ponto familiar, a decisão na Figura 12.3 de usar `Enumerable` ou `Queryable` não tem suporte explícito no compilador C#. Esses não são os únicos dois caminhos possíveis, como você verá mais tarde com Parallel LINQ e Reactive LINQ. Você pode criar sua própria interface e implementar métodos de extensão seguindo o padrão de consulta, ou até mesmo criar um tipo com métodos de instância apropriados.

A segunda grande diferença entre `Enumerable` e `Queryable` é que os métodos de extensão `Enumerable` fazem o trabalho real associado ao operador de consulta correspondente (ou pelo menos constroem iteradores que fazem o trabalho). Há código em



**Figura 12.3** Uma consulta que segue dois caminhos, dependendo se a fonte de dados implementa IQueryable ou apenas IEnumerable

Enumerable.Where para executar o filtro especificado e produzir apenas os elementos apropriados como a sequência de resultados, por exemplo. Por outro lado, as implementações do operador de consulta em Queryable fazem pouco: eles apenas criam uma nova consulta com base nos parâmetros ou chame Execute no provedor de consulta, conforme descrito no final da seção 12.2.1. Em outras palavras, eles são usados apenas para criar consultas e solicitar que sejam executados - eles não contêm a lógica por trás dos operadores. Isso significa que eles são adequados para qualquer provedor LINQ que usa árvores de expressão, mas elas são inúteis por si só. Eles estão a cola entre o seu código e os detalhes do provedor.

Com os métodos de extensão Queryable disponíveis e prontos para usar o IQueryable e implementações de IQueryable, finalmente chegou a hora de ver o que acontece quando você usa uma expressão de consulta com seu provedor personalizado.

#### 12.2.4 O falso provedor de consultas em ação

A Listagem 12.5 mostra uma expressão de consulta simples, que (supostamente) encontra todas as strings na fonte falsa, começando com abc, e projeta os resultados em uma listagem de sequência os comprimentos das strings correspondentes. Você percorre os resultados, mas não faz nada com eles, pois já sabe que eles estarão vazios. Isso é porque você tem

nenhum dado de origem e você não escreveu nenhum código para fazer nenhuma filtragem real - você está apenas registrando quais chamadas são feitas pelo LINQ durante a criação da expressão de consulta e iterando pelos resultados.

#### Listagem 12.5 Uma expressão de consulta simples usando classes de consulta falsas

```
var query = de x em new FakeQuery<string>()
    onde x.StartsWith("abc") selecione
        x.Length; foreach (int i na
    consulta) { }
```

Quais você esperaria que fossem os resultados da execução da listagem 12.5? Em particular, o que você gostaria que fosse registrado por último, no ponto em que normalmente esperaria fazer algum trabalho real com a árvore de expressões? Aqui estão os resultados, ligeiramente reformatados para maior clareza:

```
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))

FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
    .Selecione(x => x.Comprimento)

FakeQuery<Int32>.GetEnumerator
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
    .Selecione(x => x.Comprimento)
```

As duas coisas importantes a serem observadas são que GetEnumerator é chamado apenas no final, e não em consultas intermediárias; no momento em que GetEnumerator é chamado, você tem todas as informações presentes na expressão de consulta original. Você não teve que acompanhar manualmente as partes anteriores da expressão em cada etapa – uma única árvore de expressão captura todas as informações até agora.

A propósito, não se deixe enganar pela saída concisa - a árvore de expressão real é profunda e complicada, principalmente devido à cláusula where que inclui uma chamada de método extra. Essa árvore de expressão é o que o LINQ to SQL examinará para descobrir qual consulta executar. Os provedores LINQ podem criar suas próprias consultas (em qualquer formato que necessitem) quando chamadas para CreateQuery são feitas, mas normalmente é mais simples olhar para a árvore final quando GetEnumerator é chamado, porque todas as informações necessárias estão disponíveis em um só lugar.

A chamada final registrada pela listagem 12.5 foi para FakeQuery.GetEnumerator, e você pode estar se perguntando por que também precisa de um método Execute em IQueryProvider. Bem, nem todas as expressões de consulta geram sequências. Se você usar um operador de agregação como Soma, Contagem ou Média, não estará mais criando uma fonte — estará avaliando um resultado imediatamente. É quando Execute é chamado, conforme mostrado na listagem a seguir e sua saída.

#### Listagem 12.6 IQueryProvider.Execute

```
var query = de x em new FakeQuery<string>()
    onde x.StartsWith("abc")
    selecione x.Comprimento;
```

```

média dupla = query.Average();

// Saída
FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))

FakeQueryProvider.CreateQuery
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
    .Selecione(x => x.Comprimento)

FakeQueryProvider.Execute
Expression=FakeQuery.Where(x => x.StartsWith("abc"))
    .Selecione(x => x.Comprimento)
    .Média()

```

O FakeQueryProvider pode ser bastante útil quando se trata de entender o que o compilador C# está fazendo nos bastidores com expressões de consulta. Ele mostrará os identificadores transparentes introduzidos em uma expressão de consulta, juntamente com as chamadas traduzidas para SelectMany, GroupJoin e similares.

### 12.2.5 Concluindo IQuerybable

Você não escreveu nenhum código significativo que um provedor de consulta real precisaria para realizar um trabalho útil, mas esperamos que esse provedor falso tenha lhe dado uma ideia de como os provedores LINQ obtêm suas informações a partir de expressões de consulta. Tudo é construído pelos métodos de extensão Queryable , dada uma implementação apropriada de IQuerybable e IQueryProvider.

Entramos em mais detalhes nesta seção do que no restante do capítulo, pois ela envolve os fundamentos que sustentam o código LINQ to SQL que vimos anteriormente. Mesmo que seja improvável que você precise implementar interfaces de consulta sozinho, as etapas envolvidas na obtenção de uma expressão de consulta C# e (em tempo de execução) na execução de algum SQL em um banco de dados são bastante profundas e estão no centro dos grandes recursos do C# 3 Entender por que o C# ganhou esses recursos ajudará a mantê-lo mais sintonizado com a linguagem.

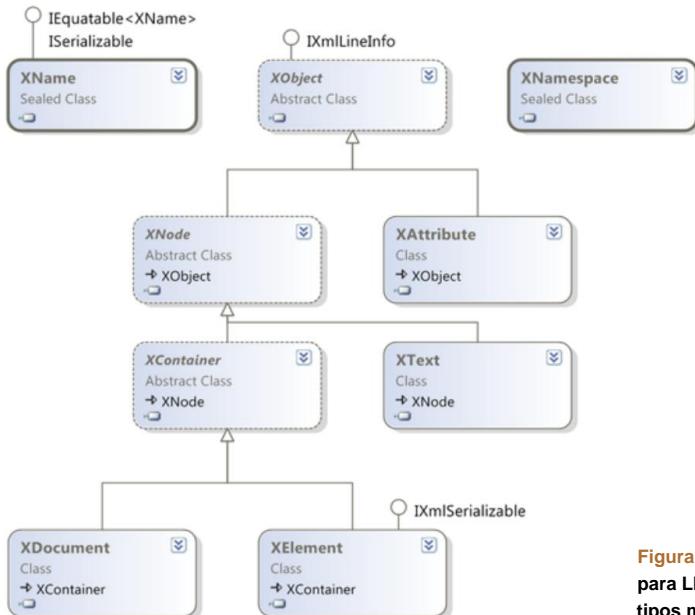
Este é o fim da nossa cobertura do LINQ usando árvores de expressão. O resto do capítulo envolve consultas em processo usando delegados, mas como você verá, ainda pode haver muita variedade e inovação na forma como o LINQ pode ser usado. Nossa primeira parada é o LINQ to XML, que é “apenas” uma API XML projetada para se integrar bem com o LINQ to Objects.

## 12.3 APIs compatíveis com LINQ e LINQ to XML

LINQ to XML é de longe a API XML mais agradável que já usei. Esteja você consumindo XML existente, gerando um novo documento ou um pouco de ambos, é fácil de usar e entender. Parte disso é completamente independente do LINQ, mas muito se deve à forma como ele interage com o restante do LINQ. Como na seção 12.1, fornecerei informações introdutórias suficientes para entender os exemplos e, em seguida, você verá como o LINQ to XML combina seus próprios operadores de consulta com os do LINQ to Objects. Ao final da seção, você poderá ter algumas ideias sobre como fazer suas próprias APIs funcionarem em harmonia com a estrutura.

### 12.3.1 Tipos principais em LINQ to XML

LINQ to XML reside no assembly System.Xml.Linq e a maioria dos tipos está no Namespace System.Xml.Linq.<sup>4</sup> Quase todos os tipos nesse namespace têm um prefixo de X, então enquanto a API DOM normal tem um tipo XmlDocument , o LINQ to XML equivalente é XElement . Isso facilita identificar quando o código está usando LINQ to XML, mesmo que você não esteja imediatamente familiarizado com o tipo exato envolvido. A Figura 12.4 mostra os tipos que você usará com mais frequência.



**Figura 12.4** Diagrama de classes para LINQ to XML, mostrando os tipos mais comumente usados

Aqui está um breve resumo dos tipos mostrados:

- ÿ XName é usado para nomes de elementos e atributos. As instâncias geralmente são criadas usando uma conversão implícita de uma string (nesse caso, nenhum namespace é usado) ou através do operador sobrecarregado +(XNamespace, string) .
- ÿ XNamespace representa um namespace XML – basicamente um URI . As instâncias geralmente são criadas pela conversão implícita de string.
- ÿ XObject é o ancestral comum de XNode e XAttribute; ao contrário do que acontece API DOM, um atributo não é um nó em LINQ to XML. Métodos retornando filho nós não incluem atributos, por exemplo.
- ÿ XNode representa um nó na árvore XML . Ele define vários membros para manipular e consultar a árvore. Existem várias outras classes derivadas do XNode que não são mostradas na figura 12.4, como XComment e XDeclaration. Estes são

<sup>4</sup> Eu costumava esquecer se era System.Xml.Linq ou System.Linq.Xml. Se você lembrar que é um XML API em primeiro lugar, você deve estar bem.

usado com pouca frequência – os tipos de nó mais comuns são documentos, elementos e texto.

þ XAttribute é um atributo com um nome e um valor. O valor é intrinsecamente texto, mas há conversões explícitas para muitos outros tipos de dados, como int e DateTime.

þ XContainer é um nó na árvore XML que pode ter conteúdo filho – é um elemento mento ou um documento, basicamente.

þ XText é um nó de texto e um tipo derivado adicional, XCData, é usado para representar nós de texto CDATA. (Um nó CDATA é aproximadamente equivalente a um literal de string literal – menos escape é necessário.) XText raramente é instanciado diretamente no código do usuário; em vez disso, quando uma string é usada como conteúdo de um elemento ou documento, ela é convertida em uma instância XText.

þ XElement é um elemento. Esta é a classe mais comumente usada em LINQ to XML, junto com XAttribute. Ao contrário da API DOM, você pode criar um XElement sem criar um documento para contê-lo. A menos que você realmente precise de um objeto de documento (talvez para uma declaração XML personalizada), muitas vezes você pode usar apenas elementos. þ XDocument é um documento. Seu elemento raiz é acessado usando a propriedade Root — isso é equivalente a XmlDocument.DocumentElement. Conforme observado anteriormente, isso geralmente não é necessário.

Mais tipos estão disponíveis até mesmo dentro do modelo de documento, e existem alguns outros tipos para opções como carregar e salvar — mas esta lista cobre os mais importantes. Dos tipos anteriores, os únicos que você precisa fazer referência explícita regularmente são XElement e XAttribute. Se você usar namespaces, também usará XNamespace, mas a maior parte do restante dos tipos poderá ser ignorada no resto do tempo. É incrível o quanto você pode fazer com tão poucos tipos.

Falando em incrível, não resisto em mostrar como funciona o suporte ao namespace no LINQ to XML. Não usaremos namespaces em nenhum outro lugar, mas é um bom exemplo de como um conjunto bem projetado de conversões e operadores pode facilitar a vida. Isso também nos facilitará o próximo tópico: construção de elementos.

Se você precisar apenas especificar o nome de um elemento ou atributo sem namespace, poderá usar uma string. Você não encontrará nenhum construtor para nenhum dos tipos com parâmetros do tipo string - todos eles aceitam um XName. Existe uma conversão implícita de string para XName e também de string para XNamespace. Adicionar um namespace e uma string também fornece um XName. Há uma linha tênue entre o abuso do operador e a genialidade, mas neste caso o LINQ to XML realmente faz com que funcione.

Aqui está um código para criar dois elementos – um dentro de um namespace e outro não:

```
XElement noNamespace = new XElement("sem namespace"); XNamespace ns  
= "http://csharpinprofundidade.com/sample/namespace"; XElement withNamespace = new  
XElement(ns + "no namespace");
```

Isso torna o código legível mesmo quando namespaces estão envolvidos, o que é um alívio bem-vindo em algumas outras APIs. Mas isso apenas cria dois elementos vazios. Como você dá a eles algum conteúdo?

### 12.3.2 Construção declarativa

Normalmente na API DOM, você cria um elemento e depois adiciona conteúdo a ele. Você pode fazer isso no LINQ to XML por meio do método Add herdados do XContainer, mas essa não é a maneira idiomática do LINQ to XML de fazer as coisas.<sup>5</sup> Porém, ainda vale a pena dar uma olhada na assinatura do XContainer.Add, porque ela apresenta o conteúdo modelo. Você poderia esperar uma assinatura de Add(XNode) ou talvez Add(XObject), mas é apenas Add(object). O mesmo padrão é usado para as assinaturas dos construtores XElement (e XDocument). Todos os construtores XElement têm um parâmetro para o nome do elemento, mas depois disso você não pode especificar nada (para criar um elemento vazio), um único objeto (para criar um elemento com um único nó filho) ou uma matriz de objetos para crie vários nós filhos. No caso de vários filhos, um array de parâmetros é usado (a palavra-chave params em C#), o que significa que o compilador criará o array para você – você pode simplesmente continuar listando os argumentos.

O uso de objeto simples para o tipo de conteúdo pode parecer loucura, mas é extremamente útil. Quando você adiciona conteúdo, seja por meio de um construtor ou do método Add, os seguintes pontos são considerados:

- ÿ Referências nulas são ignoradas.
- ÿ Instâncias XNode e XAttribute são adicionadas de maneira relativamente simples; eles são clonados se já tiverem pais, mas caso contrário, nenhuma conversão será necessária. (Algumas outras verificações de integridade são realizadas, como garantir que você não tenha atributos duplicados em um único elemento.) ÿ Strings, números, datas, horas e assim por diante são adicionados convertendo-os em Nós XText usando formatação XML padrão.
- ÿ Se o argumento implementa IEnumerable (e não é coberto por mais nada), Add irá iterar sobre seu conteúdo e adicionar cada valor por vez, recorrendo onde necessário.
- ÿ Qualquer coisa que não tenha tratamento de casos especiais é convertida em texto apenas chamando ToString().

Isso significa que muitas vezes você não precisa preparar seu conteúdo de uma maneira especial antes de adicioná-lo a um elemento — o LINQ to XML faz a coisa certa para você. Os detalhes são explicitamente documentados, então você não precisa se preocupar com o fato de ser muito mágico — mas realmente funciona.

A construção de elementos aninhados leva a um código que se assemelha naturalmente à estrutura hierárquica da árvore. Isso é melhor mostrado com um exemplo. Aqui está um trecho do código LINQ to XML :

```
novo XElement("raiz",
    new XElement("filho", new
        XElement("neto", "texto"), new XElement("outro-filho"));
```

---

<sup>5</sup> De certa forma, é uma pena que XElement não implemente IEnumerable, caso contrário, inicializadores de coleção seriam outra abordagem para construção. Mas usar o construtor funciona perfeitamente de qualquer maneira.

E aqui está o XML do elemento criado — observe a semelhança visual entre o código e a saída:

```
<raiz>
  <criança>
    <grandchild>texto</grandchild> </child> <other-child />
  </root>
```

Até agora, tudo bem, mas a parte importante é o quarto item da lista anterior, onde as sequências são processadas recursivamente, porque isso permite construir uma estrutura XML a partir de uma consulta LINQ de maneira natural. Por exemplo, o site do livro possui algum código para gerar um feed RSS a partir de seu banco de dados. A instrução para construir o documento XML tem 28 linhas – o que eu normalmente esperaria ser uma abominação – mas é extremamente agradável de ler.<sup>6</sup> Essa instrução contém duas consultas LINQ – uma para preencher um valor de atributo e outra para preencher um valor de atributo. O outro para fornecer uma sequência de elementos, cada um representando uma notícia. À medida que você lê o código, fica óbvio como será o XML resultante.

Para tornar isso mais concreto, tomemos dois exemplos simples do sistema de rastreamento de defeitos. Demonstrarei o uso dos dados de exemplo do LINQ to Objects, mas você pode usar consultas quase idênticas para trabalhar com outro provedor LINQ. Primeiro, você precisa construir um elemento contendo todos os usuários do sistema. Nesse caso, você só precisa de uma projeção, então a listagem a seguir usa a notação de ponto.

#### Listagem 12.7 Criando elementos a partir dos usuários de amostra

```
var usuários = new XElement("usuários",
    SampleData.AllUsers.Select(usuário => novo XElement("usuário",
        new XAttribute("nome", usuário.Nome), new XAttribute("tipo",
        usuário.UserType)))
);
Console.WriteLine(usuários);

// Saída
<usuários>
  <nome de usuário="Tim Trotter" tipo="Testador" /> <nome de usuário="Tara
  Tutu" tipo="Testador" /> <nome de usuário="Deborah Denton"
  type="Desenvolvedor" /> <nome de usuário=" Darren Dahlia" type="Desenvolvedor" />
  <user name="Mary Malcop" type="Manager" /> <user name="Colin Carton"
  type="Cliente" /> </users>
```

Se você quiser fazer uma consulta um pouco mais complexa, provavelmente valerá a pena usar uma expressão de consulta. A listagem a seguir cria outra lista de usuários, mas desta vez apenas o

<sup>6</sup> Um fator que contribui para a legibilidade é um método de extensão que criei para converter tipos anônimos em elementos, usando as propriedades dos elementos filhos. Se você estiver interessado, o código está disponível gratuitamente como parte do meu projeto MiscUtil (consulte <http://mng.bz/xDMt>). Isso só ajuda quando a estrutura XML necessária se ajusta a um determinado padrão, mas nesse caso pode reduzir significativamente a confusão de chamadas do construtor XElement .

desenvolvedores dentro da SkeetySoft. Para variar, desta vez o nome de cada desenvolvedor é um nó de texto dentro de um elemento em vez de um valor de atributo.

#### Listagem 12.8 Criando elementos com nós de texto

```
var desenvolvedores = new XElement("desenvolvedores", do
    usuário em SampleData.AllUsers onde
        user.UserType == UserType.Developer selecione novo
        XElement("developer", user.Name)
    );
Console.WriteLine(desenvolvedores);

// Saída
<desenvolvedores>
    <desenvolvedor>Deborah Denton</desenvolvedor>
    <desenvolvedor>Darren Dahlia</desenvolvedor>
</desenvolvedores>
```

Esse tipo de coisa pode ser aplicado a todos os dados de amostra, produzindo uma estrutura de documento como esta:

```
<sistema de defeitos>
    <projetos>
        <nome do projeto="..." id="...">
            <subscription email="..." /> </project> </
        projects>

    <usuários>
        <nome do usuário="..." id="..." type="..." /> </users> <defeitos>

        <defeito id="..." resumo="..." criado="..." projeto="..." atribuído a="..." criado-por="..." status=
            "..." gravidade="..." last-modified="..." /> </defects> </defect-system>
```

Você pode ver o código para gerar tudo isso em XmlSampleData.cs na solução para download. Ele demonstra uma alternativa à abordagem de uma grande declaração: cada um dos elementos no nível superior é criado separadamente e depois colados assim:

```
XElement root = new XElement("sistema de defeitos", projetos, usuários, defeitos);
```

Usaremos esse XML para demonstrar o próximo ponto de integração do LINQ : consultas. Vamos começar com os métodos de consulta disponíveis em um único nó.

### 12.3.3 Consultas em nós únicos

Você pode estar esperando que eu revele que o XElement implementa I Enumerable e que as consultas LINQ são gratuitas. Não é tão simples assim, porque há muitas coisas diferentes pelas quais um XElement poderia iterar. XElement contém vários *métodos de eixo* que são usados como fontes de consulta. Se você estiver familiarizado com XPath, a ideia de eixo sem dúvida será familiar para você.

Aqui estão os métodos de eixo usados diretamente para consultar um único nó, cada um dos quais retorna um `IEnumerable<T>` apropriado:

ÿ Ancestrais	ÿ Nós Descendentes
ÿ Anotações	ÿ Elementos
ÿ Descendentes	ÿ ElementsBeforeSelf
ÿ Ancestrais e eu	ÿ DescendantNodesAndSelf
ÿ Atributos	ÿ ElementsAfterSelf
ÿ DescendentesEPróprio	ÿ Nós

Tudo isso é bastante autoexplicativo (e a documentação do MSDN fornece mais detalhes). Existem sobrecargas úteis para recuperar apenas nós com um nome apropriado; chamar `Descendants("user")` em um `XElement` retornará todos os elementos do usuário abaixo do elemento que você o chamou, por exemplo.

Além dessas chamadas retornarem sequências, alguns métodos retornam um único resultado — `Atributo` e `Elemento` são os mais importantes, retornando o atributo nomeado e o primeiro elemento filho com o nome especificado, respectivamente. Adicionalmente, existem conversões explícitas de um `XAttribute` ou `XElement` para qualquer número de outros tipos, como `int`, `string` e `DateTime`. Eles são importantes tanto para filtragem e projetar resultados. Cada conversão para um tipo de valor não anulável também tem uma conversão para seu equivalente anulável — estas (e a conversão para `string`) retornam um valor nulo. valor se você invocá-los em uma referência nula. Esta propagação nula significa que você não tem que verificar a presença ou ausência de atributos ou elementos na consulta— você pode usar os resultados da consulta.

O que isso tem a ver com o LINQ? Bem, o fato de vários resultados de pesquisa serem retornado em termos de `IEnumerable<T>` significa que você pode usar o LINQ to Objects normal métodos depois de encontrar alguns elementos. A listagem a seguir mostra um exemplo de localização dos nomes e tipos dos usuários, desta vez começando com os dados de amostra em XML.

#### Listagem 12.9 Exibindo os usuários dentro de uma estrutura XML

```
Raiz XElement = XmlSampleData.GetElement();

var consulta = root.Element("usuários").Elements().Select(usuario => novo
{
    Nome = (string) user.Attribute("nome"),
    UserType = (string) user.Attribute("tipo")
});

foreach (var usuário na consulta)
{
    Console.WriteLine ("{0}: {1}", usuário.Nome, usuário.UserType);
}
```

Depois de criar os dados no início, você navega até o elemento `users` e pergunta para seus elementos filhos diretos. Essa busca em duas etapas pode ser reduzida para apenas `root.Descendants("user")`, mas é bom saber sobre a navegação mais rígida para você pode usá-lo quando necessário. Também é mais robusto diante de mudanças no

estrutura do documento, como outro elemento de usuário (não relacionado) sendo adicionado em outro local do documento.

O restante da expressão de consulta é meramente uma projeção de um XElement em um tipo anônimo. Admito que isso é um pouco trapaceiro com o tipo de usuário: ele é mantido como uma string em vez de chamar Enum.Parse para convertê-lo em um valor UserType adequado. A última abordagem funciona perfeitamente bem, mas é bastante demorada quando você só precisa do formato de string, e o código fica difícil de formatar de maneira sensata dentro dos limites estritos da página impressa.

Não há nada de especial aqui – afinal, retornar resultados de consulta como sequências é bastante comum. Vale a pena notar como você pode passar facilmente de operadores de consulta específicos de domínio para operadores de uso geral. Esse não é o fim da história, no entanto. LINQ to XML também possui alguns métodos de extensão extras para adicionar.

#### 12.3.4 Operadores de consulta nivelados

Você viu como o resultado de uma parte de uma consulta geralmente é uma sequência e, no LINQ to XML, geralmente é uma sequência de elementos. E se você quisesse realizar uma consulta específica XML em cada um desses elementos? Para apresentar um exemplo um tanto artificial, você pode encontrar todos os projetos nos dados de amostra com root.Element("projects").Elements(), mas como você pode encontrar os elementos de assinatura dentro deles? Você precisa aplicar outra consulta a cada elemento e então nivelar os resultados. (Novamente, você poderia usar root.Descendants("subscription"), mas imagine um modelo de documento mais complexo onde isso não funcionaria.)

Isso pode parecer familiar, e é: o LINQ to Objects já fornece o operador Select Many (representado por múltiplas cláusulas from em uma expressão de consulta) para fazer isso.

Você poderia escrever a consulta da seguinte maneira:

```
do projeto em root.Element("projetos").Elements() da assinatura no  
projeto.Elements("assinatura") selecione assinatura
```

Como não há elementos em um projeto além da assinatura, você pode usar a sobrecarga de Elements que não especifica um nome. Acho mais claro especificar o nome do elemento neste caso, mas muitas vezes é apenas uma questão de gosto. (O mesmo argumento poderia ser feito para chamar Element("projects").Elements("project") para começar, é certo.)

Aqui está a mesma consulta escrita usando notação de ponto e uma sobrecarga de SelectMany que retorna apenas a sequência nivelada, sem realizar quaisquer projeções adicionais:

```
root.Element("projetos").Elements()  
    .SelectMany(projeto => projeto.Elements("assinatura"))
```

Nenhuma dessas consultas é completamente ilegível, mas não são ideais. LINQ to XML fornece vários métodos de extensão (na classe System.Xml.Linq.Extensions), que atuam em um tipo de sequência específico ou são genéricos com um argumento de tipo restrito, para lidar com a falta de covariância de interface genérica antes de C# 4. Existe o InDocumentOrder, que faz exatamente o que parece

semelhantes, e a maioria dos métodos de eixo mencionados na seção 12.4.3 também estão disponíveis como métodos de extensão. Isso significa que você pode converter a consulta anterior neste formato mais simples:

```
root.Element("projetos").Elements().Elements("assinatura")
```

Esse tipo de construção facilita a gravação de consultas semelhantes a XPath em LINQ to XML sem que tudo seja uma string. Se você quiser usar XPath, isso também está disponível por meio de mais métodos de extensão, mas os métodos de consulta têm me servido bem na maioria das vezes. Você também pode misturar os métodos de eixo com os operadores do LINQ to Objects.

Por exemplo, para encontrar todas as assinaturas de projetos com um nome incluindo *Mídia*, você poderia usar esta consulta:

```
root.Element("projetos").Elements()
    .Where(projeto => ((string) projeto.Attribute("nome"))
                           .Contém("Mídia"))
    .Elements("assinatura")
```

Antes de passarmos para o Parallel LINQ, vamos pensar em como o design do LINQ to XML merece a parte "LINQ" de seu título e como você poderia aplicar as mesmas técnicas à sua própria API.

### 12.3.5 Trabalhando em harmonia com LINQ

Algumas das decisões de design no LINQ to XML parecem estranhas se você as considerar isoladamente como parte de uma API XML, mas no contexto do LINQ elas fazem todo o sentido. Os designers imaginaram claramente como seus tipos poderiam ser usados em consultas LINQ e como poderiam interagir com outras fontes de dados. Se você estiver escrevendo sua própria API de acesso a dados, em qualquer contexto, vale a pena levar as mesmas coisas em consideração. Se alguém usar seus métodos no meio de uma expressão de consulta, restará algo útil? Eles serão capazes de usar alguns dos seus métodos de consulta, depois alguns do LINQ to Objects e mais alguns dos seus em uma expressão fluente?

Vimos três maneiras pelas quais o LINQ to XML acomodou o restante do LINQ:

- ÿ É bom em *consumir* sequências com sua abordagem de construção. LINQ é deliberadamente declarativo, e LINQ to XML oferece suporte a isso com uma forma declarativa de criação de estruturas XML . ÿ Ele retorna sequências de seus métodos de consulta. Este é provavelmente o passo mais óbvio que as APIs de acesso a dados já dariam: retornar resultados de consulta como `IEnumerable<T>` ou uma classe que o implemente é praticamente óbvio. ÿ Estende o conjunto de consultas que você pode realizar em sequências de *tipos XML* ; isso faz com que pareça uma API de consulta unificada, embora algumas delas sejam específicas de XML.

Você pode pensar em outras maneiras pelas quais suas próprias bibliotecas podem funcionar bem com o LINQ; essas não são as únicas opções que você deve considerar, mas são um bom ponto de partida. Acima de tudo, recomendo que você se coloque no lugar de um desenvolvedor que deseja usar sua API dentro de um código que já usa LINQ. O que esse desenvolvedor pode querer

alcançar? O LINQ e sua API podem ser combinados facilmente ou eles estão realmente visando objetivos diferentes?

Estamos mais ou menos na metade de nosso rápido tour pelas diferentes abordagens do LINQ. Nossa próxima parada é de certa forma reconfortante e de certa forma aterrorizante: voltamos a consultar sequências simples, mas desta vez em paralelo...

#### 12.4 Substituindo LINQ por objetos com LINQ paralelo

Acompanho o Parallel LINQ há muito tempo. Eu descobri isso pela primeira vez quando Joe Duffy o apresentou em seu blog em setembro de 2006 (veja <http://mng.bz/vYCO>). O primeiro Community Technology Preview (CTP) foi lançado em novembro de 2007, e o conjunto geral de recursos também evoluiu ao longo do tempo. Agora faz parte de um esforço mais amplo chamado Extensões Paralelas, que faz parte do .NET 4, com o objetivo de fornecer blocos de construção de nível superior para programação simultânea do que o conjunto relativamente pequeno de primitivas com as quais tivemos que trabalhar até agora. Há muito mais em Extensões Paralelas do que o Parallel LINQ — ou PLINQ, como é frequentemente conhecido — mas veremos apenas o aspecto do LINQ aqui.

A ideia por trás do Parallel LINQ é que você deve ser capaz de pegar uma consulta LINQ to Objects que está demorando muito e torná-la executada mais rapidamente usando vários threads para aproveitar vários núcleos — com o mínimo possível de alterações na consulta. Como acontece com qualquer coisa relacionada à simultaneidade, não é tão simples assim, mas você pode se surpreender com o que pode ser alcançado. É claro que ainda estamos tentando pensar além das tecnologias LINQ individuais — estamos pensando nos diferentes modelos de interação envolvidos, e não nos detalhes precisos. Mas se você estiver interessado em simultaneidade, recomendo enfaticamente que mergulhe nas Extensões Paralelas — é uma das abordagens mais promissoras de paralelismo que encontrei recentemente.

Usarei um único exemplo para esta seção: renderizar uma imagem de conjunto de Mandelbrot (consulte a Wikipedia para obter uma explicação sobre conjuntos de Mandelbrot: [http://en.wikipedia.org/wiki/Mandelbrot\\_set](http://en.wikipedia.org/wiki/Mandelbrot_set)). Vamos começar tentando acertar com um único thread antes de passar para um território mais complicado.

##### 12.4.1 Plotando o conjunto de Mandelbrot com uma única rosca

Antes que qualquer matemático me ataque, devo salientar que estou usando o termo Mandelbrot definido livremente aqui. Os detalhes não são realmente importantes, mas estes aspectos são:

ÿ Você criará uma imagem retangular, com diversas opções como largura, altura, origem e profundidade de pesquisa. ÿ

Para cada pixel da imagem, você calculará um valor de byte que terminará como o índice em uma paleta de 256 entradas. ÿ O cálculo do valor de um pixel não depende de nenhum outro resultado.

O último ponto é absolutamente crucial — significa que esta tarefa é embaraçosamente paralela. Em outras palavras, não há nada na tarefa em si que dificulte a paralelização. Você ainda precisa de um mecanismo para distribuir a carga de trabalho entre threads e depois reunir os resultados, mas o resto deve ser fácil. A PLINQ será responsável pela

distribuição e coleta (com um pouco de ajuda e cuidado); você só precisa expressar o intervalo de pixels e como a cor de cada pixel deve ser calculada.

Com o propósito de demonstrar múltiplas abordagens, elaborei um classe base abstrata responsável por configurar as coisas, executar a consulta e exibir os resultados; ele também possui um método para calcular a cor de um pixel individual. Um método abstrato é responsável por criar um array de bytes de valores, que são então convertido na imagem. A primeira linha de pixels vem primeiro, da esquerda para a direita, depois a segunda linha e assim por diante. Cada exemplo aqui é apenas uma implementação deste método.

Devo observar que usar LINQ realmente não é uma solução ideal aqui — existem vários ineficiências nesta abordagem. Não se concentre nesse lado das coisas: concentre-se no ideia de que temos uma consulta embaraçosamente paralela e queremos executá-la em múltiplos núcleos.

A listagem a seguir mostra a versão single-threaded do método em toda sua glória simples.

#### Listagem 12.10 Consulta de geração de Mandelbrot de thread único

```
var query = da linha em Enumerable.Range(0, Height)
           da coluna em Enumerable.Range(0, Width)
           selecione ComputeIndex (linha, coluna);

retornar consulta.ToArray();
```

Você itera em cada linha e em cada coluna dentro de cada linha, calculando o índice do pixel relevante. Chamando `ToArray()` avalia o resultado sequência, convertendo-a em um array. A Figura 12.5 mostra os belos resultados.

Isso levou cerca de 5,5 segundos para ser gerado em meu antigo laptop dual-core. O método `ComputeIndex` tem melhor desempenho iterações do que você realmente precisa, mas torna as diferenças de tempo mais óbvias.<sup>7</sup> Agora que você tem uma referência em termos de tempo e quais os resultados deveria ser, é hora de paralelizar

A pergunta.

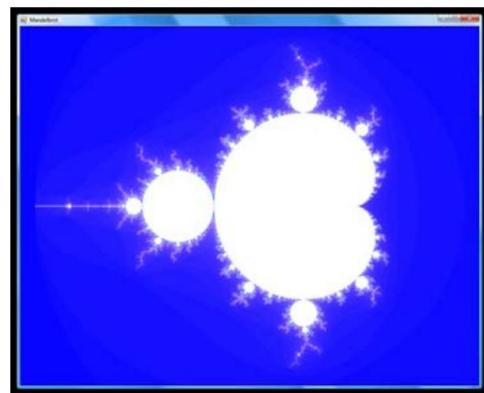


Figura 12.5 Imagem Mandelbrot gerada em um único thread

#### 12.4.2 Apresentando ParallelEnumerable, ParallelQuery e AsParallel

O Parallel LINQ traz consigo vários tipos novos, mas em muitos casos você nunca veja seus nomes mencionados. Eles vivem no namespace `System.Linq`, então você não

<sup>7</sup> O benchmarking adequado é difícil, especialmente quando há threading envolvido. Eu não tentei fazer rigorosamente medições aqui. Os tempos fornecidos destinam-se apenas a indicar mais rápido e mais lento; por favor pegue os números com uma pitada de sal.

ainda precisa mudar usando diretivas. ParallelEnumerable é uma classe estática, semelhante a Enumerable — ela contém principalmente métodos de extensão, a maioria dos quais estende um novo tipo ParallelQuery.

Este último tipo tem formas genéricas e não genéricas (ParallelQuery e ParallelQuery<TSource>), mas na maioria das vezes você usará a forma genérica, assim como I`Enumerable`<T> é mais amplamente usado que I`Enumerable`. Além disso, há Ordered ParallelQuery<TSource>, que é o equivalente paralelo de I`OrderedEnumerable`<T>.

As relações entre todos esses tipos são mostradas na figura 12.6.

Como você pode ver, ParallelQuery<TSource> implementa I`Enumerable`<TSource>, portanto, depois de construir uma consulta adequadamente, você poderá iterar pelos resultados da maneira normal. Quando você tem uma consulta paralela, os métodos de extensão em Parallel Enumerable têm precedência sobre aqueles em Enumerable (porque ParallelQuery<T> é mais específico que I`Enumerable`<T>; consulte a seção 10.2.3 se precisar de um lembrete das regras); é assim que o paralelismo é mantido ao longo de uma consulta. Há um equivalente paralelo para todos os operadores de consulta padrão do LINQ, mas você deve ter cuidado se tiver criado algum de seus próprios métodos de extensão. Você ainda poderá chamá-los, mas eles forçarão a consulta a ser de thread único a partir desse ponto.

Como você consegue uma consulta paralela para começar? Chamando AsParallel, um método de extensão em ParallelEnumerable que estende I`Enumerable`<T>. Isso significa que você pode paralelizar a consulta Mandelbrot de forma incrivelmente simples, conforme mostrado na listagem a seguir.

#### Listagem 12.11 Primeira tentativa de uma consulta de geração de Mandelbrot multithread

```
var query = da linha em Enumerable.Range(0, Height)
            .AsParallel()
            da coluna em Enumerable.Range(0, Width) selecione
            ComputeIndex(linha, coluna);

return consulta.ToArray();
```

Tarefa concluída? Bem, não exatamente. Esta consulta é executada em paralelo, mas os resultados não são exatamente o que você precisa: ela não mantém a ordem em que você processa as linhas. Em vez da bela imagem de Mandelbrot, obtemos algo como a figura 12.7, mas os detalhes exatos mudam sempre, é claro.

Ops. Pelo lado positivo, isso foi renderizado em cerca de 3,2 segundos, então minha máquina estava claramente fazendo uso de seu segundo núcleo. Mas obter a resposta certa é muito importante.

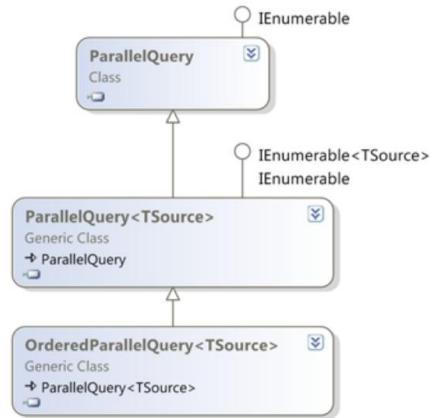


Figura 12.6 Diagrama de classes para Parallel LINQ, incluindo relacionamento com interfaces LINQ normais

Você pode se surpreender ao saber que este é um recurso deliberado do Parallel LINQ.

Ordenar uma consulta paralela requer mais coordenação entre os threads, e todo o propósito da parallelização é melhorar o desempenho, portanto o PLINQ assume como padrão uma consulta não ordenada. É um pouco incômodo neste caso, no entanto.

#### 12.4.3 Ajustando consultas paralelas

Felizmente, há uma saída para isso: você só precisa forçar a consulta a ser tratada como ordenada, o que pode ser feito com o método de extensão AsOrdered.

A listagem a seguir mostra o código fixo, que produz a imagem original.

É um pouco mais lento que a consulta não ordenada, mas ainda assim significativamente mais rápido que a versão de thread único.

#### Listagem 12.12 Consulta Mandelbrot multithread mantendo a ordem

```
var query = da linha em Enumerable.Range(0, Height)
            .AsParallel().AsOrdered() da coluna
            em Enumerable.Range(0, Width) select ComputeIndex(row,
            column);

retornar consulta.ToArray();
```

As nuances do pedido estão além do escopo deste livro, mas recomendo que você leia a postagem do blog MSDN “PLINQ Ordering” (<http://mng.bz/9x9U>), que aborda os detalhes sangrentos.

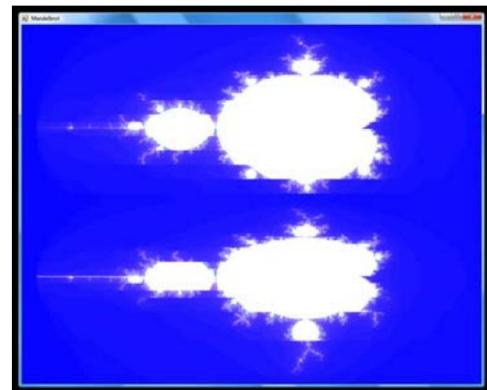


Figura 12.7 Imagem Mandelbrot gerada usando uma consulta não ordenada, resultando no posicionamento incorreto de algumas seções

Vários outros métodos podem ser usados para alterar o comportamento da consulta:

- ÿ AsUnordered — Torna uma consulta ordenada desordenada; se você precisar apenas que os resultados sejam ordenados para a primeira parte de uma consulta, isso permitirá que os estágios posteriores sejam executados com mais eficiência.

- ÿ WithCancellation—Especifica um token de cancelamento a ser usado com esta consulta.

- Os tokens de cancelamento são usados em extensões paralelas para permitir que tarefas sejam canceladas de maneira segura e controlada.

- ÿ WithDegreeOfParallelism — Permite especificar o número máximo de tarefas simultâneas usadas para executar a consulta. Você poderia usar isso para limitar o número de threads usados se quisesse evitar sobrecarregar a máquina ou para aumentar o número de threads usados para uma consulta que não estivesse vinculada à CPU.

- ÿ WithExecutionMode — Pode ser usado para forçar a execução da consulta em paralelo, mesmo se o Parallel LINQ achar que seria executado mais rapidamente como uma consulta de thread único.

ÿ WithMergeOptions—Permite ajustar como os resultados são armazenados em buffer.

Desabilitar o buffer proporciona o menor tempo antes que o primeiro resultado seja retornado, mas também reduz o rendimento; o buffer completo fornece o rendimento mais alto, mas nenhum resultado é retornado até que a consulta seja executada completamente. O padrão é um compromisso entre os dois.

O ponto importante é que, além da ordenação, esses métodos não devem afetar os resultados da consulta. Você pode projetar sua consulta e testá-la no LINQ to Objects, depois parelizá-la, definir seus requisitos de pedido e ajustá-la, se necessário, para funcionar exatamente como você deseja. Se você mostrasse a consulta final para alguém que conhecesse LINQ, mas não PLINQ, bastaria explicar as chamadas de método específicas do PLINQ — o restante da consulta seria familiar. Você já viu uma maneira tão fácil de obter simultaneidade?

(O restante das extensões paralelas também visa alcançar a simplicidade sempre que possível.)

**BRINQUE COM O CÓDIGO VOCÊ MESMO** Alguns pontos adicionais são demonstrados no código-fonte disponível para download. Se você paralelizar toda a consulta de pixels em vez de apenas as linhas, uma consulta não ordenada parecerá ainda mais estranha e haverá um método ParallelEnumerable.Range que fornece ao PLINQ um pouco mais de informações do que chamar Enumerable.Range(...).AsParallel(). Usei AsParallel() nesta seção, porque essa é a maneira mais geral de paralelizar uma consulta; a maioria das consultas não começa com um intervalo.

Mudar o modelo de consulta em processo de thread único para paralelo é, na verdade, um pequeno salto conceitual. Na próxima seção, viraremos o modelo de cabeça para baixo.

## 12.5 Invertendo o modelo de consulta com LINQ to Rx

Todas as bibliotecas LINQ que você viu até agora têm uma coisa em comum: você extraí dados delas usando I Enumerable<T>. À primeira vista, isso parece tão óbvio que nem vale a pena dizer: qual seria a alternativa? Bem, que tal se você enviar os dados por push em vez de extraí-los? Em vez de o consumidor de dados estar no controle, o provedor pode estar no comando, deixando o consumidor de dados reagir quando novos dados estiverem disponíveis.

Não se preocupe muito se isso parecer assustadoramente diferente; na verdade, você já conhece o conceito fundamental, na forma de eventos. Se você se sente confortável com a ideia de se inscrever em um evento, reagir a ele e cancelar a inscrição mais tarde, esse é um bom ponto de partida.

Extensões reativas para .NET é um projeto da Microsoft (<http://mng.bz/R7ip>); existem várias versões disponíveis, incluindo uma voltada para JavaScript. Hoje em dia, a maneira mais simples de obter a versão mais recente é por meio do NuGet. Você pode ouvir extensões reativas com vários nomes, mas Rx e LINQ to Rx são as abreviaturas mais comuns e são as que usarei aqui. Ainda mais do que para as outras tecnologias abordadas neste capítulo, mal arranharemos a superfície aqui. Não só há muito o que aprender sobre a biblioteca em si, mas também há uma maneira totalmente diferente de pensar. Existem muitos vídeos no Canal 9 (veja <http://channel9.msdn.com/tags/Rx/>) – alguns são baseados em aspectos matemáticos, enquanto outros são mais práticos. Nesta seção enfatizarei a maneira como os conceitos do LINQ podem ser aplicados a esse modelo push para fluxo de dados.

Chega de introdução... vamos conhecer as duas interfaces que formam a base do LINQ to Rx.

### 12.5.1 IObservable<T> e IObserver<T>

O modelo de dados do LINQ to Rx é o *dual matemático* do modelo `IEnumerable<T>` normal.<sup>8</sup> Ao iterar em uma coleção pull, você efetivamente começa dizendo: “Por favor, me dê um iterador” (a chamada para `GetEnumerator`) e em seguida, pergunte repetidamente: “Existe outro item? Se sim, eu gostaria agora” (através de chamadas para `MoveNext` e `Current`). LINQ to Rx inverte isso. Em vez de solicitar um iterador, você fornece um observador. Então, em vez de solicitar o próximo item, seu código será informado quando um estiver pronto – ou quando ocorrer um erro ou quando o final dos dados for atingido.

Aqui estão as declarações das duas interfaces envolvidas:

```
interface pública IObservable<T> {

    IDisposable Subscribe(IObserver<T> observador);
}

interface pública IObserver<T> {

    void OnNext (valor T); void
    OnCompleted(); void
    OnException (erro de exceção);
}
```

Na verdade, essas interfaces fazem parte do .NET 4 (no namespace `System`), embora o restante do LINQ to Rx esteja em um download separado. Na verdade, eles são `IObservable<out T>` e `IObserver<in T>` no .NET 4, expressando a covariância de `IObservable` e a contravariância de `IObserver`. Você aprenderá mais sobre variância genérica no próximo capítulo, mas estou apresentando as interfaces aqui como se fossem invariantes por uma questão de simplicidade. Um conceito de cada vez!

A Figura 12.8 mostra a dualidade em termos de como os dados fluem em cada modelo.

Suspeito que não sou o único a achar o modelo push mais difícil de pensar, pois ele tem a capacidade natural de funcionar de forma assíncrona. Mas veja como ele é muito mais simples que o modelo pull, em termos de diagrama de fluxo. Isto se deve em parte à abordagem de métodos múltiplos do modelo pull; se `IEnumerable<T>` tivesse apenas um método com uma assinatura `bool TryGetNext(out T item)`, seria um pouco mais simples.

Mencionei anteriormente que LINQ to Rx é semelhante aos eventos com os quais você já está familiarizado. Chamar `Subscribe` em um observável é como usar `+=` com um evento para registrar um manipulador. O valor descartável retornado por `Subscribe` lembra o observador que você passou; descartá-lo é como usar `=` com o mesmo manipulador. Em muitos casos, você não precisa cancelar a assinatura do observável; isso está disponível apenas no caso de você precisar cancelar a assinatura no meio de uma sequência - o equivalente a interromper um loop `foreach` antecipadamente. Deixar de descartar um valor `IDisposable` pode parecer um anátema

---

<sup>8</sup> Para um exame mais detalhado dessa dualidade — e da própria essência do LINQ — recomendo a postagem do blog “The Essence of LINQ — MINLINQ” de Bart de Smet em <http://mng.bz/96Wh>.

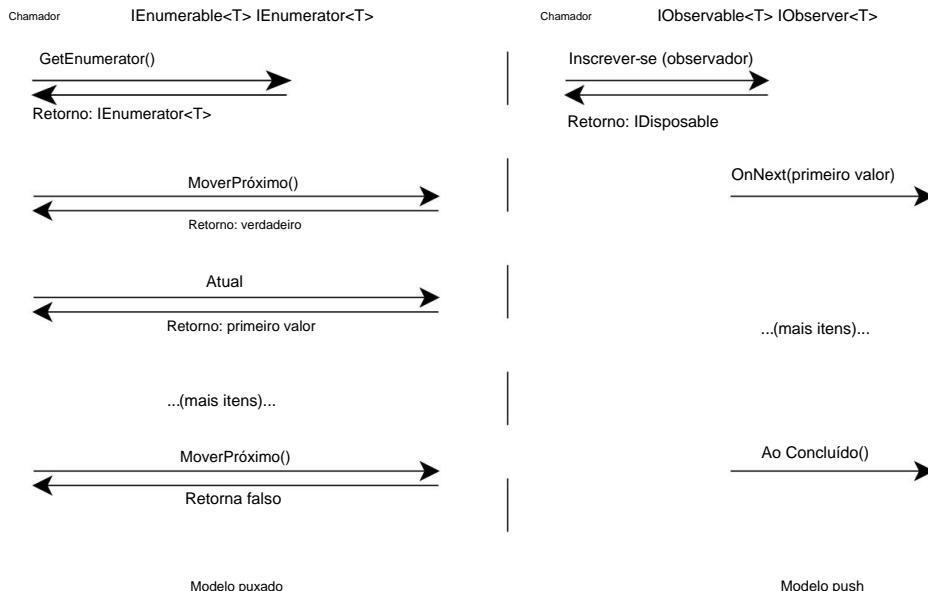


Figura 12.8 Diagrama de sequência mostrando a dualidade de `IEnumerable<T>` e `IObservable<T>`

para você, mas geralmente é seguro no LINQ to Rx. Nenhum dos exemplos neste capítulo usa o valor de retorno de `Subscribe`.

`IObservable<T>` é tudo o que existe, mas o próprio observador? Por que tem três métodos? Considere o modelo pull normal onde, para qualquer par de chamadas `MoveNext/Current`, três coisas podem acontecer: ✓ Você pode estar no final da sequência e, nesse caso, `MoveNext` retorna falso. ✓ Você pode não ter chegado ao final da

sequência; nesse caso, `MoveNext`

retorna verdadeiro e `Current` retorna o novo valor.

✓ Pode ocorrer um erro – a leitura da próxima linha de uma conexão de rede pode falhar, por exemplo. Nesse caso, uma exceção seria lançada.

A interface `IObserver<T>` representa cada uma dessas opções como um método separado.

Normalmente, um observador terá seu método `OnNext` chamado repetidamente e, finalmente, `OnCompleted`, a menos que haja algum tipo de erro; nesse caso, `OnError` será chamado. Depois que a sequência for concluída ou encontrar um erro, nenhuma outra chamada de método será feita. Porém, raramente você precisa implementar `IObserver<T>` diretamente. Existem muitos métodos de extensão em `IObservable<T>`, incluindo sobrecargas para `Subscribe`, e eles permitem que você assine um observável apenas fornecendo delegados apropriados. Normalmente você fornece um delegado para ser executado para cada item e, opcionalmente, um para ser executado na conclusão, em caso de erro ou ambos.

Com essa parte da teoria fora do caminho, vamos dar uma olhada em alguns códigos reais usando LINQ to Rx.

### 12.5.2 Começando de forma simples (de novo)

Demonstraremos o LINQ to Rx da mesma forma que começamos com o LINQ to Objects— usando um intervalo. Em vez de Enumerable.Range, usaremos Observable.Range, que cria um intervalo observável. Cada vez que um observador assina o intervalo, os números são emitidos para esse observador usando OnNext, seguido por OnCompleted. Começaremos como da maneira mais simples possível, apenas imprimindo cada valor conforme ele é recebido e imprimindo uma mensagem de confirmação no final ou se ocorrer um erro.

A listagem a seguir mostra que isso envolve menos código do que você precisa para o pull modelo.

#### Listagem 12.13 Primeiro contato com IObserver<T>

```
var observável = Observable.Range(0, 10);
observável.Subscribe(x => Console.WriteLine("Recebido {0}", x),
                     e => Console.WriteLine("Erro: {0}", e),
                     () => Console.WriteLine("Concluído"));
```

Nesse caso, é difícil ver como você pode obter um erro, mas inclui o erro delegado de notificação para integridade. Os resultados são os esperados:

```
Recebido 0
Recebido 1
...
Recebido 9
Finalizado
```

O observável retornado pelo método Range é conhecido como observável frio: ele permanece inativo até que um observador o assine, momento em que emitirá os valores para esse observador individual. Se você se inscrever com outro observador, verá outra cópia do intervalo. Isso não é exatamente o mesmo que um evento normal, como um clique de botão, onde vários observadores poderiam estar inscritos na mesma sequência real de valores, e os valores podem ser efetivamente obtidos, quer haja observadores ou não. (Você pode clicar em um botão mesmo que não haja nenhum manipulador de eventos anexado, afinal.) Sequências como este são conhecidos como observáveis quentes. É importante saber com que tipo você está lidando com, mesmo que o mesmo conjunto de operações se aplique a ambos os tipos.

Agora que você fez a coisa mais simples possível, vamos tentar alguns comandos LINQ familiares operadores.

### 12.5.3 Consultando observáveis

Até agora tenho certeza de que você está familiarizado com o padrão – existem vários métodos de extensão em uma classe estática (chamada Observable, de forma um tanto previsível) que executam transformações apropriadas. Veremos apenas alguns dos operadores disponíveis e pensaremos um pouco sobre o que não está disponível e por que não está.

#### FILTRAGEM E PROJEÇÃO

Vamos direto para uma expressão de consulta que usa uma sequência de números, filtros elimina os ímpares e enquadra tudo o que resta. Então assinaremos o Console

.WriteLine ao resultado final da consulta, para que quaisquer itens produzidos sejam exibidos. A listagem a seguir mostra o código – veja como a expressão de consulta poderia facilmente ser uma consulta LINQ to Objects.

#### Listagem 12.14 Filtrando e projetando em LINQ to Rx

```
var números = Observable.Range(0, 10); var consulta = do
número em números onde número% 2 == 0

selecionar número * número;
query.Subscribe(Console.WriteLine);
```

Para simplificar, não adicionei manipuladores para conclusão ou erro, e usar a conversão do grupo de métodos Console.WriteLine para Action<int> mantém o código agradável e curto. Isso produz os mesmos resultados que produziria em LINQ to Objects: 0, 4, 16 e assim por diante. Vamos passar para o agrupamento.

#### AGRUPAMENTO

Uma expressão de consulta group by em LINQ to Rx produz um novo IGroupedObservable<T> para cada grupo, embora o que você faz com o agrupamento nem sempre seja óbvio. Por exemplo, não é incomum ter uma assinatura aninhada para que cada vez que um novo grupo é produzido, você inscreva um observador nesse grupo. Os resultados *dentro de* cada grupo são produzidos à medida que são recebidos pela construção de agrupamento – na verdade, ele atua como uma espécie de escolha de redirecionamento, como um apresentador de uma peça examinando o ingresso de cada pessoa quando ela chega e direcionando-os para a seção relevante do grupo. Por outro lado, LINQ to Objects reúne um grupo inteiro antes de retorná-lo, o que significa que ele precisa ler até o final da sequência, armazenando em buffer todos os resultados.

A listagem a seguir mostra um exemplo dessa assinatura aninhada e também demonstra como os resultados do grupo são emitidos.

#### Listagem 12.15 Agrupamento de números mod 3

```
var números = Observable.Range(0, 10); var query = de
número em números agrupar número por número % 3;

query.Subscribe(group => group.Subscribe (x =>
    Console.WriteLine("Valor: {0}; Grupo: {1}", x, group.Key)));
```

A melhor maneira de entender isso é provavelmente lembrar que lidar com grupos no LINQ to Objects geralmente envolve ter um loop foreach aninhado – portanto, você tem assinaturas aninhadas no LINQ to Rx.

Na dúvida, tente encontrar a dualidade entre os dois modelos de dados. No LINQ to Objects, você normalmente processaria cada grupo inteiro por vez, enquanto a ordem no LINQ to Rx significa que a saída da listagem 12.15 se parece com isto:

```
Valor: 0; Grupo: 0
Valor: 1; Grupo 1
Valor: 2; Grupo: 2
Valor: 3; Grupo: 0
```

```
Valor: 4; Grupo: 1
Valor: 5; Grupo: 2
Valor: 6; Grupo: 0
Valor: 7; Grupo: 1
Valor: 8; Grupo: 2
Valor: 9; Grupo: 0
```

Isso faz todo o sentido quando você pensa no modelo push e, em alguns casos, significa que operações que exigiriam muito buffer de dados no LINQ to Objects podem ser implementadas no LINQ to Rx com muito mais eficiência.

Como exemplo final, vejamos outro operador que usa múltiplas sequências.

#### ACHATAMENTO

LINQ to Rx fornece algumas sobrecargas de SelectMany, e a ideia ainda é a mesma do LINQ to Objects: cada item na sequência original produz uma nova sequência, e o resultado é a combinação de todas essas novas sequências, achatadas. A listagem a seguir mostra isso em ação — é um pouco como a listagem 11.16, quando discutimos pela primeira vez Select-Many no LINQ to Objects.

#### Listagem 12.16 SelectMany produzindo múltiplos intervalos

```
var query = de x em Observable.Range(1, 3) de y em
    Observable.Range(1, x) selecione novo { x, y };
    query.Subscribe(Console.WriteLine);
```

Aqui estão os resultados, que devem ser razoavelmente previsíveis:

```
{ x = 1, y = 1 } { x = 2, y =
= 1 } { x = 2, y = 2 } { x =
3, y = 1 } { x = 3, y = 2 }
{ x = 3, y = 3}
```

Nesse caso, os resultados são determinísticos, mas isso ocorre apenas porque, por padrão, Observable.Range emite itens no thread atual. É perfeitamente possível produzir múltiplas sequências em vários threads.

Por diversão, você pode querer alterar a segunda chamada para Observable.Range para especificar Scheduler.ThreadPool como um terceiro argumento. Nesse caso, cada uma das sequências internas aparece em ordem em relação a si mesma, mas essas sequências separadas podem ser misturadas entre si. Imagine um estádio esportivo com um oficial disparando uma pistola de largada para várias corridas diferentes em rápida sucessão — mesmo que você conheça o vencedor de cada corrida, você não sabe qual corrida terminará em primeiro lugar.

Peço desculpas se isso faz você querer ir se deitar. Se serve de consolo, dá eu o mesmo sentimento. Eu acho isso fascinante ao mesmo tempo.

#### O QUE ESTÁ E O QUE ESTÁ FORA?

Você já sabe que uma cláusula let funciona apenas chamando Select, de modo que funciona naturalmente no LINQ to Rx, mas nem todos os operadores LINQ to Objects são implementados no LINQ to Rx.

Os operadores ausentes são geralmente aqueles que teriam que armazenar em buffer toda a sua saída e retornar um novo observável. Por exemplo, não há método Reverse e nem OrderBy. O C# está bastante satisfeito com isso — ele simplesmente não permite usar uma cláusula orderby em uma expressão de consulta baseada em observáveis. Existe um método Join , mas ele não lida diretamente com observáveis — ele lida com *planos de junção*. Isso faz parte da implementação Rx do cálculo de junção e está muito além do escopo deste livro. Da mesma forma, não há método GroupJoin , então join...into não é suportado.

Para os vários operadores de consulta padrão LINQ que não são cobertos pela sintaxe da expressão de consulta — e para ver a variedade de métodos extras que ela disponibiliza — consulte a documentação System.Reactive . Embora você possa começar a ficar desapontado com a funcionalidade familiar do LINQ to Objects que está faltando no LINQ to Rx (geralmente porque simplesmente não faz sentido), você pode se surpreender com o quanto rico é realmente o conjunto de métodos disponíveis. Muitos dos novos métodos são então portados para LINQ to Objects no assembly System.Interactive .

#### 12.5.4 Qual é o objetivo?

Estou ciente de que ainda não forneci nenhum motivo convincente para usar o LINQ to Rx. Isto é deliberado, pois não pretendo mostrar um exemplo completo e útil — é incidental ao ponto deste capítulo e ocuparia muito espaço. Mas o Rx fornece uma maneira elegante de pensar sobre todos os tipos de processos assíncronos, como eventos normais do .NET (que podem ser vistos como observáveis usando Observable.FromEvent), E/S assíncrona e chamadas para serviços da Web. Ele fornece uma maneira de gerenciar a complexidade e a simultaneidade de maneira eficiente. Não há dúvida de que é mais difícil de entender do que LINQ to Objects, mas se você estiver no tipo de situação em que seria útil, já estará enfrentando uma montanha de complexidade.

A razão pela qual quis abordar Rx neste livro, apesar de não ser capaz de fazer qualquer tipo de justiça, é porque mostra por que o LINQ foi projetado da maneira que foi. Embora existam métodos de conversão disponíveis entre IEnumerable<T> e IObservable<T>, não há relacionamento de herança. Se a linguagem tivesse exigido que os tipos envolvidos no LINQ fossem sequências pull, não haveria suporte a expressões de consulta para Rx. Teria sido ainda mais desastroso se os métodos de extensão tivessem sido limitados a IEnumerable<T> de alguma forma. Da mesma forma, você viu que nem todos os operadores LINQ normais são aplicáveis ao Rx, e é por isso que é importante que a linguagem especifique as traduções de consulta em termos de um padrão que deve ser suportado na medida em que faça sentido para o provedor específico. . Espero que você tenha a sensação de que, embora os modelos push e pull sejam muito diferentes para trabalhar, o LINQ atua como uma espécie de força unificadora sempre que possível.

Você pode ficar aliviado ao saber que nosso último tópico é muito mais simples – está de volta à casa base do LINQ to Objects, mas desta vez estamos escrevendo nossos próprios métodos de extensão.

## 12.6 Estendendo LINQ para objetos

Uma das coisas boas do LINQ é que ele é extensível. Você não só pode subir com seus próprios provedores de consulta e modelos de dados, você também pode adicionar aos existentes. Em minha experiência, a situação mais comum em que isso é útil é com LINQ to Objetos. Se você precisar de um tipo específico de consulta que não tenha suporte direto (ou que seja estranho ou ineficiente com os operadores de consulta padrão), você poderá escrever a sua própria. De É claro que escrever um método genérico de uso geral pode ser mais desafiador do que apenas resolvendo seu problema imediato, mas se você estiver escrevendo um código semelhante algumas vezes, vale a pena considerar se você poderia refatorá-lo em um novo operador.

Pessoalmente, gosto de escrever operadores de consulta. Existem desafios técnicos interessantes, mas raramente requerem uma grande quantidade de código e os resultados podem ser elegantes. Em esta seção, veremos algumas maneiras de tornar seus operadores personalizados comportar-se de forma eficiente e previsível, seguido por uma amostra completa para selecionar um elemento de uma sequência.

### 12.6.1 Diretrizes de design e implementação

A maioria dessas diretrizes pode parecer bastante óbvia, mas esta seção pode constituir uma ferramenta útil lista de verificação ao escrever um operador.

#### TESTES DE UNIDADE

Geralmente é muito fácil escrever um bom conjunto de testes unitários para operadores, embora você pode se surpreender com quantos você obtém para o que originalmente parecia ser um código simples. Não se esqueça de testar casos extremos, como sequências vazias, bem como inválidas argumentos. MoreLINQ (<http://code.google.com/p/morelinq/>) tem algum ajudante métodos em seu projeto de teste de unidade que você pode querer usar em seus próprios testes.

#### VERIFICAÇÃO DE ARGUMENTOS

Bons métodos verificam seus argumentos, mas há um problema quando se trata de LINQ operadores. Muitos operadores retornam outra sequência, como você já viu, e os blocos iteradores são a maneira mais fácil de implementar essa funcionalidade. Mas você realmente deveria execute a verificação de argumentos assim que seu método for chamado, em vez de esperar até que o chamador decida iterar os resultados. Se você for usar um iterador bloco, divida seu método em dois: execute a verificação de argumentos em um método público e então chame um método privado para fazer a iteração.

#### OTIMIZAÇÃO

O próprio `IEnumerable<T>` é bastante fraco em termos das operações que suporta, mas o tipo de tempo de execução de uma sequência em que você está trabalhando pode ter consideravelmente mais funcionalidade. Por exemplo, o operador `Count()` sempre funcionará, mas geralmente seja uma operação  $O(n)$ . Se você chamar isso em uma implementação de `ICollection<T>`, porém, ele pode usar a propriedade `Count` diretamente, que geralmente será  $O(1)$ . Em .NET 4, essa otimização é estendida para abranger também o `ICollection`. Da mesma forma, recuperar um elemento específico por índice é lento no caso geral, mas pode ser eficiente se o sequência implementa `IList<T>`.

Se o seu operador puder se beneficiar dessas otimizações, você poderá ter diferentes caminhos de execução dependendo do tipo de tempo de execução. Para testar o caminho lento em testes unitários, você sempre pode chamar `Select(x => x)` em um `List<T>` para recuperar uma sequência não listada. `LinkedList<T>` pode testar o caso em que você deseja um `ICollection<T>` que não implemente `IList<T>`.

#### DOCUMENTAÇÃO

É importante documentar o que seu código fará com suas entradas e também o desempenho esperado do operador. Isto é particularmente importante se o seu método precisa trabalhar com múltiplas sequências: qual delas será avaliada primeiro e até que ponto? Seu código transmite seus dados, armazena-os em buffer ou é uma mistura? Ele usa diferido ou execução imediata? Qualquer parâmetro pode ser nulo e, em caso afirmativo, isso tem um valor especial significado?

#### ITERAR UMA VEZ QUE POSSÍVEL

No nível da interface, `IEnumerable<T>` permitirá iterar na mesma sequência várias vezes - você pode ter vários iteradores ativos ao mesmo tempo no mesmo sequência, potencialmente. Mas isso raramente é uma boa ideia dentro de uma operadora. Sempre que possível, é aconselhável iterar suas sequências de entrada apenas uma vez. Isso significa que seu código será funcionam mesmo para sequências não repetíveis, como linhas lidas de um fluxo de rede. Se você precisa ler a sequência várias vezes (e não deseja armazenar em buffer a sequência inteira, como `Reverse` faz), você deve chamar atenção especial para isso em a documentação.

#### LEMBRE-SE DE DESCARTAR OS ITERADORES

Na maioria dos casos, você pode usar uma instrução `foreach` para iterar sua fonte de dados. Mas às vezes é útil tratar o primeiro item de maneira diferente e, nesse caso, usar um iterador diretamente pode levar ao código mais simples. Nessa situação, lembre-se de incluir um uso bloco para o iterador. Você provavelmente não está acostumado a descartar iteradores sozinho porque normalmente o `foreach` faz isso por você, o que pode dificultar a detecção do bug.

#### APOIE COMPARAÇÕES PERSONALIZADAS

Muitos operadores LINQ possuem sobrecargas que permitem especificar um valor apropriado `IEqualityComparer<T>` ou `IComparer<T>`. Se você estiver construindo uma biblioteca de uso geral para outros (potencialmente desenvolvedores com os quais você não tem contato), pode valer a pena fornecer você mesmo sobrecargas semelhantes. Por outro lado, se você for o único usuário ou apenas serão membros de sua equipe que o usarão, você pode fazer isso conforme a necessidade de implementação base. Porém, é fácil: normalmente as sobrecargas mais simples chamam apenas uma mais complexa, passando `EqualityComparer<T>.Default` ou `Comparer<T>.Default` como comparação.

Agora que já falei o que falar, vamos verificar se consigo realmente fazer o mesmo.

### 12.6.2 Extensão da amostra: selecionando um elemento aleatório

O objetivo do método de extensão que veremos aqui é simples: dada uma sequência e uma instância de `Random`, retorna um elemento aleatório da sequência. Você poderia adicionar uma sobrecarga que não requer a instância de `Random`, mas prefiro fazer o

dependência de um gerador de números aleatórios explícita. A aleatoriedade é um tema complicado por vários motivos e, em vez de discuti-lo aqui, inclui um artigo no site do livro (veja <http://mng.bz/h483>). Também por questões de espaço, não inclui a documentação XML ou testes de unidade na listagem a seguir, mas é claro que eles estão no código para download.

#### Listagem 12.17 Método de extensão para escolher um elemento aleatório de uma sequência

```
public static T RandomElement<T>(esta fonte I Enumerable<T>,
                                  Aleatório aleatório)
{
    if (fonte == nulo) {B Valida argumentos
        lançar novo ArgumentNullException("fonte");
    }
    if (aleatório == nulo) {
        lançar novo ArgumentNullException("aleatório");
    }
    Coleção ICollection = fonte como ICollection; if (coleção! = nulo) {C Otimiza para
Coleções C
        int contagem = coleção.Contagem; if (contagem
        == 0) {
            throw new InvalidOperationException("A sequência estava vazia.");
        }
        int índice = random.Next(contagem); retornar
        fonte.ElementAt(índice);
    }
    } usando (IEnumerator<T> iterador = source.GetEnumerator()) {D ElementAt
otimiza ainda mais
        if (!iterador.MoveNext()) {
            throw new InvalidOperationException("A sequência estava vazia.");
        }
        int contagemSoFar = 1; T atual
        = iterador.Current; enquanto (iterador.MoveNext())
        {
            contarSoFar++; if
            (random.Next(contaSoFar) == 0) {E Alças
Caso D lento
                atual = iterador.Atual;
            }
        }
        corrente de retorno;
    }
}
```

A Listagem 12.17 não mostra a técnica de divisão de um método de extensão em validação de argumento e depois implementação, porque ela não usa um bloco iterador.

Reveja a implementação do operador Where na seção 10.3.3 para obter uma

exemplo disso. Também não são necessárias comparações personalizadas, mas, fora isso, cada item da lista de verificação é apropriado.

Primeiro você valida seus argumentos da maneira óbvia B. As coisas ficam mais interessantes onde a sequência de origem implementa `ICollection<T>`.<sup>9</sup> Isso permite que você pegue contar de forma barata e depois gerar um único número aleatório para decidir qual elemento escolher. Você não trata *explicitamente* o caso em que a sequência de origem implementa `IList<T>`; em vez disso, você confia no `ElementAt` para fazer isso por você (como está documentado na documentação).

Se você estiver lidando com uma sequência que não é de coleção (como o resultado de outra operador de consulta), você deseja evitar fazer a contagem e depois escolher um elemento; isso exigiria que você armazenasse em buffer o conteúdo da sequência ou iterasse sobre ela duas vezes. Em vez disso, você percorre uma vez, buscando explicitamente o iterador D para que você pode testar facilmente uma sequência vazia. O bit<sup>10</sup> inteligente está em E – você substitui sua ideia atual de um elemento aleatório pelo elemento do iterador com uma probabilidade de  $1/n$ , onde  $n$  é o número de elementos que você viu até agora. Há meia chance de substituir o primeiro elemento pelo segundo, um terço de chance de substituir o resultado após dois elementos com o terceiro elemento e assim por diante. O resultado final é que cada elemento na sequência tem chances iguais de ser escolhido e você conseguiu iterar apenas uma vez.

É claro que o ponto importante não é o que este método específico faz – é o problemas potenciais que tiveram que ser considerados durante a implementação. Depois de saber o que procurar, realmente não é preciso muito esforço para implementar um método robusto como este, e sua caixa de ferramentas pessoal crescerá com o tempo.

## 12.7 Resumo

Ufa! Este capítulo foi exatamente o oposto da maior parte do restante do livro.

Em vez de focar em um único tópico detalhadamente, cobrimos uma série de LINQ tecnologias, mas a um nível superficial.

Eu não esperaria que você se sentisse particularmente familiarizado com qualquer uma das tecnologias específicas que examinamos aqui, mas espero que você tenha uma compreensão mais profunda do porquê LINQ é importante. Não se trata de XML ou consultas na memória, consultas SQL, observáveis ou enumeradores - trata-se de consistência de expressão e de dar ao compilador C# a oportunidade de validar suas consultas pelo menos até certo ponto, independentemente de sua plataforma de execução final.

Agora você deve entender por que as árvores de expressão são tão importantes que são entre os poucos elementos *da estrutura* que o compilador C# tem conhecimento íntimo direto de (junto com strings, `IDisposable`, `IEnumerable<T>` e `Nullable<T>`, por exemplo).

Eles atuam como passaportes, permitindo que o comportamento atravesses a fronteira da máquina local, expressar lógica em qualquer língua estrangeira atendida por um provedor LINQ .

<sup>9</sup> O código para download contém o mesmo teste para implementações de `ICollection<T>`, assim como `Count()` faz no .NET 4. É exatamente o mesmo bloco de código, apenas com um tipo diferente e um nome de variável diferente.

<sup>10</sup> Sinto-me confortável em afirmar que isso é inteligente porque, embora seja minha implementação, não é ideia minha.

Não são apenas árvores de expressão – também contamos com a tradução da expressão de consulta empregado pelo compilador e a maneira como as expressões lambda podem ser convertidas em delegados e árvores de expressão. Mas os métodos de extensão também são importantes, pois sem eles, cada fornecedor teria que implementar todas as métodos. Se você observar todos os novos recursos do C#, encontrará alguns que não contribuem significativamente para o LINQ de uma forma ou de outra. Essa é parte da razão para isso existência do capítulo: para mostrar as conexões entre todos os recursos.

Eu não deveria ser lírico por muito tempo, no entanto. Além das vantagens do LINQ, nós vi algumas pegadinhas. O LINQ nem sempre permitirá que você expresse tudo o que precisa em um consulta, nem oculta *todos* os detalhes da fonte de dados subjacente. Quando se trata de provedores de banco de dados LINQ , as incompatibilidades de impedância que fizeram com que os desenvolvedores muitos problemas do passado ainda estão conosco: você pode reduzir seu impacto com sistemas ORM e similares, mas sem uma compreensão adequada da consulta que está sendo executada em seu nome, é provável que você enfrente problemas significativos. Em particular, não pense em LINQ como uma forma de eliminar sua necessidade de entender SQL – pense nisso como uma forma de esconder o SQL quando você não está interessado nos detalhes. Da mesma forma, para planejar uma consulta paralela eficaz, você precisa saber onde a ordem é importante e onde não. e talvez ajudar um pouco a estrutura, fornecendo mais informações de ajuste.

Desde que o .NET 3.5 foi lançado, fiquei encantado em ver como a comunidade abraçou o LINQ de todo o coração. Da mesma forma, tem havido muitos usos interessantes de os recursos do C# 4, que você verá na próxima parte do livro.

## Parte 4

# *C# 4: Jogando bem com os outros*

C# 4 é uma fera engraçada. Não tem os “vários grandes, quase não relacionados, novos recursos” do C# 2, nem a sensação de “tudo na causa do LINQ” do C# 3. Em vez disso, os novos recursos do C# 4 ficam em algum lugar entre os dois. A interoperabilidade é um tema importante, mas muitos dos recursos são igualmente úteis, mesmo que você nunca precise trabalhar com outros ambientes.

Meus recursos favoritos do C# 4 são parâmetros opcionais e argumentos nomeados. Eles são relativamente simples, mas podem ser bem utilizados em muitos lugares, melhorando a legibilidade do código e geralmente tornando a vida mais agradável. Você perde tempo descobrindo qual argumento significa o quê? Coloque alguns nomes neles. Você está cansado de escrever sobrecargas intermináveis para evitar que os chamadores tenham que especificar tudo? Torne alguns parâmetros opcionais.

Se você trabalha com COM, C# 4 será uma lufada de ar fresco para você. Para começar, os recursos que acabei de descrever tornam o trabalho com algumas APIs muito mais simples, onde os designers de componentes presumiram que você trabalhará com uma linguagem que suporta parâmetros opcionais e argumentos nomeados. Além disso, há uma história de implantação melhor, suporte para indexadores nomeados e um atalho útil para evitar a necessidade de passar argumentos por referência em todos os lugares. O maior recurso do C# 4 – digitação dinâmica – também facilita a integração COM .

Veremos todas essas áreas no capítulo 13, juntamente com o tópico exaustivo da variação genérica aplicada a interfaces e delegados. Não se preocupe; faremos isso razoavelmente devagar, e a melhor parte é que na maioria das vezes você não

precisa saber os detalhes - isso apenas faz o código funcionar onde você esperava no C# 3 de qualquer maneira!

O Capítulo 14 aborda a digitação dinâmica e o Dynamic Language Runtime (DLR). Este é um tópico enorme. Concentrei-me em como a linguagem C# implementa tipagem dinâmica, mas também veremos alguns exemplos de interoperação com linguagens dinâmicas, como IronPython, e veremos exemplos de como um tipo pode responder dinamicamente a chamadas de método, acessos de propriedade e breve. Vale a pena aplicar um pouco de perspectiva aqui: o fato de esse ser um recurso importante não significa que você deva esperar ver expressões dinâmicas surgindo em toda a sua base de código. Isso não será tão difundido quanto o LINQ, por exemplo, mas quando você quiser digitação dinâmica, você a encontrará bem implementada em C# 4.

# 13

## Pequenas alterações para simplificar o código

### Este capítulo cobre

- ÿ Parâmetros opcionais
- ÿ Argumentos nomeados
- ÿ Simplificando parâmetros de referência em COM
- ÿ Incorporação de assemblies de interoperabilidade primária COM
- ÿ Chamando indexadores nomeados declarados em COM
- ÿ Variação genérica para interfaces e delegados
- ÿ Mudanças em eventos de bloqueio e semelhantes a campos

Assim como nas versões anteriores, o C# 4 possui alguns recursos menores que não merecem capítulos individuais. Na verdade, há apenas um recurso realmente *importante* no C# 4 — tipagem dinâmica — que abordaremos no próximo capítulo. As mudanças que abordaremos aqui tornam o C# um pouco mais agradável de trabalhar, especialmente se você trabalha com COM regularmente. Esses recursos geralmente tornam o código mais claro, eliminam o trabalho enfadonho das chamadas COM ou simplificam a implantação.

Algum desses recursos fará seu coração disparar de excitação? É improvável. Mas, mesmo assim, são recursos interessantes e alguns deles podem ser amplamente aplicáveis. Vamos começar observando como chamamos métodos.

## 13.1 Parâmetros opcionais e argumentos nomeados

Esses talvez sejam os recursos do Batman e do Robin<sup>1</sup> do C# 4. Eles são distintos, mas geralmente vistos juntos. Vou mantê-los separados por enquanto para que possamos examinar cada um por vez, mas depois os usaremos juntos para alguns exemplos mais interessantes.

### Parâmetros e argumentos Esta

seção obviamente fala muito sobre parâmetros e argumentos. Em conversas casuais, os dois termos são frequentemente usados de forma intercambiável, mas vou usá-los de acordo com suas definições formais. Só para lembrar, um parâmetro (também conhecido como parâmetro formal) é uma variável que faz parte da declaração do método ou do indexador. Um argumento é uma expressão usada ao chamar o método ou indexador. Por exemplo, considere este trecho:

```
void Foo(int x, int y) {
    // Faça algo com x e y
}
...
intuma = 10;
Foo(a, 20);
```

Aqui os parâmetros são xey e os argumentos são a e 20.

Começaremos examinando os parâmetros opcionais.

### 13.1.1 Parâmetros opcionais

O Visual Basic tem parâmetros opcionais há muito tempo e eles estão no CLR desde o .NET 1.0. O conceito é tão óbvio quanto parece: alguns parâmetros são opcionais, portanto seus valores não precisam ser especificados explicitamente pelo chamador. Qualquer parâmetro que não tenha sido especificado como argumento pelo chamador recebe um valor padrão.

#### MOTIVAÇÃO

Parâmetros opcionais são geralmente usados quando vários valores são necessários para uma operação, e os mesmos valores são usados muitas vezes. Por exemplo, suponha que você queira ler um arquivo de texto; você pode querer fornecer um método que permita ao chamador especificar o nome do arquivo e a codificação a ser usada. A codificação é quase sempre UTF-8, então é bom poder usá-la automaticamente se for tudo que você precisa.

Historicamente, a maneira idiomática de permitir isso em C# tem sido usar a sobrecarga de método: declarar um método com todos os parâmetros possíveis e outros que o chamem

---

<sup>1</sup> Ou Cavalleria rusticana e Pagliacci se você se sentir mais culto.

método, passando valores padrão quando apropriado. Por exemplo, você pode criar métodos como este:

```
public IList<Customer> LoadCustomers(string nome do arquivo,
                                         Codificação de codificação)
{
    ...
}

public IList<Cliente> LoadCustomers(string nome do arquivo) {

    retornar LoadCustomers (nome do arquivo, Encoding.UTF8);
} ← Faça um trabalho de verdade aqui ← Padrão para UTF-8
```

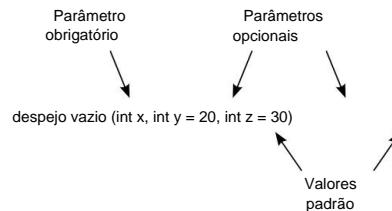
Isso funciona bem para um único parâmetro, mas fica complicado quando há múltiplas opções, pois cada opção extra dobra o número de sobrecargas possíveis. Se dois deles forem do mesmo tipo, esta abordagem levaria naturalmente a vários métodos com a mesma assinatura, o que é inválido. Freqüentemente, o mesmo conjunto de sobrecargas também é necessário para vários tipos de parâmetros. Por exemplo, o método `XmlReader.Create()` pode criar um `XmlReader` a partir de um `Stream`, um `TextReader` ou uma `string`, mas também oferece a opção de especificar um `XmlReaderSettings` e outros argumentos. Devido a esta duplicação, existem 12 sobrecargas para o método.

Isto poderia ser significativamente reduzido com parâmetros opcionais. Vamos ver como isso é feito.

#### DECLARANDO PARÂMETROS OPCIONAIS E OMITINDO-OS AO FORNECER ARGUMENTOS

Tornar um parâmetro opcional é tão simples quanto fornecer um valor padrão para ele, usando o que parece ser um inicializador de variável. A Figura 13.1 mostra um método com três parâmetros: dois são opcionais e um é obrigatório.

Tudo o que esse método faz é imprimir os argumentos, mas isso é o suficiente para ver o que está acontecendo. A listagem a seguir fornece o código completo e chama o método três vezes, especificando um número diferente de argumentos para cada chamada.



**Figura 13.1** Declarando parâmetros opcionais

#### Listagem 13.1 Declarando e chamando um método com parâmetros opcionais

```
static void Dump(int x, int y = 20, int z = 30) {
    Console.WriteLine("x={0} y={1} z={2}", x, y, z);
}
...
Despejar(1, 2, 3);
Despejar(1, 2);
Despejar(1);
```

← Declara método com opcional Parâmetros B

C Chama método com todos os argumentos

← D Omite um argumento

E Omite dois argumentos

Os *parâmetros opcionais* são aqueles com valores padrão especificados B. Se o chamador não especificar y, seu valor inicial será 20, e da mesma forma z terá um valor padrão de 30.

A primeira chamada C especifica explicitamente todos os argumentos; as chamadas restantes (D e E) omitem um ou dois argumentos, respectivamente, portanto, os valores padrão são usados. Quando falta um argumento, o compilador assume que o parâmetro final foi omitido, depois o penúltimo e assim por diante.

A saída é a seguinte:

```
x=1 y=2 z=3 x=1
y=2 z=30 x=1 y=20
z=30
```

Observe que embora o compilador possa usar algumas análises inteligentes dos tipos de parâmetros opcionais e dos argumentos para descobrir o que foi deixado de fora, isso não acontece: ele assume que você está fornecendo argumentos na mesma ordem que o parâmetro -ters.<sup>2</sup> Isso significa que o código a seguir é inválido:

 INVÁLIDO static void TwoOptionalParameters(int x = 10, string y = "default")

```
{
    Console.WriteLine("x={0} y={1}", x, y);
}
...
TwoOptionalParameters("segundo parâmetro");
```

← Erro!

Isso tenta chamar o método TwoOptionalParameters especificando uma string para o *primeiro* argumento. Não há sobrecarga com um primeiro parâmetro que pode ser conversível de uma string, então o compilador emite um erro. Isso é uma coisa boa - a resolução de sobrecarga é bastante complicada (especialmente quando a inferência de tipo genérico está envolvida) sem que o compilador tente todos os tipos de permutações diferentes para encontrar algo que você possa estar tentando chamar.

Se quiser omitir o valor de um parâmetro opcional, mas especificar um posterior, você precisará usar argumentos nomeados.

#### RESTRIÇÕES AOS PARÂMETROS OPCIONAIS

Existem algumas regras para parâmetros opcionais. Todos os parâmetros opcionais devem vir depois dos parâmetros obrigatórios. A exceção a isso é um *array de parâmetros* (conforme declarado com o modificador params), que ainda precisa vir no final de uma lista de parâmetros, mas pode vir depois de parâmetros opcionais. Uma matriz de parâmetros não pode ser declarada como um parâmetro opcional — se o chamador não especificar nenhum valor para ela, uma matriz vazia será usada. Parâmetros opcionais também não podem ter modificadores ref ou out .

Um parâmetro opcional pode ser de qualquer tipo, mas há restrições quanto ao valor padrão especificado. Você sempre pode usar constantes: literais numéricos e de string, null, membros const , membros enum e o operador default(T) . Além disso, para tipos de valor, você pode chamar o construtor sem parâmetros, embora isso seja equivalente a usar o operador padrão (...) de qualquer maneira. Deve haver uma conversão implícita do valor especificado para o tipo de parâmetro, mas não deve ser uma conversão definida pelo usuário.

A Tabela 13.1 mostra alguns exemplos de listas de parâmetros válidos.

<sup>2</sup> A menos que você esteja usando argumentos nomeados, é claro – você aprenderá sobre eles em breve.

**Tabela 13.1** Listas de parâmetros de métodos válidos usando parâmetros opcionais

Declaração	Notas
Foo(int x, int y = 10)	Literal numérico usado como valor padrão
Foo(decimal x = 10)	Conversão integrada implícita de int para decimal
Foo(string nome = "padrão")	Literal de string usado para valor padrão
Foo(DateTime dt = new DateTime())	Valor zero de DateTime
Foo(DateTime dt = padrão(DateTime))	Sintaxe alternativa para o valor zero
Foo<T>(valor T = padrão(T))	O operador de valor padrão funciona com parâmetros de tipo
Foo(int? x = nulo)	Conversão anulável
Foo(int x, int y = 10, parâmetros int[] z)	Matriz de parâmetros após parâmetros opcionais

Por outro lado, a tabela 13.2 mostra algumas listas de parâmetros inválidos e explica por que eles não são permitido.

**Tabela 13.2** Listas de parâmetros de métodos inválidos usando parâmetros opcionais

Declaração (inválida)	Notas
Foo(int x = 0, int y)	Parâmetro não params obrigatório após parâmetro opcional
Foo(DateTime dt = DateTime.Agora)	Os valores padrão devem ser constantes
Foo(XNome nome = "padrão")	A conversão de string para XName é definida pelo usuário
Foo(params string[] nomes = nulo)	Matrizes de parâmetros não podem ser opcionais
Foo(ref string nome = "padrão")	parâmetros ref/out não podem ser opcionais

O fato de o valor padrão ter que ser constante é um problema de duas maneiras diferentes. Um deles é familiar a partir de um contexto ligeiramente diferente, como veremos agora.

#### VERSIONAMENTO E PARÂMETROS OPCIONAIS

As restrições nos valores padrão para parâmetros opcionais podem lembrá-lo da restrições em campos const ou valores de atributos e se comportam de maneira muito semelhante. Em ambos casos, quando o compilador faz referência ao valor, ele o copia diretamente na saída. O IL gerado age exatamente como se seu código-fonte original contivesse o padrão valor. Isto significa que se você alterar o valor padrão sem recompilar tudo que faz referência a ele, os chamadores antigos ainda usarão o valor padrão antigo.

Para tornar isso concreto, imagine este conjunto de etapas:

**1** Crie uma biblioteca de classes (Library.dll) com uma classe como esta:

```
classe pública BibliotecaDemo {  
  
    public static void PrintValue(int valor = 10) {  
  
        System.Console.WriteLine(valor);  
    }  
}
```

**2** Crie um aplicativo de console (Application.exe) que faça referência à biblioteca de classes:

```
aula pública Programa {  
  
    static void Principal() {  
  
        BibliotecaDemo.PrintValue();  
    }  
}
```

**3** Execute o aplicativo – ele imprimirá 10, previsivelmente.

**4** Altere a declaração de PrintValue conforme a seguir e recompile *apenas* o

```
biblioteca de  
classes: public static void PrintValue (int valor = 20)
```

**5** Execute novamente o aplicativo – ele ainda imprimirá 10. O valor foi compilado diretamente no executável.

**6** Recompile o aplicativo e execute-o novamente – desta vez ele imprimirá 20.

Esse problema de controle de versão pode causar bugs difíceis de rastrear, porque todo o código parece correto. Essencialmente, você está restrito ao uso de constantes genuínas que nunca devem ser alteradas como valores padrão para parâmetros opcionais.<sup>3</sup> Há um benefício nesta configuração: ela dá ao chamador uma garantia de que o valor que ele conhecia em tempo de compilação é aquele que irá ser usado. Os desenvolvedores podem se sentir mais confortáveis com isso do que com um valor calculado dinamicamente ou que depende da versão da biblioteca usada no tempo de execução.

Claro, isso também significa que você não pode usar quaisquer valores que não possam ser expressos como constantes. Você não pode criar um método com um valor padrão de “hora atual”, por exemplo.

#### TORNANDO OS PADRÓES MAIS FLEXÍVEIS COM A NULIDADE

Felizmente, há uma maneira de contornar a restrição de que os valores padrão devem ser constantes. Essencialmente, você introduz um valor mágico para representar o padrão e, em seguida, substitui esse valor mágico pelo padrão *real* dentro do próprio método. Se a frase *valor mágico* te incomoda, não me surpreende, mas usaremos null para o valor mágico, que já representa a ausência de um valor normal. Se o tipo de parâmetro normalmente for um tipo de valor, simplesmente o tornaremos o tipo de valor anulável correspondente e, nesse ponto, ainda poderemos especificar que o valor padrão é nulo.

---

<sup>3</sup> Ou você pode simplesmente aceitar que precisará recompilar tudo se alterar o valor. Em muitos contextos, essa é uma troca razoável.

Como exemplo disso, vejamos uma situação semelhante à que usei para apresentar todo o tópico: permitir que o chamador forneça uma codificação de texto apropriada para um método, mas com o padrão UTF-8. Você não pode especificar a codificação padrão como `Encoding.UTF8`, pois não é um valor constante, mas você pode tratar um valor de parâmetro nulo como “usar o padrão”. Para demonstrar como você pode lidar com tipos de valor, vamos fazer o método acrescenta um carimbo de data/hora a um arquivo de texto com uma mensagem. Iremos padronizar a codificação para UTF-8 e o carimbo de data/hora para a hora atual. A listagem a seguir mostra o código completo e alguns exemplos de como usá-lo.

### Listagem 13.2 Usando valores padrão nulos para lidar com situações não constantes

```
static void AppendTimestamp(string nome do arquivo,
                            mensagem de sequência,
                            Codificação codificação = null,
                            Data hora? carimbo de data / hora = null)
{
    Codificação realEncoding = codificação ?? Codificação.UTF8;
    DateTime realTimestamp = carimbo de data e hora ?? DateTime.Agora;
    usando (escritorTextWriter = new StreamWriter (nome do arquivo,
                                                verdadeiro,
                                                codificação real));
    {
        escritor.WriteLine("{0:s}: {1}", realTimestamp, mensagem);
    }
}
...
AppendTimestamp("utf8.txt", "Primeira mensagem");
AppendTimestamp("ascii.txt", "ASCII", Encoding.ASCII);
AppendTimestamp("utf8.txt", "Mensagem no futuro", null,
                novo DateTime(2030, 1, 1));
```

A Listagem 13.2 mostra alguns recursos interessantes dessa abordagem. Primeiro, resolve o versionamento problema. Os valores padrão para os parâmetros opcionais são `null` B, mas os valores efetivos são “a codificação UTF-8” e “a data e hora atuais”. Nenhum destes poderia ser expresso como constantes, e caso você queira alterar o padrão efetivo—por exemplo, para usar a hora UTC atual em vez da hora local – você poderia fazer isso sem ter que recompilar tudo o que chamou `AppendTimestamp`. Claro, alterar o padrão efetivo altera o comportamento do método; você precisa levar o mesmo tipo de cuidado com isso que você teria com qualquer outra alteração de código. Neste ponto, você (como autor da biblioteca) é responsável pela história do versionamento – você está assumindo a responsabilidade de não quebrar clientes, de forma eficaz. Pelo menos é um território mais familiar; você sabe que todos os chamadores terão o mesmo comportamento, independentemente da recompilação.

Esta listagem também introduz um nível extra de flexibilidade. Os parâmetros opcionais não apenas significam que você pode tornar as chamadas mais curtas, mas também ter um valor específico para “usar o padrão” significa que se você desejar, você pode fazer explicitamente uma chamada permitindo o método para escolher o valor apropriado. No momento, esta é a única maneira que você conhece de especificar explicitamente o carimbo de data/hora sem fornecer também uma codificação D, mas isso mudará quando olhamos para argumentos nomeados.

Os valores dos parâmetros opcionais são simples de lidar, graças ao operador de coalescência nula C. Este exemplo usa variáveis separadas para fins de formatação impressa, mas em código real você provavelmente usaria as mesmas expressões diretamente nas chamadas ao construtor StreamWriter e ao método WriteLine .

Há duas desvantagens nessa abordagem: primeiro, significa que se um chamador passar *acidentalmente* nulo devido a um bug, ele obterá o valor padrão em vez de uma exceção. Nos casos em que você está usando um tipo de valor anulável e os chamadores usarão nulo explicitamente ou terão um argumento não anulável, isso não é um grande problema, mas para tipos de referência pode ser um problema.

Por falar no assunto, exige que você não queira usar nulo como um valor “real”.<sup>4</sup> Há ocasiões em que você deseja que *nulo* signifique *nulo* e, se não quiser que esse seja o valor padrão, você teremos que encontrar uma constante diferente ou apenas deixar o parâmetro como obrigatório. Mas em outros casos, onde não há um valor constante óbvio que será *sempre* o padrão correto, recomendo esta abordagem para parâmetros opcionais como uma abordagem fácil de seguir de forma consistente e que elimina algumas das dificuldades normais.

Precisaremos ver como os parâmetros opcionais afetam a resolução de sobrecarga, mas faz sentido esperar até ver como funcionam os argumentos nomeados. Falando nisso...

### 13.1.2 Argumentos nomeados

A ideia básica dos argumentos nomeados é que, ao especificar um valor de argumento, você também pode especificar o nome do parâmetro para o qual está fornecendo o valor. O compilador então certifica-se de que existe *um* parâmetro com o nome correto e usa o valor para esse parâmetro. Mesmo por si só, isso pode aumentar a legibilidade em alguns casos. Na prática, argumentos nomeados são mais úteis em casos onde parâmetros opcionais também podem aparecer, mas examinaremos primeiro a situação simples.

**INDEXADORES, PARÂMETROS OPCIONAIS E ARGUMENTOS NOMEADOS** Você pode usar parâmetros opcionais e argumentos nomeados com indexadores e também com métodos. Mas isso só é útil para indexadores com mais de um parâmetro: você não pode acessar um indexador sem especificar pelo menos um argumento. Dada essa limitação, não espero ver o recurso muito usado com indexadores e não o demonstrei no livro. No entanto, funciona exatamente como você esperaria.

Tenho certeza que você já viu um código parecido com este:

```
MessageBox.Show("Por favor, não pressione este botão novamente", //texto
    "Ai!"); // título
```

Eu escolhi um exemplo bastante inofensivo; pode ficar muito pior quando há muitas discussões, especialmente se muitas delas forem do mesmo tipo. Mas isto ainda é realista; mesmo com

<sup>4</sup> Quase precisamos de um segundo valor especial semelhante a nulo, que significa “use o valor padrão para este parâmetro”, e então poderíamos permitir que esse valor especial fosse fornecido automaticamente para argumentos ausentes ou explicitamente na lista de argumentos. Tenho certeza de que isso causaria dezenas de problemas, mas é uma experiência mental interessante.  
eu quis dizer.

apenas dois parâmetros, eu me pegaria adivinhando qual argumento significava o que com base no texto ao ler este código, a menos que houvesse comentários como os que tenho aqui. Porém, há um problema: os comentários podem mentir sobre o código que descrevem. Nada os verifica. Em contraste, argumentos nomeados pedem ajuda ao compilador.

### SINTAXE

Tudo o que você precisa fazer no exemplo anterior para tornar o código mais claro é prefixar cada argumento com o nome do parâmetro correspondente e dois pontos:

```
MessageBox.Show(text: "Por favor, não pressione este botão novamente",
               legenda: "Ai!");
```

É certo que agora você não pode escolher o nome que considera mais significativo (prefiro o título à legenda), mas pelo menos você saberá se errar alguma coisa.

É claro que a maneira mais comum pela qual você pode entender algo errado aqui é interpretar os argumentos de maneira errada. Sem argumentos nomeados, isso seria um problema: você acabaria com os trechos de texto trocados na caixa de mensagem. Com argumentos nomeados, a ordem torna-se amplamente irrelevante. Você pode reescrever o código anterior assim:

```
MessageBox.Show(caption: "Ai!", text: "Por favor,
                           não pressione este botão novamente");
```

Você ainda teria o texto certo no lugar certo, porque o compilador descobriria o que você quis dizer com base nos nomes.

Para outro exemplo, veja a chamada do construtor StreamWriter na listagem 13.2.

O segundo argumento é simplesmente verdadeiro – o que isso significa? Isso forçará uma liberação de fluxo após cada gravação? Incluir uma marca de ordem de bytes? Anexar a um arquivo existente em vez de criar um novo? Aqui está a chamada equivalente usando argumentos nomeados:

```
novo StreamWriter (caminho: nome do arquivo,
                     acréscimo:
                     verdadeiro, codificação: realEncoding)
```

Em ambos os exemplos, você viu como os argumentos nomeados atribuem efetivamente *significado* semântico aos valores. Na busca incessante para fazer com que o código se comunique melhor com os humanos e também com os computadores, este é um avanço definitivo.

Não estou sugerindo que argumentos nomeados devam ser usados quando o significado já é óbvio, é claro. Como todos os recursos, deve ser usado com discrição e reflexão.

### Argumentos nomeados com out e ref

Se você quiser especificar o nome de um argumento para um parâmetro ref ou out , coloque o modificador ref ou out após o nome e antes do argumento. Usando int.TryParse como exemplo, você pode ter um código como este:

```
número interno;
bool sucesso = int.TryParse("10", resultado: número de saída);
```

Para explorar alguns outros aspectos da sintaxe, a listagem a seguir mostra um método com três parâmetros inteiros, exatamente como aquele que usamos para começar a examinar os parâmetros opcionais.

#### Listagem 13.3 Exemplos simples de uso de argumentos nomeados

```
static void Dump(int x, int y, int z) {  
    Declarar método  
    B normalmente  
  
    Console.WriteLine("x={0} y={1} z={2}", x, y, z);  
}  
...  
Despejar(1, 2, 3);  
Despejar(x: 1, y: 2, z: 3);  
Despejar(z: 3, y: 2, x: 1);  
Despejar(1, y: 2, z: 3);  
Despejar(1, z: 3, y: 2);  
  
C Chama o método  
normalmente  
Especifica nomes para  
D todos os argumentos  
Especifica nomes para  
E alguns argumentos
```

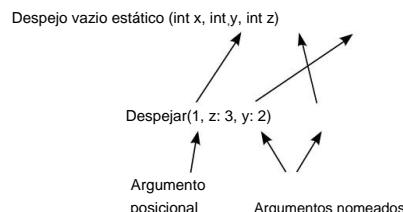
A saída é a mesma para cada chamada na listagem 13.3:  $x=1, y=2, z=3$ . Esse código efetivamente faz a mesma chamada de método de cinco maneiras diferentes. Vale a pena notar que não há truques na declaração do método B; você pode usar argumentos nomeados com qualquer método que possua parâmetros. Primeiro, você chama o método da maneira normal, sem usar nenhum recurso novo C. Isso é uma espécie de ponto de controle para ter certeza de que as outras chamadas são realmente equivalentes. Em seguida, você faz duas chamadas ao método usando apenas os argumentos nomeados D. A segunda dessas chamadas inverte a ordem dos argumentos, mas o resultado ainda é o mesmo, porque os argumentos são comparados aos parâmetros por nome, não por posição. Finalmente, há duas chamadas usando uma mistura de argumentos nomeados e *argumentos posicionais* E. Um argumento posicional é aquele que não é nomeado, portanto, todo argumento no código C# 3 válido é um argumento posicional do ponto de vista do C# 4.

A Figura 13.2 mostra como funciona a linha final do código.

Todos os argumentos nomeados devem vir depois dos argumentos posicionais — você não pode alternar entre os estilos. Argumentos posicionais *sempre* se referem ao parâmetro correspondente na declaração do método — você não pode fazer com que argumentos posicionais ignorem um parâmetro especificando-o posteriormente com um argumento nomeado. Isso significa que essas chamadas de método seriam inválidas:

- ✗ Dump(z: 3, 1, y: 2)—Os argumentos posicionais devem vir antes dos nomeados.
- ✗ Dump(2, x: 1, z: 3)—x já foi especificado pelo primeiro argumento posicional, então você não pode especificá-lo novamente com um argumento nomeado.

Agora, embora *neste caso específico* as chamadas de método sejam equivalentes, nem *sempre* é esse o caso. Vejamos por que a reordenação de argumentos pode mudar o comportamento.



**Figura 13.2** Argumentos posicionais e nomeados na mesma chamada

**ORDEM DE AVALIAÇÃO DE ARGUMENTOS**

Você está acostumado com o C# avaliando seus argumentos na ordem em que são especificados, que, até o C# 4, sempre foi a ordem em que os parâmetros também foram declarados. No C# 4, apenas a primeira parte ainda é verdadeira: os argumentos ainda são avaliados na ordem em que foram escritos, mesmo que essa não seja a mesma ordem em que foram declarados como parâmetros. Isso é importante se a avaliação dos argumentos tiver efeitos colaterais.

Geralmente vale a pena tentar evitar efeitos colaterais nas discussões, mas há casos em que isso pode tornar o código mais claro. Uma regra mais realista é tentar evitar efeitos colaterais que possam interferir uns nos outros. Para demonstrar a ordem de execução, quebraremos essas duas regras. Por favor, não trate isso como uma recomendação para que você faça a mesma coisa.

Primeiro, criaremos um exemplo relativamente inofensivo, introduzindo um método que registra sua entrada e a retorna — uma espécie de eco de registro. Usaremos os valores de retorno de três chamadas para chamar o método Dump (que não é mostrado, pois não foi alterado). A listagem a seguir mostra duas chamadas para Dump que resultam em resultados ligeiramente diferentes.

**Listagem 13.4 Avaliação do argumento de log**

```
log int estático (valor int) {  
  
    Console.WriteLine("Log: {0}", valor); valor de retorno;  
  
}  
...  
Despejo(x: Log(1), y: Log(2), z: Log(3));  
Despejo(z: Log(3), x: Log(1), y: Log(2));
```

Os resultados da listagem 13.4 mostram o que acontece:

```
Registro: 1  
Registro: 2  
Registro:  
3 x=1 y=2 z=3  
Registro: 3  
Registro:  
2 x=1 y=2 z=3
```

Em ambos os casos, os parâmetros x, y e z no método Dump ainda possuem os valores 1, 2 e 3, respectivamente. Mas você pode ver que embora eles tenham sido avaliados nessa ordem na primeira chamada (o que equivalia ao uso de argumentos posicionais), a segunda chamada avaliou primeiro o valor usado para o parâmetro z .

Você pode tornar o efeito ainda mais significativo usando efeitos colaterais que alteram os resultados da avaliação do argumento, conforme mostrado na listagem a seguir, novamente usando o mesmo método Dump .

### Listagem 13.5 Abuso da ordem de avaliação de argumentos

```
int eu = 0;
Despejar(x: ++i, y: ++i, z: ++i);
eu = 0;
Despejar(z: ++i, x: ++i, y: ++i);
```

Os resultados da listagem 13.5 podem ser melhor expressos em termos do padrão de respingos de sangue em uma cena de crime, depois que alguém que mantinha um código como esse persegue o autor original com um machado. Sim, *teoricamente falando*, os resultados da última linha são `x=2 y=3 z=1`, mas tenho certeza que você entende onde estou chegando. Apenas diga *não* a códigos como este.

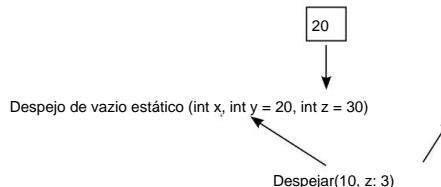
Certamente, reordene seus argumentos para facilitar a leitura. Você pode pensar que fazer uma chamada para `MessageBox.Show` com o título acima do texto no próprio código reflete o layout da tela mais de perto, por exemplo. Se você quiser confiar porém, em uma ordem de avaliação específica para os argumentos, introduza algumas variáveis locais para executar o código relevante em instruções separadas. O compilador não vai se importar de qualquer forma - seguirá as regras da especificação - mas isso reduz o risco de um "inofensivo refatoração" que inadvertidamente introduz um bug sutil.

Para voltar aos assuntos mais alegres, vamos combinar os dois recursos (parâmetros opcionais e argumentos nomeados) e veja o quanto mais organizado o código pode ser.

### 13.1.3 Juntando os dois

Parâmetros opcionais e argumentos nomeados funcionam em conjunto sem nenhum esforço extra exigido de sua parte. Não é incomum ter vários parâmetros onde há são padrões óbvios, mas é difícil prever quais deles o chamador desejará especifique explicitamente. A Figura 13.3 mostra praticamente todas as combinações: um parâmetro obrigatório, dois parâmetros opcionais, um argumento posicional, um argumento nomeado e um argumento ausente para um parâmetro opcional.

Voltando a um exemplo anterior, na listagem 13.2 você queria anexar um timestamp para um arquivo usando o padrão codificação de UTF-8, mas com um carimbo de data/hora específico. Esse código usou `null` para o argumento de codificação, mas agora você pode escrever o mesmo código de forma mais simples, conforme mostrado na listagem a seguir.



**Figura 13.3** Misturando argumentos nomeados e parâmetros opcionais

### Listagem 13.6 Combinando argumentos nomeados e parâmetros opcionais

```
static void AppendTimestamp(string nome do arquivo,
                            mensagem de sequência,
                            Codificação codificação = null,
                            Data hora? carimbo de data / hora = null)
{
}
...
```

Mesma implementação de antes

```
AppendTimestamp("utf8.txt", "Mensagem no futuro", timestamp: new DateTime(2030, 1,
    1));
```

Nesta situação bastante simples, o benefício não é particularmente grande, mas nos casos em que você deseja omitir três ou quatro argumentos, mas especificar o final, é uma verdadeira bênção.

Você viu como os parâmetros opcionais reduzem a necessidade de longas listas de sobrecargas, mas um padrão específico onde vale a pena mencionar isso é com relação à imutabilidade.

#### IMUTABILIDADE E INICIALIZAÇÃO DE OBJETOS

Um aspecto do C# 4 que me decepciona um pouco é que ele não fez muito *explicitamente* para facilitar a imutabilidade. Tipos imutáveis são uma parte essencial da programação funcional, e o C# tem gradualmente suportado cada vez mais o estilo funcional... exceto pela imutabilidade. Os inicializadores de objetos e coleções facilitam o trabalho com tipos *mutáveis*, mas os tipos imutáveis foram deixados de lado. (As propriedades implementadas automaticamente também se enquadram nesta categoria.) Felizmente, embora não sejam especialmente projetadas para ajudar na imutabilidade, argumentos nomeados e parâmetros opcionais permitem escrever código semelhante ao inicializador de objeto que chama um construtor ou outro método de fábrica.

Por exemplo, suponha que você estivesse criando uma classe `Message`, que exigia um endereço `remetente`, um endereço `para` e um `corpo`, com o assunto e o anexo sendo opcionais.  
(Ficaremos com destinatários únicos para manter o exemplo o mais simples possível.)  
Você *poderia* criar um tipo mutável com propriedades graváveis apropriadas e construir instâncias como esta:

```
Mensagem mensagem = nova mensagem { From =
    "skeet@pobox.com", To = "csharp-in-
profundidade-readers@everywhere.com", Body = "Espero que goste da
terceira edição", Assunto = "Uma mensagem rápida"

};
```

Isso tem dois problemas: primeiro, não impõe o fornecimento dos dados necessários. Você poderia forçar que eles fossem fornecidos ao construtor, mas (antes do C # 4) não seria óbvio qual argumento significava o quê:

```
Mensagem mensagem = nova
    mensagem( "skeet@pobox.com",
    "csharp-in-profundidade-readers@everywhere.com", "Espero que
    gostem da terceira edição"
{
    Assunto = "Uma mensagem rápida"
};
```

O segundo problema é que esse padrão de inicialização simplesmente não funciona para tipos imutáveis. O compilador precisa chamar um configurador de propriedades *depois* de inicializar o objeto.

Mas você poderá usar parâmetros opcionais e argumentos nomeados para criar algo que tenha as características interessantes da primeira forma (especificando apenas o que lhe interessa)

entrada e fornecimento de nomes) sem perder a validação de quais aspectos da mensagem são necessários ou os benefícios da imutabilidade. A listagem a seguir mostra uma possível assinatura do construtor e a etapa de construção para a mesma mensagem anterior.

#### Listagem 13.7 Usando parâmetros opcionais e argumentos nomeados para imutabilidade

```
Mensagem pública (string de, string para,
    corpo da string, string subject = null, byte[] anexo = null)
```

```
{
}
...
}
```

```
Mensagem mensagem = nova mensagem( de:
    "skeet@pobox.com", para: "csharp-in-
    profundidade-readers@everywhere.com", corpo: "Espero que goste da
    terceira edição", assunto: "Uma mensagem rápida "
```

```
);
```



Inicialização normal  
o código vai aqui

Eu realmente gosto disso em termos de legibilidade e limpeza geral. Você não precisa de centenas de sobrecargas de construtor — apenas uma com alguns dos parâmetros sendo opcionais.

A mesma sintaxe também funcionará com métodos de criação estáticos, diferentemente dos inicializadores de objetos. A única desvantagem é que ele realmente depende de seu código ser consumido por uma linguagem que suporta parâmetros opcionais e argumentos nomeados; caso contrário, os chamadores serão forçados a escrever um código feio para especificar valores para todos os parâmetros opcionais. Obviamente, a imutabilidade envolve mais do que apenas obter valores no código de inicialização, mas mesmo assim este é um passo bem-vindo na direção certa.

Há alguns pontos finais a serem feitos sobre esses recursos antes de passarmos para COM — pontos relativos aos detalhes de como o compilador lida com seu código e a dificuldade de um bom design de API .

#### RESOLUÇÃO DE SOBRECARGA

É claro que tanto os argumentos nomeados quanto os parâmetros opcionais afetam o modo como o compilador resolve as sobrecargas — se houver diversas assinaturas de métodos disponíveis com o mesmo nome, qual delas ele deverá escolher? Parâmetros opcionais podem *aumentar* o número de métodos aplicáveis (se alguns métodos tiverem mais parâmetros do que o número de argumentos especificados) e argumentos nomeados podem *diminuir* o número de métodos aplicáveis (descartando métodos que não possuem os nomes de parâmetros apropriados).

Na maior parte, as mudanças são intuitivas: para verificar se algum método específico é aplicável, o compilador tenta construir uma lista dos argumentos que passaria , usando os argumentos posicionais em ordem e, em seguida, combinando os argumentos nomeados com os parâmetros restantes. Se um parâmetro obrigatório não tiver sido especificado ou se um argumento nomeado não corresponder a nenhum parâmetro restante, o método não será aplicável. A especificação fornece mais detalhes sobre isso na seção 7.5.3, mas há duas situações para as quais gostaria de chamar atenção especial.

Primeiro, se dois métodos são aplicáveis e um deles recebeu *todos* os seus argumentos explicitamente, enquanto o outro usa um parâmetro opcional preenchido com um valor padrão, o método que não usa nenhum valor padrão vencerá. Mas isso não se estende apenas à comparação do número de valores padrão usados – é uma divisão estrita do tipo “ele usa valores padrão ou não”. Por exemplo, considere o seguinte:

```
static void Foo(int x = 10) {} static void Foo(int x =
10, int y = 20) {}
...
Foo();
Foo(1);
Foo(y: 2);
Foo(1, 2);
```

Na primeira chamada B, ambos os métodos são aplicáveis devido aos seus parâmetros opcionais. Mas o compilador não consegue descobrir qual deles você pretendia chamar; isso gerará um erro. Na segunda chamada C, ambos os métodos ainda são aplicáveis, mas a primeira sobrecarga é usada porque pode ser aplicada sem usar nenhum valor padrão, enquanto a segunda sobrecarga usa o valor padrão para y. Tanto para a terceira como para a quarta chamadas, apenas a segunda sobrecarga é aplicável. A terceira chamada D nomeia o argumento y, e a quarta chamada E tem dois argumentos; ambos significam que a primeira sobrecarga não é aplicável.

**SOBRECARGAS E HERANÇA NEM SEMPRE SE MISTURAM BEM.** Tudo isso pressupõe que o compilador chegou ao ponto de encontrar múltiplas sobrecargas para escolher entre elas. Se alguns métodos forem declarados em um tipo base, mas existirem métodos aplicáveis em um tipo mais derivado, o último vencerá. Este sempre foi o caso e pode causar alguns resultados surpreendentes (veja o site do livro para mais detalhes e exemplos: <http://mng.bz/aEmE>), mas agora os parâmetros opcionais significam que pode haver métodos mais aplicáveis do que você esperaria. Aconselho você a evitar sobreclarregar um método de classe base dentro de uma classe derivada, a menos que obtenha um grande benefício.

O segundo ponto é que às vezes argumentos nomeados podem ser uma alternativa à conversão para ajudar o compilador a resolver sobrecargas. Às vezes, uma chamada pode ser ambígua porque os argumentos podem ser convertidos em tipos de parâmetros em dois métodos diferentes, mas nenhum dos métodos é melhor que o outro em todos os aspectos. Por exemplo, considere as seguintes assinaturas e chamadas de método:

```
Método void (int x, objeto y) { ... } Método void (objeto a, int b)
{ ... }
...
Método(10, 10);
```

Ambos os métodos são aplicáveis e nenhum é melhor que o outro. Existem duas maneiras de resolver isso, supondo que você não possa alterar os nomes dos métodos para torná-los inequívocos dessa forma. (Essa é minha abordagem preferida. Torne o nome de cada método mais informativo e específico, o que melhorará a legibilidade geral do código.) Você

pode lançar um dos argumentos explicitamente ou usar argumentos nomeados para resolver a ambiguidade:

```
Método void (int x, objeto y) { ... } Método void (objeto a,
int b) { ... }
...
Método(10, (objeto) 10);
Método(x: 10, y: 10);
```



Claro, isso só funciona se os parâmetros tiverem nomes diferentes nos diferentes métodos, mas é um truque útil de saber. Às vezes, o elenco fornecerá um código mais legível; às vezes o nome vai. É apenas uma arma extra na luta por um código claro.

Infelizmente, tem uma desvantagem, juntamente com argumentos nomeados em geral: outra coisa é ter cuidado ao alterar os nomes dos parâmetros.

### O HORROR SILENCIOSO DE MUDAR NOMES

No passado, os nomes dos parâmetros não importavam muito se você usasse apenas C#.

Outras linguagens podem ter se importado, mas em C# as únicas vezes em que os nomes dos parâmetros foram importantes foram quando você estava olhando para o IntelliSense e quando estava olhando para o próprio código do método. Agora, os nomes dos parâmetros de um método fazem efetivamente parte da API , mesmo se você estiver usando apenas C#. Se você alterá-los posteriormente, o código poderá quebrar – qualquer coisa que estivesse usando um argumento nomeado para se referir a um de seus parâmetros falhará na compilação se você decidir alterá-lo. Isso pode não ser um grande problema se o seu código for consumido apenas por si mesmo, mas se você estiver escrevendo uma API pública, esteja ciente de que alterar o nome de um parâmetro é um grande problema. Sempre foi, na verdade, mas se tudo que chama o código foi escrito em C#, você conseguiu ignorar isso até agora.

Renomear parâmetros é ruim; mudar os nomes é pior. O código de chamada ainda pode ser compilado, mas com um significado diferente. Uma forma particularmente maligna disso é substituir um método e mudar os nomes dos parâmetros na versão substituída. O compilador sempre examinará a substituição mais profunda que conhece, com base no tipo estático da expressão usada como destino da chamada do método. Você não quer entrar em uma situação em que chamar a mesma implementação de método com a mesma lista de argumentos resulte em um comportamento diferente com base no tipo estático de uma variável.

### PARA CONCLUIR...

Argumentos nomeados e parâmetros opcionais são possivelmente dois dos recursos mais simples do C# 4 e, ainda assim, possuem uma quantidade razoável de complexidade, como você viu. As ideias básicas são facilmente expressas e compreendidas, e a boa notícia é que na maioria das vezes isso é tudo com que você precisa se preocupar. Você pode aproveitar os parâmetros opcionais para reduzir o número de sobrecargas escritas, e os argumentos nomeados podem tornar o código muito mais legível quando vários argumentos facilmente confundíveis são usados.

A parte mais complicada provavelmente é decidir quais valores padrão usar, tendo em mente possíveis problemas de versão. Da mesma forma, agora é mais óbvio do que antes que os nomes dos parâmetros são importantes, e você precisa ter cuidado ao substituir métodos existentes, para evitar ser malvado com seus chamadores.

Falando em mal, vamos passar às novidades relacionadas ao COM. Eu sou apenas uma criança... principalmente, de qualquer maneira.

## 13.2 Melhorias para interoperabilidade COM

Admito prontamente que estou longe de ser um especialista em COM . Quando tentei usá-lo antes do surgimento do .NET , sempre me deparei com problemas que, sem dúvida, eram parcialmente causados por minha falta de conhecimento e parcialmente causados por componentes com os quais eu estava trabalhando sendo mal projetados ou implementados. Minha impressão geral do COM como uma espécie de magia negra perdurou. Fui informado com segurança de que há muito o que gostar nisso, mas infelizmente não voltei para aprender em detalhes - e parece haver muitos *detalhes* para estudar.

**ESTA SEÇÃO É ESPECÍFICA DA MICROSOFT** As alterações na interoperabilidade COM não farão sentido para todos os compiladores C#, e um compilador ainda pode ser considerado compatível com a especificação sem implementar esses recursos.

O .NET tornou o COM um pouco mais amigável em geral, mas até agora havia vantagens distintas em usá-lo no Visual Basic em vez do C#. O campo de jogo foi nivelado significativamente pelo C# 4, como você verá nesta seção. Para fins de familiaridade, usarei o Word como exemplo neste capítulo e o Excel no próximo capítulo.

Porém, não há nada específico do Office sobre os novos recursos; você deve achar que a experiência de trabalhar com COM é melhor em C # 4, independentemente do que estiver fazendo.

### 13.2.1 Os horrores da automação do Word antes do C# 4

Nosso exemplo é simples: basta iniciar o Word, criar um documento com um único parágrafo de texto, salvá-lo e sair. Parece fácil, certo? Se ao menos fosse assim. A listagem a seguir mostra o código necessário antes do C# 4.

#### Listagem 13.8 Criando e salvando um documento em C# 3

```
objeto ausente = Type.Missing;
Aplicativo app = novo Aplicativo { Visivel = true }; app.Documents.Add(ref ausente,
ref ausente, ← B inicia palavra
← C Cria um novo documento
    falta de referência, falta de referência);
Documento doc = app.ActiveDocument; Parágrafo
para = doc.Paragraphs.Add(ref faltando); para.Range.Text = "Graças a Deus
pelo C# 4";
nome do arquivo do objeto = "demo.doc";
formato do objeto = WdSaveFormat.wdFormatDocument97; doc.SaveAs(nome
do arquivo ref, formato ref,
    ref faltando, ref faltando, ref faltando, ref
    faltando, ref faltando, ref faltando, ref faltando,
    ref faltando, ref faltando, ref faltando, ref
    faltando, ref faltando); ← D Salva o documento
doc.Close(ref faltando, ref faltando, ref faltando); app.Application.Quit(ref
faltando, ref faltando, ref faltando); ← E desliga o Word
```

Cada etapa deste código parece simples: primeiro você cria uma instância do COM tipo **B** e a torna visível usando uma expressão inicializadora de objeto; então você cria e preenche um novo documento C. O mecanismo para inserir algum texto em um documento não é tão simples quanto você poderia esperar, mas vale lembrar que um documento do Word pode ter uma estrutura bastante complexa; isso não é tão ruim quanto poderia ser. Algumas chamadas de método aqui possuem parâmetros opcionais de referência; você não precisa deles, então você passa uma variável local por referência com um valor `Type.Missing`. Se você já fez algum trabalho COM antes, provavelmente está familiarizado com esse padrão.

A seguir vem a parte realmente desagradável: salvar o documento D. Sim, o método `SaveAs` realmente possui 16 parâmetros, dos quais você está usando apenas 2. Mesmo esses 2 precisam ser passados por referência, o que significa criar variáveis locais para eles. Em termos de legibilidade, este é um pesadelo completo. Não se preocupe, em breve resolveremos isso.

Por fim, você fecha o documento e a aplicação E. Além do fato de ambas as chamadas terem três parâmetros opcionais com os quais você não se importa, não há nada de interessante aqui.

Vamos começar usando os recursos que já vimos neste capítulo – eles são suficientes por si só para reduzir significativamente o exemplo.

### 13.2.2 A vingança de parâmetros opcionais e argumentos nomeados

Comecemos pelo princípio: vamos nos livrar de todos os argumentos correspondentes a parâmetros opcionais nos quais você não está interessado. Isso também significa que você não precisa da variável ausente .

Isso ainda deixa você com 2 parâmetros de 16 possíveis para o método `SaveAs` . No momento é óbvio qual é qual baseado nos nomes das variáveis locais, mas e se você os colocar ao contrário? Todos os parâmetros são digitados de maneira fraca, então você realmente está fazendo suposições. Você pode facilmente dar nomes aos argumentos para esclarecer a chamada. Se você quiser usar um dos parâmetros posteriores, terá que especificar o nome de qualquer maneira, apenas para pular aqueles nos quais não está interessado.

A listagem a seguir mostra o código – ele já parece muito mais limpo.

#### Listagem 13.9 Automatizando o Word usando recursos normais do C# 4

```
Aplicativo app = novo Aplicativo { Visível = true }; app.Documents.Add(); Documento
doc = app.ActiveDocument;
Parágrafo para = doc.Paragraphs.Add();
para.Range.Text = "Graças a Deus pelo C# 4";

nome do arquivo do objeto = "demo.doc";
formato do objeto = WdSaveFormat.wdFormatDocument97;
doc.SaveAs(FileName: nome do arquivo ref, FileFormat: formato ref);

doc.Fechar();
app.Application.Quit();
```

Isso é muito melhor, embora ainda seja feio ter que criar variáveis locais para os argumentos `SaveAs` que você está especificando. Além disso, se você leu com atenção, pode estar preocupado com os parâmetros opcionais que foram removidos. Eles foram árbitros

parâmetros - mas opcionais - que não é uma combinação que o C# normalmente suporta. O que está acontecendo?

### 13.2.3 Quando um parâmetro ref não é um parâmetro ref?

C# normalmente segue uma linha bastante rígida nos parâmetros de referência . Você tem que marcar o argumento com ref assim como o parâmetro, para mostrar que você entende o que está acontecendo – que sua variável pode ter seu valor alterado pelo método que você está chamando.

Tudo bem no código normal, mas as APIs COM geralmente usam parâmetros de referência para quase *tudo*, por motivos de desempenho percebidos — elas geralmente não modificam a variável que você passa. Passar argumentos por referência é um pouco doloroso em C#. Você não apenas precisa especificar o modificador ref , mas também deve ter uma variável. Você não pode simplesmente passar valores por referência.

No C# 4, o compilador torna isso muito mais fácil, permitindo que você passe um argumento por valor para um método COM , mesmo que seja para um parâmetro ref . Considere uma chamada como esta, onde o argumento pode ser uma variável do tipo string, mas o parâmetro é declarado como objeto ref:

```
comObject.SomeMethod(argumento);
```

O compilador emite código equivalente a este:

```
objeto tmp = argumento;
comObject.SomeMethod(ref tmp);
```

Observe que quaisquer alterações feitas por SomeMethod são descartadas, portanto a chamada realmente se comporta como se você estivesse passando argumento por valor. Este mesmo processo é usado para parâmetros ref opcionais; cada um envolve uma variável local inicializada em Type.Missing e passada por referência para o método COM . Se você descompilar o código C# simplificado, verá que o IL emitido é bastante volumoso com todas essas variáveis extras.

Agora você pode aplicar os retoques finais ao exemplo do Word, conforme mostrado na listagem a seguir.

#### Listagem 13.10 Passando argumentos por valor em métodos COM

```
Aplicativo app = novo Aplicativo { Visível = true }; app.Documents.Add(); Documento
doc = app.ActiveDocument;
Parágrafo para = doc.Paragraphs.Add();
para.Range.Text = "Graças a Deus pelo C# 4";
doc.SaveAs(Nome do arquivo: "test.doc",
← Argumentos passados por  
valor FileFormat: WdSaveFormat.wdFormatDocument97);
doc.Fechar();
app.Application.Quit();
```

Como você pode ver, o resultado final é um código muito mais limpo do que o inicial. Com uma API como o Word, você ainda precisa trabalhar com um conjunto um tanto confuso de métodos, propriedades e eventos nos tipos principais, como Aplicativo e Documento, mas pelo menos seu código será muito mais fácil de ler.

Em termos de alterações no código-fonte, há um aspecto final do suporte COM para veja antes de passarmos para as melhorias de implantação disponíveis no C# 4.

#### 13.2.4 Chamando indexadores nomeados

Vários aspectos do C# 4 fornecem suporte para recursos que o Visual Basic utiliza há muito tempo, e este é outro. O CLR, o COM e o Visual Basic permitem propriedades sem falhas com parâmetros - denominados *indexadores* em termos C#. Até a versão 4, o C# não apenas proibia você de declarar diretamente seus próprios indexadores nomeados,<sup>5</sup> mas também não fornecia uma maneira de acessá-los usando sintaxe de propriedade. O único indexador que você pode usar em C# é aquele declarado como *propriedade padrão* para o tipo. Este não tem sido um grande problema para componentes .NET escritos em Visual Basic, já que indexadores nomeados geralmente são desencorajados. Mas os componentes COM , como os do Office, os utilizam com mais intensidade. O C# 4 permite chamar indexadores nomeados de uma maneira mais natural, mas você ainda não pode declará-los para seus próprios tipos de C#.

**CONFLITOS DE TERMINOLOGIA NOVAMENTE** Usei o termo *indexador* ao longo desta seção para descrever o que em termos VB seria conhecido como uma *propriedade parametrizada*. A especificação CLI chama isso de *propriedade indexada*. Qualquer que seja a terminologia, ela é declarada como uma propriedade na IL e possui parâmetros. O indexador normal (no que diz respeito ao C#) é definido pelo *membro padrão* (ou *propriedade padrão*) para o tipo — por exemplo, o membro padrão do StringBuilder é a propriedade Chars (que possui um parâmetro Int32 ). Quando falo sobre *indexadores nomeados* aqui, estou falando daqueles que *não* são o padrão para o tipo, então você deve referenciá-los pelo nome.

Usaremos o Word novamente como exemplo, desta vez mostrando os diferentes significados das palavras. O tipo \_Application no Word define um indexador chamado SynonymInfo com uma declaração como esta:

```
SynonymInfo SynonymInfo[string Palavra,  
                        objeto ref LangaugeId = Type.Missing]
```

Essa não é uma sintaxe C# válida, porque você não pode declarar um indexador nomeado, mas espero que seja óbvio o que isso significa. O nome do indexador é SynonymInfo. Ele *retorna* uma referência a um objeto SynonymInfo e possui dois parâmetros, um dos quais é opcional. (O fato de o nome do indexador e o nome do tipo de retorno serem iguais neste caso é inteiramente coincidência.)

O SynonymInfo pode ser usado para encontrar significados para a palavra e sinônimos para cada significado. A listagem a seguir mostra três maneiras diferentes de usar o indexador para exibir o número de significados de três palavras diferentes.

<sup>5</sup> Diretamente, pelo menos. Você pode aplicar System.Runtime.CompilerServices.IndexerNameAttribute manualmente, mas não é algo que o C# conheça como linguagem.

### Listagem 13.11 Usando o indexador `SynonymInfo` para contar significados de palavras

```

static void ShowInfo (informações de SynonymInfo)
{
    Console.WriteLine("{0} tem {1} significados", info.Word,
                      info.MeaningCount);
}
...
Aplicativo app = novo Aplicativo { Visível = false };

objeto ausente = Type.Missing;
ShowInfo(app.get_SynonymInfo("doloroso", ref ausente));
ShowInfo(app.SynonymInfo["legal", WdLanguageID.wdEnglishUS]);
ShowInfo(app.SynonymInfo[Palavra: "recursos"]);

app.Application.Quit();

```

**Aproveita  
o parâmetro  
opcional**

**D**

**B** Usa anteriormente  
Sintaxe C#

**C** Usa indexador  
com dois  
Argumentos

Mesmo sem indexadores nomeados, os recursos anteriores que você viu teriam ajudado a aliviar o sofrimento da sintaxe B anterior do C#: você poderia ter chamado `app.get_SynonymInfo("better")` e aproveitado os parâmetros opcionais, por exemplo. Mas você pode ver na segunda e terceira chamadas `ShowInfo` (C e D) que a sintaxe do indexador parece menos estranha do que a chamada `get_`. Você poderia argumentar que esta deveria ser uma chamada de método de qualquer maneira, ou que deveria haver uma propriedade `SynonymInfo` sem parâmetros que retornasse uma coleção com um indexador padrão apropriado. Esse é um caso do argumento geral dado pelos designers de C# para não implementar suporte *completo* para indexadores nomeados, incluindo declará-los em C#. Mas a questão é que ele já é um indexador no Word, então é bom poder usá-lo dessa forma.<sup>6</sup> A segunda chamada `ShowInfo` C usa o recurso de parâmetro `ref` implícito da seção 13.2.3, e a terceira chamada D omite o parâmetro opcional e nomeia o argumento restante apenas por diversão.

Há uma pequena diferença nos parâmetros e indexadores opcionais: se *todos* os parâmetros forem opcionais e você não quiser especificar nenhum argumento, será necessário omitir os colchetes. Em vez de escrever `foo.Indexer[]`, você usaria `foo.Indexer`. Tudo isso se aplica tanto para obter do indexador quanto para configurá-lo.

Até agora, tudo bem, mas escrever o código é apenas parte da batalha. Você geralmente precisa ser capaz de implantá-lo em outras máquinas também. Novamente, o C# 4 torna essa tarefa mais fácil.

#### 13.2.5 Vinculando assemblies de interoperabilidade primários

Ao construir em um tipo COM, você usa um assembly gerado para a biblioteca de componentes. Normalmente você usa um *Primary Interop Assembly* (PIA), que é o assembly de interoperabilidade canônico para uma biblioteca COM, assinado pelo editor. Você pode gerá-los usando a ferramenta Type Library Importer (tlbimp) para suas próprias bibliotecas COM. Os PIAs facilitam a vida em termos de ter uma maneira verdadeira de acessar os tipos COM, mas são um problema de outras maneiras. Eles podem ser bem grandes e toda a PIA precisa estar presente

<sup>6</sup> Poderia ter sido mais interessante exibir os significados reais, mas isso leva a problemas de interoperabilidade que não são relevantes para este capítulo.

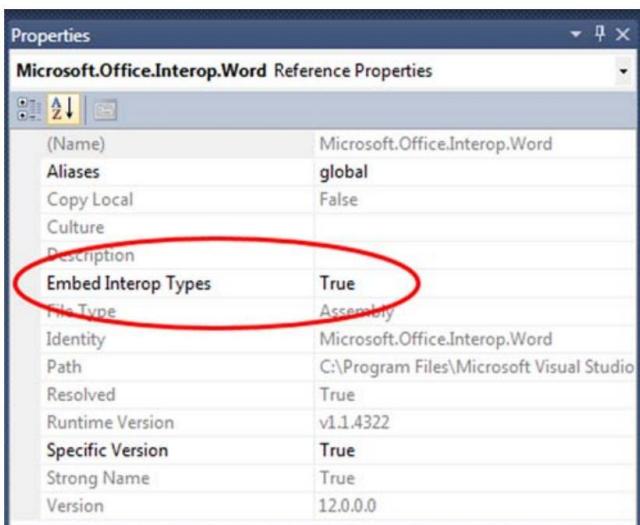


Figura 13.4 Vinculando PIAs no Visual Studio 2010

mesmo se você estiver usando apenas um pequeno subconjunto da funcionalidade. Além disso, você precisa ter a mesma versão do PIA na máquina de implantação daquela em que você compilou.

Isso pode ser estranho em situações em que problemas de licenciamento impedem que você redistribua a própria PIA , contanto com a versão correta já instalada. Se houver várias versões disponíveis, mas todas elas expõem a funcionalidade necessária, talvez seja necessário enviar versões diferentes do seu código para que as referências funcionem.

C# 4 permite uma abordagem muito diferente. Em vez de *fazer referência* a um PIA como qualquer outro assembly, você pode *vincular* -lo. No Visual Studio 2010 e superior, esta é uma opção nas propriedades da referência do assembly, conforme mostrado na figura 13.4.

Os fãs da linha de comando podem usar a opção /l (ou /link) em vez de /r (ou /reference) para vincular em vez de referência:

```
csc /l:Caminho\Para\PIA.dll MyCode.cs
```

Quando você vincula uma PIA, o compilador incorpora apenas os bits necessários da PIA diretamente em seu próprio assembly. Leva apenas os tipos necessários e apenas os membros desses tipos. Por exemplo, o compilador cria estes tipos para o código que vimos neste capítulo:

```
espaço para nome Microsoft.Office.Interop.Word {  
  
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")] interface pública _Application  
  
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")] interface pública _Document  
  
[ComImport, CompilerGenerated, TypeIdentifier, Guid("...")] interface pública Aplicação:  
_Application  
  
[ComImport, Guid("..."), TypeIdentifier, CompilerGenerated] interface pública Documento:  
_Document
```

```
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
interface pública Documentos : IEnumable
{
    [TypeIdentifier("...", "WdSaveFormat"), CompilerGenerated]
    enum público WdSaveFormat
}

}
```

Se você olhar na interface `_Application`, ficará assim:

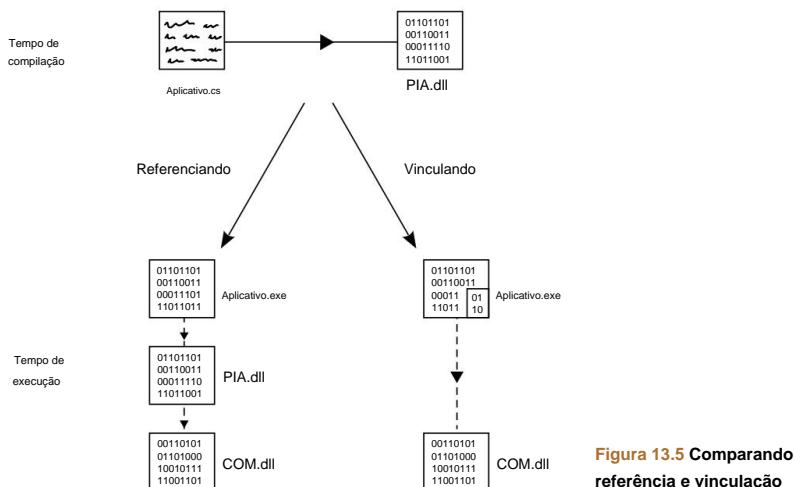
```
[ComImport, TypeIdentifier, CompilerGenerated, Guid("...")]
interface pública _Application {
    void_VtblGap 1_4(); Documentos
    Documentos { [...] obter; } void_VtblGap2_1(); Documento
    ActiveDocument { [...] obter; }

}
```

Omiti os GUIDs e os atributos de propriedade aqui apenas por questão de espaço, mas você sempre pode usar o Reflector para observar os tipos incorporados. Estas são apenas interfaces e enums – não há implementação. Enquanto uma PIA normal tem uma CoClass representando a implementação real (mas fazendo proxy de tudo para o tipo COM real , é claro), quando o compilador precisa criar uma instância de um tipo COM por meio de uma PIA vinculada , ele cria a instância usando o GUID associado ao tipo. Por exemplo, a linha no exemplo do Word que cria uma instância de Application é traduzida neste código quando a vinculação está habilitada:<sup>7</sup>

```
Aplicativo aplicativo = (Aplicativo) Activator.CreateInstance(
    Type.GetTypeFromCLSID (novo Guid("...")));
```

A Figura 13.5 mostra como isso funciona em tempo de execução.



<sup>7</sup> Bem, quase. O inicializador de objeto torna tudo um pouco mais complicado, porque o compilador usa uma variável temporária extra.

Existem vários benefícios em incorporar bibliotecas de tipos:

ÿ A implantação é mais fácil: o PIA original não é necessário, então você não precisa confiar na versão correta já presente ou ter que enviar o PIA você mesmo. ÿ O versionamento é mais simples: contanto que você use apenas membros da versão da biblioteca COM que está *realmente* instalada, não importa se você compila em uma PIA anterior ou posterior.

ÿ As variantes são tratadas como tipos dinâmicos, reduzindo a quantidade de conversão necessária.

Não se preocupe com o último ponto por enquanto – preciso explicar a digitação dinâmica antes que ela faça sentido. Tudo será revelado no próximo capítulo.

Como você pode ver, a Microsoft realmente levou a sério a interoperabilidade COM para C# 4, tornando todo o processo de desenvolvimento menos doloroso. É claro que o grau de dificuldade sempre foi variável dependendo da biblioteca COM em que você está desenvolvendo – alguns se beneficiarão mais do que outros com os novos recursos.

O próximo recurso é totalmente separado do COM, argumentos nomeados e parâmetros opcionais, mas novamente facilita um pouco o desenvolvimento.

### 13.3 Variação genérica para interfaces e delegados

Você deve se lembrar que no capítulo 3 mencionei que o CLR tinha algum suporte para variação em tipos genéricos, mas que o C# ainda não havia exposto esse suporte. Isso mudou com o C# 4. O C# ganhou a sintaxe necessária para declarar a variação genérica, e o compilador agora sabe sobre as possíveis conversões para interfaces e delegados.

Este não é um recurso de mudança de vida - é mais um caso de atenuar alguns redutores de velocidade que você pode ter atingido ocasionalmente. Nem mesmo remove todos os inchaços; existem várias limitações, principalmente em nome de manter os genéricos absolutamente seguros. Mas ainda é um recurso interessante para se ter na manga.

Caso você precise de um lembrete sobre o que é a variância, vamos começar com um recapitulação das duas formas básicas em que ele se apresenta.

#### 13.3.1 Tipos de variância: covariância e contravariância

Em essência, variância é ser capaz de usar um objeto de um tipo como se fosse outro, de maneira segura. Você está acostumado com a variação em termos de herança normal: se um método tem um tipo de retorno declarado Stream, você pode retornar um MemoryStream da implementação, por exemplo. A variação genérica é o mesmo conceito, mas aplicada aos genéricos, onde se torna um pouco mais complicado. A variação é aplicada aos parâmetros de tipo nas interfaces e tipos delegados. Essa é a parte que você precisa concentrar-se em.

Em última análise, não importa se você se lembra da terminologia que usarei nesta seção. Será útil enquanto você estiver lendo o capítulo, mas é improvável que você precise dele durante uma conversa. Os conceitos são muito mais importantes.

Existem dois tipos de variância: *covariância* e *contravariância*. Eles são essencialmente a mesma ideia, mas são usados no contexto de valores que se movem em direções diferentes.

Começaremos com a covariância, que geralmente é mais fácil de entender.

**COVARIÂNCIA: VALORES QUE SAEM DE UMA API**

*Covariância* trata de valores retornados de uma operação ao chamador.

Vamos imaginar uma interface genérica muito simples representando o padrão de fábrica. Possui um único método, `CreateInstance`, que retornará uma instância do tipo apropriado. Aqui está o código:

```
interface IFactory<T> {
    T CriarInstância();
}
```

Agora, T ocorre apenas uma vez na interface (além do nome). É usado apenas como valor *de retorno* – é a saída do método. Isso significa que faz sentido poder tratar uma fábrica de um tipo específico como uma fábrica de um tipo mais geral. Em termos reais, você pode pensar em uma fábrica de pizza como uma fábrica de alimentos.

**CONTRAVARIÂNCIA: VALORES QUE ENTRAM EM UMA API**

*A contravariância* é o oposto. Trata-se de valores sendo passados *para a API* pelo chamador: a API está consumindo os valores em vez de produzi-los. Vamos imaginar outra interface simples – uma que possa imprimir um tipo de documento específico no console. Novamente, há apenas um método, desta vez chamado `Print`:

```
interface IPrettyPrinter<T> {
    impressão nula (documento T);
}
```

Desta vez T ocorre apenas nas posições *de entrada* da interface, como parâmetro. Para colocar isso em termos concretos novamente, se você tivesse uma implementação de `IPrettyPrinter <SourceCode>`, deveria ser capaz de usá-lo como um `IPrettyPrinter<CSharpCode>`.

**INVARIÂNCIA: VALORES EM AMBOS OS CAMINHOS**

Se a covariância se aplica quando os valores saem apenas *de* uma API, e a contravariância se aplica quando os valores vão apenas *para a* API, o que acontece quando um valor vai nos dois sentidos? Resumindo: nada. Esse tipo seria *invariante*.

Aqui está uma interface que representa um tipo que pode serializar e desserializar um tipo de dados:

```
interface IStorage<T> {
    byte[] Serialize(valor T);
    T Deserialize(byte[] dados);
}
```

Desta vez, se você tiver uma instância de `IStorage<T>` para um tipo T específico, não poderá tratá-la como uma implementação da interface para um tipo mais ou menos específico. Se você tentar usá-lo de maneira covariante (por exemplo, usando um `IStorage<Customer>` como `IStorage<Person>`), poderá fazer uma chamada para `Serialize` com um objeto que ele não pode manipular. Da mesma forma, se você tentar usá-lo de maneira contravariante, poderá obter um tipo inesperado ao desserializar alguns dados.

Se ajudar, você pode pensar na invariância como parâmetros de referência ; para passar uma variável por referência, ela tem que ser *exatamente* do mesmo tipo que o próprio parâmetro, porque o valor entra no método e efetivamente sai novamente também.

### 13.3.2 Usando variação em interfaces

C# 4 permite especificar na declaração de uma interface genérica ou delegado que um parâmetro de tipo pode ser usado de forma covariante usando o modificador `out` ou, de forma contrária, usando o modificador `in`. Depois que o tipo for declarado, os tipos relevantes de conversão estarão disponíveis implicitamente. Isso funciona exatamente da mesma maneira nas interfaces e nos delegados, mas vou mostrá-los separadamente para maior clareza. Vamos começar com as interfaces, pois elas podem ser um pouco mais familiares e já as usamos para descrever a variação.

**CONVERSÕES DE VARIANTES SÃO CONVERSÕES DE REFERÊNCIA** Qualquer conversão que use variância ou covariância é uma *conversão de referência*, o que significa que a mesma referência é retornada após a conversão. Não cria um novo objeto; apenas trata a referência existente como se correspondesse ao tipo de destino. Isso é o mesmo que converter entre tipos de referência em uma hierarquia: se você converter um `Stream` em `MemoryStream` (ou usar a conversão implícita de outra maneira), ainda haverá apenas um objeto. A natureza dessas conversões introduz algumas limitações, como você verá mais adiante, mas significa que elas são eficientes e facilitam a compreensão do comportamento em termos de identidade do objeto.

Desta vez usaremos interfaces familiares para demonstrar as ideias, com alguns tipos simples definidos pelo usuário para os argumentos de tipo.

#### EXPRESSANDO VARIÂNCIA COM IN E OUT

Existem duas interfaces que demonstram a variação de maneira particularmente eficaz: `IEnumerable<T>` é covariante em `T` e `IComparer<T>` é contravariante em `T`. Aqui estão suas novas declarações de tipo no .NET 4:

```
interface pública IEnumerable<out T> interface pública  
IComparer<in T>
```

É fácil lembrar: se um parâmetro de tipo for usado apenas para saída, você poderá usar `out`; se for usado apenas para entrada, você poderá usar `in`. O compilador não sabe se você consegue lembrar qual forma é chamada de covariância e qual é chamada de contravariância!

Infelizmente, a estrutura não contém muitas hierarquias de herança que me ajudariam a demonstrar a variância de maneira particularmente clara; portanto, voltarei ao exemplo padrão de formas orientadas a objetos. O código-fonte para download inclui as definições de `IShape`, `Circle` e `Square`, que são bastante óbvias. A interface expõe propriedades para a caixa delimitadora da forma e sua área. Usarei muito duas listas nos exemplos a seguir, então mostrarei o código de construção delas apenas para referência:

```

Lista<Círculo> círculos = new Lista<Círculo>
{
    novo Círculo(novo Ponto(0, 0), 15), novo Círculo(novo
    Ponto(10, 5), 20),
};

Lista<Quadrado> quadrados = new Lista<Quadrado> {

    novo Quadrado(novo Ponto(5, 10), 5), novo
    Quadrado(novo Ponto(-10, 0), 2)
};

```

O único ponto importante diz respeito aos tipos das variáveis — elas são declaradas como `List<Circle>` e `List<Square>` em vez de `List<IShape>`. Muitas vezes, isso pode ser útil: se você acessar a lista de círculos em outro lugar, talvez queira acessar membros específicos do círculo sem precisar transmitir, por exemplo. Os valores reais envolvidos no código de construção são totalmente irrelevantes; Usarei os nomes círculos e quadrados em outros lugares para me referir às mesmas listas, mas sem duplicar o código.<sup>8</sup>

#### USANDO COVARIÂNCIA DE INTERFACE

Para demonstrar a covariância, tentaremos construir uma lista de formas a partir de uma lista de círculos e de uma lista de quadrados. A listagem a seguir mostra duas abordagens diferentes, nenhuma das quais funcionaria em C# 3.

#### Listagem 13.12 Usando variância para construir uma lista de formas gerais a partir de listas específicas

```

List<IShape> formasByAdding = new List<IShape>();
formasByAdding.AddRange(círculos);
formasByAdding.AddRange(quadrados);

List<IShape> formasByConcat = círculos.Concat<IShape>(quadrados).ToList();

```

 **B** Adiciona listas diretamente

**C** Usa LINQ para concatenação

Efetivamente, a Listagem 13.12 mostra covariância em quatro lugares, cada um convertendo uma sequência de círculos ou quadrados em uma sequência de formas gerais, no que diz respeito ao sistema de tipos. Primeiro você cria um novo `List<IShape>` e chama `AddRange` para adicionar as listas circulares e quadradas a ele B. (Você poderia ter passado um deles para o construtor e depois chamado `AddRange` uma vez.) O parâmetro para `List<T>.AddRange` é do tipo `IEnumerable<T>`, portanto, neste caso, você está tratando cada lista como um `IEnumerable<IShape>` — algo que não seria possível antes. `AddRange` poderia ter sido escrito como um método genérico com seu próprio parâmetro de tipo, mas não foi — fazer isso tornaria algumas otimizações difíceis ou impossíveis.

Outra maneira de criar uma lista que contém os dados em duas sequências existentes é usar LINQ C. Você não pode chamar diretamente `Circles.Concat(squares)`, pois isso confundiria o mecanismo de inferência de tipo, mas especificando explicitamente o argumento de tipo, tudo está bem. Tanto os círculos quanto os quadrados são convertidos implicitamente em `IEnumerable<IShape>` via covariância. Esta conversão não está realmente alterando o valor - apenas como o compilador

<sup>8</sup> Na solução de código-fonte completo, eles são expostos como propriedades na classe estática `Shapes`, mas na versão de snippets inclui o código de construção onde for necessário, para que você possa ajustá-lo facilmente se desejar.

trata o valor. Não se trata de construir uma cópia separada, que é o ponto importante. A covariância é particularmente importante no LINQ to Objects, porque grande parte da API é expressa em termos de `IEnumerable<T>` — a contravariância não é tão importante, porque menos tipos envolvidos são contravariantes.

No C# 3 certamente haveria outras maneiras de abordar o mesmo problema. Você poderia ter criado instâncias de `List<IShape>` em vez de `List<Circle>` e `List<Square>` para as formas originais; você poderia ter usado o operador LINQ `Cast` para converter as listas específicas em listas mais gerais; você poderia ter escrito sua própria classe de lista com um método `AddRange` genérico. Mas nada disso teria sido tão conveniente ou eficiente quanto as alternativas oferecidas aqui.

#### USANDO CONTRAVARIÂNCIA DE INTERFACE

Usaremos os mesmos tipos de forma para demonstrar a contravariância. Desta vez usaremos apenas a lista de círculos, mas um comparador que é capaz de comparar duas formas *qualsquer* apenas comparando as áreas. Não podíamos fazer isso antes do C# 4 porque um `IComparer<IShape>` não podia ser usado como um `IComparer<Circle>`, mas a listagem a seguir mostra a contravariância vindo em socorro.

##### Listagem 13.13 Classificando círculos usando um comparador e contravariância de uso geral

```
class AreaComparer: IComparer<IShape> {
    public int Comparar(ISforma x, IForma y) {
        return x.Area.CompareTo(y.Area);
    }
}
...
IComparer<IShape> areaComparer = new AreaComparer(); círculos.Sort(areaComparer);
```

Não há nada complicado aqui. A classe `AreaComparer` **B** é tão simples quanto uma implementação de `IComparer<T>` pode ser; não precisa de nenhum estado, por exemplo. Normalmente haveria algum tratamento nulo no método `Compare`, mas isso não é necessário para demonstrar a variação.

Depois de ter um `IComparer<IShape>`, você o usa para classificar uma lista de círculos **C**. O argumento para `circles.Sort` precisa ser um `IComparer<Circle>`, mas a contravariância permite converter seu comparador implicitamente. É simples assim.

**SURPRESA, SURPRESA** Se alguém lhe apresentasse esse código como se fosse C# 3, você poderia ter olhado para ele e esperado que funcionasse. Parece óbvio que *deveria* poder funcionar, e este é um sentimento comum; a invariância em C# 2 e 3 geralmente é uma surpresa indesejável. As novas habilidades do C# 4 nesta área não estão introduzindo novos conceitos nos quais você nunca teria pensado antes; eles apenas permitem mais flexibilidade.

Ambos foram exemplos simples usando interfaces de método único, mas os mesmos princípios se aplicam a APIs mais complexas. Claro, quanto mais complexa for a interface,

maior será a probabilidade de um parâmetro de tipo ser usado tanto para entrada quanto para saída, o que o tornaria invariável. Voltaremos a alguns exemplos complicados mais tarde, mas primeiro vamos olhe para os delegados.

### 13.3.3 Usando variação em delegados

Agora que você viu como usar a variação com interfaces, é fácil aplicar o mesmo conhecimento aos delegados. Usaremos alguns tipos familiares novamente:

```
delegado T Func<out T>()
delegar ação nula<in T>(T obj)
```

Elas são realmente equivalentes às interfaces `IFactory<T>` e `IPrettyPrinter<T>` que começou com. Usando expressões lambda, podemos demonstrar ambos facilmente, e até mesmo acorrentar os dois. A listagem a seguir mostra um exemplo usando o tipos de forma.

#### Listagem 13.14 Usando variância com delegados `Func<T>` e `Action<T>` simples

```
Func<Quadrado> squareFactory = () => new Square(new Point(5, 5), 10);
Func<IShape> shapeFactory = squareFactory;
```

 **Converte `Func<T>`**  
**B** usando covariância

```
Action<IShape> shapePrinter = shape => Console.WriteLine(shape.Area);
Action<Quadrado> squarePrinter = shapePrinter;
```

 **Converte ação<T>**  
**C** usando contravariância

```
impressoraquadrado(quadradoFactory());
shapePrinter(shapeFactory());
```

 Verificação de sanidade...

Esperamos que agora o código precise de pouca explicação. A fábrica de quadrados sempre produz um quadrado na mesma posição, com lados de comprimento 10. A covariância permite que você trate uma fábrica quadrada como uma fábrica de formas gerais sem problemas B. Você então cria uma ação de propósito geral que imprime a área de qualquer forma dada a ela. Desta vez você usa uma conversão contravariante para tratar a ação como algo que pode ser aplicado a qualquer quadrado C. Finalmente, você alimenta a ação do quadrado com o resultado de chamar o quadrado fábrica e a ação de forma com o resultado de chamar a fábrica de formas. Ambos imprimem 100, como seria de esperar.

Claro, você usou delegados com um único parâmetro de tipo aqui. O que acontece se você usar delegados ou interfaces com vários parâmetros de tipo? A respeito argumentos de tipo que são tipos delegados genéricos? Bem, tudo pode ficar bastante complicado.

### 13.3.4 Situações complexas

Antes de tentar fazer sua cabeça girar, devo lhe dar um pouco de conforto. Embora nós vamos fizer algumas coisas estranhas e maravilhosas nesta seção, o compilador irá impedi-lo de cometendo erros. Você ainda pode ficar confuso com as mensagens de erro se tiver usado vários parâmetros de tipo de maneiras estranhas, mas depois de compilar, você deve estar

seguro. A complexidade é possível nas formas de variação delegada e de interface, embora a versão delegada seja geralmente mais concisa para trabalhar. Vamos começar com um exemplo relativamente simples.

### COVARIÂNCIA E CONTRAVARIÂNCIA SIMULTÂNEAS COM CONVERTER<TINPUT,

**TOUTPUT>** O tipo de delegado Converter<TInput, TOutput> existe desde o .NET 2.0. É efetivamente Func<T, TResult>, mas com um propósito esperado mais claro. No .NET 4, isso se torna Converter<in TInput, out TOutput>, que mostra qual parâmetro de tipo possui qual tipo de variação.

A listagem a seguir mostra algumas combinações de variância usando um conversor simples.

#### Listagem 13.15 Demonstrando covariância e contravariância com um único tipo

Converte strings em objetos



```
Converter<objeto, string> conversor = x => x.ToString(); Converter<string, string>
contravariância = conversor; Conversor<objeto, objeto> covariância = conversor;
Conversor<string, objeto> ambos = conversor;
```

← Converte  
objetos para  
Cordas B

A Listagem 13.15 mostra as conversões de variância disponíveis em um delegado do tipo Converter<objeto, string> — um delegado que pega qualquer objeto e produz uma string.

Primeiro, você implementa o delegado usando uma expressão lambda simples que chama ToString. B. Na verdade, você nunca *chama* o delegado, então poderia ter usado uma referência nula, mas acho mais fácil pensar na variação se você puder definir uma ação concreta que *aconteceria* se você a chamasse.

As próximas duas linhas são relativamente diretas, desde que você se concentre apenas em um parâmetro de tipo por vez. O parâmetro do tipo TInput é usado apenas em uma posição de entrada, então faz sentido que você possa usá-lo de forma contrária, usando um Converter<objeto, string> como um Converter<Button, string>. Em outras palavras, se você puder passar *qualquer* referência de objeto para o conversor, certamente poderá entregar a ele uma referência de Button. Da mesma forma, o parâmetro de tipo TOutput é usado apenas em uma posição de saída (o tipo de retorno), portanto faz sentido usá-lo de forma covariante; se o conversor sempre retornar uma referência de string, você poderá usá-lo com segurança onde precisar apenas garantir que ele retornará uma referência de objeto.

A linha final é apenas uma extensão lógica dessa ideia C. Ela usa contravariância e covariância na mesma conversão para terminar com um conversor que aceita apenas botões e apenas declara que retornará uma referência de objeto. Observe que você *não pode* converter isso de volta para o tipo de conversão original sem uma conversão — você basicamente relaxou as garantias em todos os pontos e não pode restringi-las novamente implicitamente.

Vamos aumentar um pouco a aposta e ver como as coisas podem ficar complexas se você se esforçar o suficiente.

#### INSANIDADE DE FUNÇÃO DE ORDEM SUPERIOR

Coisas realmente estranhas começam a acontecer quando você combina tipos de variantes. Não entrarei em muitos detalhes aqui — só quero que você aprecie o potencial de complexidade.

Vejamos quatro declarações de delegado:

```
delegar Func<T> FuncFunc<out T>(); delegar void
ActionAction<out T>(Action<T> action); delegar void ActionFunc<in T>(Func<T>
função); delegar Action<T> FuncAction<in T>();
```

Cada uma dessas declarações equivale a aninhar um dos delegados padrão dentro de outro. Por exemplo, FuncAction<T> é equivalente a Func<Action<T>>. Ambos representam uma função que retornará uma Action que pode receber um T. Mas isso deveria ser covariante ou contravariante? Bem, a função retornará algo relacionado a T, então parece covariante, mas esse algo então *recebe* um T, então soa contravariante. A resposta é que o delegado é contravariante em T, e é por isso que é declarado com o modificador in .

Como regra geral, você pode pensar na contravariância aninhada como uma reversão da variância anterior, enquanto a covariância não o faz, portanto, enquanto Action<Action<T>> é covariante em T, Action<Action<Action<T>>> é contravariante. Compare isso com a variação Func<T>, onde você pode escrever Func<Func<Func<...Func<T>...>>> com quantos níveis de aninhamento desejar e ainda obter covariância.

Só para dar um exemplo semelhante usando interfaces, imagine que você tem algo que pode comparar sequências. Se puder comparar duas sequências de objetos arbitrários, certamente poderá comparar duas sequências de strings, mas não vice-versa. Convertendo isso em código (sem implementar a interface!), você pode ver isso da seguinte maneira:

```
IComparer<IEnumerable<objeto>> objetosComparer = ...;
IComparer<IEnumerable<string>> stringsComparer = objectComparer;
```

Esta conversão é legal: I Enumerable<string> é um tipo “menor” que I Enumerable <object> devido à covariância de I Enumerable<T>. A contravariância de I Comparer <T> permite então a conversão de um comparador de tipo “maior” para um comparador de tipo menor.

É claro que nesta seção analisamos apenas delegados e interfaces com um único parâmetro de tipo — tudo isso também pode ser aplicado a vários parâmetros de tipo. Mas não se preocupe: é improvável que você precise desse tipo de variação arrasadora com muita frequência e, quando precisar, terá o compilador para ajudá-lo. Eu realmente só queria alertá-lo sobre as possibilidades.

Por outro lado, há algumas coisas que você espera poder fazer, mas que não são suportadas.

### 13.3.5 Restrições e notas

O suporte à variação fornecido pelo C# 4 é principalmente limitado pelo que é fornecido pelo CLR. Seria difícil para a linguagem suportar conversões proibidas pela plataforma subjacente. Isso pode levar a algumas surpresas.

#### SEM VARIAÇÃO PARA PARÂMETROS DE TIPO NAS CLASSES

Somente interfaces e delegados podem ter parâmetros de tipo variante. Mesmo se você tiver um classe que usa apenas o parâmetro type para entrada (ou apenas para saída), você não pode especifique os modificadores in ou out . Por exemplo, Comparer<T>, a implementação comum de IComparer<T>, é invariável – não há conversão de Comparer<IShape> para Comparador<Círculo>.

Além de quaisquer dificuldades de implementação que isso possa ter causado, eu diria que faz certo sentido conceitualmente. As interfaces representam uma maneira de ver um objeto de uma perspectiva particular, enquanto as classes estão mais enraizadas no tipo *real* do objeto . Este argumento é um pouco enfraquecido pela herança, permitindo que você tratar um objeto como uma instância de qualquer uma das classes em sua hierarquia de herança, é certo. De qualquer forma, o CLR não permite isso.

#### VARIÂNCIA APENAS SUPORTA CONVERSÕES DE REFERÊNCIA

Você não pode usar variação entre dois argumentos de tipo arbitrário só porque há um conversão entre eles. Tem que ser uma *conversão de referência*. Basicamente, isso limita conversões que operam em tipos de referência e que não afetam a representação binária da referência. Isso ocorre para que o CLR possa saber que as operações serão seguras para o tipo, sem a necessidade de injetar nenhum código de conversão real em qualquer lugar. Como mencionei em seção 13.3.2, as conversões de variantes são conversões de referência, portanto, não haveria lugar para o código extra ir de qualquer maneira.

Em particular, esta restrição proíbe quaisquer conversões de tipos de valor e conversões definidas. Por exemplo, as seguintes conversões são todas inválidas:

- ÿ I Enumerable<int> para I Enumerable<object> —Conversão de boxe
- ÿ I Enumerable<short> para I Enumerable<int> —Conversão de tipo de valor
- ÿ I Enumerable<string> para I Enumerable<XName>—Conversão definida pelo usuário

As conversões definidas pelo usuário provavelmente não serão um problema, pois são relativamente raras, mas você pode achar a restrição em torno dos tipos de valor uma dor.

#### PARÂMETROS DE SAÍDA NÃO SÃO POSIÇÕES DE SAÍDA

Este foi uma surpresa para mim, embora faça sentido em retrospecto. Considere um delegado com a seguinte definição:

delegado bool TryParser<T>(entrada de string, valor de saída T)

Você poderia esperar que pudesse fazer T covariante - afinal, ele só é usado em uma posição de saída... ou não?

O CLR realmente não conhece nossos parâmetros . No que diz respeito, eles são apenas refira parâmetros com um atributo [Out] aplicado a eles. C# anexa especial significado para o atributo em termos de atribuição definida, mas o CLR não. referência parâmetros significam dados indo em ambos os sentidos, então se você tiver um parâmetro ref do tipo T, isso significa que T é invariante.

Na verdade, mesmo que o CLR suportasse parâmetros nativamente, ainda assim não seria seguro, porque pode ser usado em uma posição de entrada dentro do próprio método ; depois de escrever na variável, você também poderá lê-la. Não haveria problema se os parâmetros de saída fossem tratados como “valor de cópia no momento do retorno”, mas isso essencialmente cria um alias para o argumento e o parâmetro, o que causaria problemas se eles não fossem exatamente do mesmo tipo. É um pouco complicado de demonstrar, mas há um exemplo no site do livro.

Delegados e interfaces que usam parâmetros out são raros, então isso pode nunca afetar você de qualquer maneira, mas vale a pena saber, por precaução.

#### A VARIÂNCIA DEVE SER EXPLÍCITA

Quando apresentei a sintaxe para expressar variação — aplicando os modificadores in ou out a parâmetros de tipo — você deve ter se perguntado por que precisava se preocupar. O compilador é capaz de *verificar* se qualquer variação que você tenta aplicar é válida, então por que ele não a aplica automaticamente?

Poderia *fazer* isso – pelo menos em muitos casos – mas estou feliz que isso não aconteça. Normalmente você pode adicionar métodos a uma interface e afetar apenas as implementações, e não os chamadores. Mas se você declarou que um parâmetro de tipo é variante e deseja adicionar um método que quebre essa variação, todos os *chamadores* também serão afetados. Posso ver que isso está causando muita confusão. A variação requer alguma reflexão sobre o que você pode querer fazer no futuro, e forçar os desenvolvedores a incluir explicitamente o modificador os incentiva a planejar cuidadosamente antes de se comprometerem com a variação.

Há menos argumentos a favor desta natureza explícita quando se trata de delegados; qualquer alteração na assinatura que afetasse a variação provavelmente interromperia os usos existentes de qualquer maneira. Mas há muito a ser dito sobre consistência – seria estranho se você tivesse que especificar a variação nas interfaces, mas não nas declarações de delegação.

#### CUIDADO COM ALTERAÇÕES QUEBRADAS

Sempre que novas conversões ficam disponíveis, existe o risco de quebra do seu código atual. Por exemplo, se você confiar nos resultados dos operadores is ou as que *não* permitem variação, seu código se comportará de maneira diferente ao ser executado no .NET 4. Da mesma forma, há casos em que a resolução de sobrecarga escolherá um método diferente devido a

sendo opções mais aplicáveis agora. Este é outro motivo para a variação ser especificada explicitamente: ela reduz o risco de quebrar seu código.

Estas situações devem ser bastante raras e o benefício da variação é mais significativo do que as potenciais desvantagens. Você *tem* testes de unidade para detectar mudanças sutis, certo? Falando sério, a equipe C# leva muito a sério a quebra de código, mas às vezes não há como introduzir um novo recurso sem quebrar o código.

#### DELEGADOS MULTICAST E VARIÂNCIA NÃO SE MISTURAM

Normalmente, os genéricos garantem que, a menos que você tenha conversões envolvidas, você não terá problemas de segurança de tipo em tempo de execução. Infelizmente, há uma situação desagradável com tipos de delegados variantes quando se trata de combiná-los. Isso é melhor demonstrado no código:

```
Func<string> stringFunc = () => "";
Func<objeto> objectFunc = () => new object();
Func<objeto> combinado = objectFunc + stringFunc;
```

Isso é compilado sem problemas porque há uma conversão de referência covariante de uma expressão do tipo `Func<string>` para `Func<object>`. Mas o objeto em si ainda é um `Func<string>`, e o método `Delegate.Combine` que faz o trabalho exige que seus argumentos sejam do mesmo tipo — caso contrário, ele não sabe que tipo de delegado deve criar. O código anterior lançará uma `ArgumentException` no momento da execução.

Esse problema foi encontrado relativamente tarde no ciclo de lançamento do .NET 4, mas a Microsoft está ciente disso e há esperança de que ele possa ser corrigido na maioria dos casos em uma versão futura (não foi corrigido no .NET 4.5). Até então, há uma solução alternativa: você pode criar um novo objeto delegado do tipo correto com base na variante e combiná-lo com outro delegado do mesmo tipo. Por exemplo, você pode modificar ligeiramente o código anterior para fazê-lo funcionar:

```
Func<string> stringFunc = () => "";
Func<objeto> defensivoCopy = new Func<objeto>(stringFunc);
Func<objeto> objectFunc = () => new object();
Func<objeto> combinado = objectFunc + defensivoCopy;
```

Felizmente, isso raramente é um problema, na minha experiência.

#### SEM VARIAÇÃO ESPECIFICADA PELO CHAMADOR OU PARCIAL

Isso é realmente uma questão de interesse e comparação, e não qualquer outra coisa, mas vale a pena notar que a variação do C# é *muito* diferente do sistema Java. A variação genérica do Java consegue ser extremamente flexível ao abordá-la do outro lado: em vez de o próprio tipo declarar a variação, o código que *usa* o tipo pode expressar a variação necessária.

**QUER SABER MAIS?** Este livro não é sobre genéricos de Java, mas se este pequeno teaser despertou seu interesse, você pode conferir as “FAQs sobre genéricos de Java” de Angelika Langer (<http://mng.bz/3qgO>). Esteja avisado: é um tema enorme e complexo!

Por exemplo, a interface `List<T>` em Java é aproximadamente equivalente a `IList<T>` em C#. Ele contém métodos para adicionar itens e buscá-los, então claramente em C# é invariável, mas em Java você pode decorar o tipo no código de chamada para explicar qual variação você deseja. O compilador então impede você de usar os membros que vão contra essa variação. Por exemplo, o seguinte código seria perfeitamente válido:

```
List<Forma> formas1 = new ArrayList<Forma>(); Lista<? super Quadrado> quadrados = formas1; quadrados.add(novo Quadrado(10, 10, 20, 20));
```

← Declaração usando contravariância

```
List<Círculo> círculos = new ArrayList<Círculo>(); círculos.add(novo Círculo(10, 10, 20)); Lista<? estende Forma> formas2 = círculos; Forma forma = formas2.get(0);
```

← Declaração usando covariância

Na maioria das vezes, prefiro genéricos em C# a Java, e o apagamento de tipo em particular pode ser um problema em muitos casos. Mas acho esse tratamento da variância realmente interessante. Não espero ver nada semelhante em versões futuras do C#, então pense cuidadosamente sobre como você pode dividir suas interfaces para permitir flexibilidade, mas sem introduzir mais complexidade do que é realmente garantido.

Pouco antes de encerrar o capítulo, há duas mudanças quase triviais a serem abordadas: como o compilador C# lida com instruções de bloqueio e eventos semelhantes a campos.

### 13.4 Pequenas mudanças em eventos de bloqueio e semelhantes a campos

Não quero fazer muitas dessas mudanças; é provável que eles nunca afetem você.

Mas se você estiver olhando para um código compilado e se perguntando por que ele tem essa aparência, é útil saber o que está acontecendo.

#### 13.4.1 Bloqueio robusto

Vamos considerar um trecho simples de código C# que usa um bloqueio. Os detalhes do que acontece dentro do bloco não são importantes, mas inclui uma única declaração apenas para maior clareza:

```
bloquear (listaLock) {
    lista.Add("item");
}
```

Antes do C# 4 - e incluindo o C# 4 se você estiver visando algo anterior ao .NET 4 - isso seria efetivamente compilado neste código:

```
objeto tmp = listLock;
Monitor.Enter(tmp); tentar {
    lista.Add("item");

} finalmente
{
    Monitor.Exit(tmp);
}
```

**Referência de cópias**

Adquire o bloqueio B para travar antes de tentar

As versões bloqueiam tudo o que Add faz

Isso é *quase* aceitável – em particular, evita alguns problemas. Você deseja ter certeza de liberar o mesmo monitor adquirido, então primeiro copie a referência em uma variável local temporária B. Isso também significa que a expressão de bloqueio é avaliada apenas uma vez. Em seguida, você adquire o bloqueio *antes* do bloco try. Isso ocorre para que você não tente liberar o bloqueio no bloco final se o thread for abortado sem adquiri-lo com sucesso. Isso leva a um problema diferente: agora, se o thread for abortado *após* o bloqueio ser adquirido, mas *antes* de você entrar no bloco try, você não terá liberado o bloqueio. Isso poderia levar a um impasse – outro thread poderia estar esperando eternamente que este liberasse o bloqueio. Embora o CLR tenha historicamente tentado impedir que isso acontecesse, não é totalmente impossível.

O que você quer é alguma forma de adquirir atomicamente o bloqueio e saber que ele foi adquirido. Felizmente, isso é exposto no .NET 4 por meio de uma nova sobrecarga no Monitor.Enter, que o compilador C# 4 usa desta forma:

```
bool adquirido = false; objeto tmp =
listLock; tentar {
```

```
    Monitor.Enter(tmp, ref adquirido); lista.Add("item");
```

Adquire bloqueio dentro do bloco try

```
} finalmente
{
    se (adquirido) {
        Monitor.Release(tmp);
    }
}
```

Liberar o bloqueio  
condicionalmente

Agora o bloqueio será liberado se e somente se você o adquiriu com sucesso, de forma consistente.

Deve-se notar que em alguns casos um impasse não é o pior resultado; ocasionalmente é mais perigoso para um aplicativo continuar do que simplesmente parar.<sup>9</sup> Mas seria ridículo *confiar* na condição de deadlock; é melhor evitar abortar threads, se possível. (Abortar o thread em execução no momento é um pouco melhor, pois você tem mais controle - isso é o que Response.Redirect faz no ASP.NET, por exemplo, mas eu ainda sugeriria encontrar formas melhores de controle de fluxo.)

Há um último ajuste a ser feito antes de passarmos para o grande recurso do C# 4.

#### 13.4.2 Mudanças em eventos semelhantes a campos

Há duas mudanças na maneira como os *eventos semelhantes a campos* são implementados no C# 4 que vale a pena mencionar brevemente. É improvável que eles afetem você, embora possam *quebrar* alterações.

Só para recapitular, eventos semelhantes a campos são eventos declarados como se fossem campos, com nenhum bloco de adição/remoção explícito, como este:

```
evento público EventHandler Click;
```

Primeiro, a forma como a segurança do thread é alcançada foi alterada. Antes do C# 4, eventos semelhantes a campos resultavam em código que travava tanto neste (para eventos de instância) quanto no tipo de declaração (para eventos estáticos). A partir do C# 4, o compilador obtém assinatura e cancelamento de assinatura atômica e segura para thread usando Interlocked.CompareExchange<T>. Ao contrário da alteração anterior na instrução lock , isso se aplica mesmo ao direcionar versões anteriores do .NET Framework.

---

<sup>9</sup> Eric Lippert tem uma excelente postagem no blog sobre esse assunto, intitulada “Bloqueios e exceções não se misturam”: <http://mng.bz/Qy7p>.

Segundo, o significado do nome do evento *na classe declarante* mudou. Anteriormente, se você assinasse (ou cancelasse a assinatura) do evento dentro da classe que continha a declaração — como com Click += DefaultClickHandler; — isso iria direto para o campo de apoio, ignorando completamente a implementação de adicionar/remover. Agora isso não acontece; quando você usa += ou -=, o nome do evento se refere ao evento em si, não ao campo de apoio. Quando o nome é usado para qualquer outra finalidade (normalmente atribuição ou invocação), ele ainda se refere diretamente ao campo de apoio.

Ambas são mudanças sensatas que tornam tudo mais organizado, embora você provavelmente não as tenha notado no uso diário. Chris Burrows aborda o assunto detalhadamente em seu blog, se você quiser saber mais (veja <http://mng.bz/Kyr4>).

## 13.5 Resumo

Este foi um capítulo meio escolhido e misturado, com várias áreas distintas. Dito isto, o COM se beneficia muito com argumentos nomeados e parâmetros opcionais, portanto há alguma sobreposição entre eles.

Suspeito que demorará um pouco para que os desenvolvedores de C# aprendam a melhor forma de usar os novos recursos para parâmetros e argumentos. A sobrecarga ainda fornece portabilidade extra para linguagens que não suportam parâmetros opcionais, e argumentos nomeados podem parecer estranhos em algumas situações até que você se acostume com eles. Os benefícios podem ser significativos, como demonstrei com o exemplo da construção de instâncias de tipos imutáveis. Você precisará tomar algum cuidado ao atribuir valores padrão a parâmetros opcionais, mas espero que você ache a sugestão de usar null como um “valor padrão padrão” útil e flexível, que efetivamente evite alguns dos as limitações e armadilhas que você poderia encontrar.

Trabalhar com COM já percorreu um *longo* caminho no C# 4. Ainda prefiro usar soluções puramente gerenciadas onde estiverem disponíveis, mas pelo menos o código que chama o COM é muito mais legível agora, além de ter uma história de implantação melhor. Ainda não analisamos todas as melhorias na interoperabilidade COM , já que os recursos de digitação dinâmica que discutiremos no próximo capítulo também têm impacto no COM , mas mesmo sem levar isso em consideração, vimos um pequeno exemplo torne-se muito mais agradável apenas aplicando alguns passos simples.

O último tópico importante deste capítulo foi a variação genérica agora disponível para interfaces e delegados. Às vezes, você pode acabar usando a variação sem nem mesmo saber, e acho que a maioria dos desenvolvedores está mais propensa a usar a variação declarada nas interfaces e delegados da estrutura, em vez de criar a sua própria. Peço desculpas se ocasionalmente ficou complicado, mas é bom saber o que está por aí. Se servir de consolo para você, o ex-membro da equipe C#, Eric Lippert, reconheceu publicamente em uma postagem de blog (consulte <http://mng.bz/79d8>) que funções de ordem superior fazem até sua cabeça doer, então você está em boa companhia . A postagem de Eric faz parte de uma longa série sobre variância (veja <http://mng.bz/94H3>), que é, acima de tudo, um diálogo sobre a

decisões de projeto envolvidas. Se você ainda não teve variação suficiente, é uma excelente leitura.

Para completar, também demos uma olhada rápida nas mudanças na forma como o compilador C# lida com bloqueios e eventos semelhantes a campos.

Este capítulo tratou de mudanças *relativamente* pequenas em C#. O Capítulo 14 trata de alguns algo muito mais fundamental: a capacidade de usar C# de maneira dinâmica.

# 14

## Vinculação dinâmica em uma linguagem estática

### Este capítulo cobre

- ÿ O que significa ser dinâmico
- ÿ Como usar digitação dinâmica em C# 4
- ÿ Exemplos com COM, Python e reflexão
- ÿ Como a digitação dinâmica é implementada
- ÿ Reagindo dinamicamente

C# sempre foi uma linguagem de tipo estaticamente, sem exceções. Houve algumas áreas em que o compilador procurou nomes específicos em vez de interfaces, como encontrar métodos Add apropriados para inicializadores de coleção, mas não houve nada verdadeiramente dinâmico na linguagem além do polimorfismo normal.

Isso muda com o C# 4 – pelo menos parcialmente. A maneira mais simples de explicar isso é que existe um novo tipo estático chamado dinâmico, com o qual você pode tentar fazer quase tudo em tempo de compilação e deixar o framework resolver isso em tempo de execução. Claro, há mais do que isso, mas esse é o resumo executivo.

Dado que C# ainda é uma linguagem de tipo estaticamente em todos os lugares onde você *não* está usando dinâmica, não espero que os fãs de programação dinâmica se tornem repentinamente C#

defensores. Não foi por isso que o recurso foi introduzido: ele visa principalmente a interoperabilidade. Quando linguagens dinâmicas como IronRuby e IronPython se juntaram ao ecossistema .NET, teria sido uma loucura não poder chamar código C# do IronPython e vice-versa. Da mesma forma, desenvolver em APIs COM costumava ser estranho em C#, com uma abundância de conversões sobrecregendo o código. A digitação dinâmica aborda todas essas preocupações. Por outro lado, existem muitos projetos que usam digitação dinâmica em C# para simplificar os limites de acesso a dados.

Um aviso que repetirei ao longo do capítulo: vale a pena ter cuidado com a digitação dinâmica. É divertido explorar e foi bem implementado, mas ainda assim recomendo que você pense com cuidado antes de usá-lo intensamente. Assim como qualquer outro recurso novo, avalie os prós e os contras, em vez de se apressar só porque é legal (o que sem dúvida é). A estrutura faz um bom trabalho de otimização do código dinâmico, mas será mais lento que o código estático na maioria dos casos. Mais importante, você perde muita segurança em tempo de compilação. Embora o teste de unidade ajude você a encontrar muitos erros que podem surgir quando o compilador não é capaz de ajudá-lo muito, ainda prefiro o feedback imediato do compilador me dizendo se estou tentando usar um método que não existe ou não pode ser chamado com um determinado conjunto de argumentos.

Por outro lado, há situações em que o nível de segurança fornecido pelo compilador não é muito forte para começar. Por exemplo, há muito mais coisas que podem dar errado com o código que usa reflexão do que apenas os erros que um compilador pode detectar. Se você está tentando invocar um método com seu nome, esse método existe? É acessível ao seu código? Você está fornecendo argumentos apropriados? O compilador não pode ajudá-lo com nada disso. O código dinâmico equivalente ainda não consegue detectar esses erros em tempo de compilação, mas pelo menos o código pode ser consideravelmente mais fácil de ler e entender. É tudo uma questão de usar a abordagem mais apropriada para o problema específico no qual você está trabalhando.

O comportamento dinâmico pode ser útil em situações em que você está lidando naturalmente com ambientes ou dados dinâmicos, mas se você realmente deseja escrever grandes partes do seu código dinamicamente, sugiro que você use uma linguagem onde esse seja o estilo normal em vez da exceção. C# ainda é uma linguagem projetada *para* digitação estática; linguagens que foram dinâmicas desde o início geralmente possuem vários recursos para ajudá-lo a trabalhar de forma mais produtiva com comportamento dinâmico. Agora que você pode facilmente chamar essas linguagens a partir do C#, você pode separar as partes do seu código que se beneficiam de um estilo amplamente dinâmico daquelas onde a digitação estática funciona melhor.

Não quero estragar muito as coisas. Onde a digitação dinâmica é útil, ela pode ser muito mais simples que as alternativas. Neste capítulo, veremos as regras básicas de digitação dinâmica em C# 4 e, em seguida, mergulharemos em alguns exemplos: usar COM dinamicamente, chamar algum código IronPython e tornar a reflexão muito mais simples. Você pode fazer tudo isso sem conhecer os detalhes, mas assim que tiver o sabor da digitação dinâmica, veremos o que está acontecendo nos bastidores. Em particular, discutiremos o Dynamic Language Runtime e o que o compilador C# faz quando encontra código dinâmico. Finalmente, você verá como fazer com que seus próprios tipos respondam dinamicamente a chamadas de métodos, acessos a propriedades e coisas do gênero. Mas primeiro, vamos dar um passo atrás.

## 14.1 O quê? Quando? Por que? Como?

Antes de chegarmos a qualquer código que mostre esse novo recurso do C# 4, vale a pena entender melhor por que ele foi introduzido. Não conheço nenhuma outra linguagem que tenha passado de puramente estática a parcialmente dinâmica; isto é um

um passo significativo na evolução do C#, quer você o use com frequência ou apenas ocasionalmente.

Começaremos dando uma nova olhada no que significa *dinâmico* e *estático*, considerando alguns dos principais casos de uso para digitação dinâmica em C# e, em seguida, nos aprofundaremos em como isso funciona implementado em C# 4.

### 14.1.1 O que é digitação dinâmica?

No capítulo 2 expliquei as características de um sistema de tipos e descrevi como o C# anteriormente era uma linguagem de tipo estaticamente. O compilador conhece o tipo de expressões no código e conhece os membros disponíveis em qualquer tipo. Ele aplica um conjunto bastante complexo de regras para determinar qual membro exato deve ser usado e quando. Esse inclui resolução de sobrecarga; a única escolha que resta até mais tarde é escolher a implementação de métodos virtuais dependendo do tipo de tempo de execução do objeto. O processo de descobrir qual membro usar é chamado *de ligação* e, em um método digitado estaticamente linguagem ocorre em tempo de compilação.

Em uma linguagem de tipo dinâmico, toda essa ligação ocorre em tempo de execução. A compilador ou analisador pode verificar se o código está *sintaticamente* correto, mas não pode verificar que os métodos que você chama e as propriedades que você acessa estão realmente presentes. É um pouco como um processador de texto sem dicionário: pode verificar sua pontuação, mas não a sua ortografia, então se você quiser ter algum tipo de confiança em seu código, você realmente precisa de um bom conjunto de testes unitários. Algumas linguagens dinâmicas são sempre interpretadas, sem nenhum compilador envolvido. Outros fornecem um intérprete e um compilador, para permitir o desenvolvimento rápido com um *REPL* – um loop de leitura, avaliação e impressão.

**REPL E C#** Estritamente falando, REPL não está associado apenas a linguagens dinâmicas. Algumas linguagens de tipo estaticamente possuem *intérpretes* que compilam no voar. Notavelmente, o F# vem com uma ferramenta chamada *F# Interactive*, que faz exatamente isso. Mas os intérpretes são muito mais comuns para linguagens dinâmicas do que estáticas. uns.

C# possui ferramentas semelhantes: o avaliador de expressão subjacente ao Watch e janelas imediatas no Visual Studio podem ser consideradas uma forma de REPL, e o Mono possui um Shell C# (consulte [www.mono-project.com/CsharpRepl](http://www.mono-project.com/CsharpRepl)).

É importante notar que os novos recursos dinâmicos do C# 4 *não* incluem a interpretação do C# código-fonte em tempo de execução; não há equivalente direto para a avaliação JavaScript função, por exemplo. Para executar código baseado em dados em strings, você precisa usar a API CodeDOM (e CSharpCodeProvider em particular) ou reflexão simples para invocar membros individuais. O projeto Roslyn é outra opção aqui, embora ainda esteja no Community Technology Preview enquanto escrevo isto.

É claro que o mesmo tipo de trabalho deverá ser feito em *algum* momento, independentemente da abordagem adotada. Ao solicitar ao compilador que trabalhe mais antes da execução, os sistemas estáticos geralmente apresentam melhor desempenho do que os dinâmicos. Dadas as desvantagens que mencionei até agora, você deve estar se perguntando por que alguém iria querer se preocupar com a digitação dinâmica em primeiro lugar.

### 14.1.2 Quando a digitação dinâmica é útil e por quê?

A digitação dinâmica tem dois pontos importantes a seu favor. Primeiro, se você souber o nome de um membro que deseja chamar, os argumentos com os quais deseja chamá-lo e o objeto no qual deseja chamá-lo, isso é tudo que você precisa. De qualquer forma, isso pode parecer toda a informação que você poderia ter, mas o compilador C# normalmente gostaria de saber mais. Crucialmente, para identificar exatamente o membro (substituição do módulo), seria necessário saber o tipo do objeto que você está chamando e os tipos dos argumentos. Às vezes você não conhece esses tipos em tempo de compilação, mesmo sabendo o suficiente para ter certeza de que o membro estará presente e correto quando o código for executado.

Por exemplo, se você sabe que o objeto que está usando tem uma propriedade Length que deseja usar, não importa se é uma String, um StringBuilder, um Array, um Stream ou qualquer outro tipo com essa propriedade. . . Você não precisa que essa propriedade seja definida por alguma classe base ou interface comum, o que pode ser útil se esse tipo não existir. Isso é chamado de *digitação de pato*, a partir da noção de que “se ele anda como um pato e grasna como um pato, eu o chamaria de pato”.<sup>1</sup> Mesmo quando existe um tipo que oferece tudo que você precisa, às vezes pode ser irritante para informar ao compilador exatamente de qual tipo você está falando. Isto é particularmente relevante ao usar APIs do Microsoft Office via COM.

Muitos métodos e propriedades são declarados para retornar apenas VARIANT, o que significa que o código C# que usa essas chamadas geralmente é repleto de conversões. A digitação Duck permite que você omita todas essas conversões, contanto que você esteja confiante sobre o que está fazendo.

A segunda característica importante da digitação dinâmica é a capacidade de um objeto responder a uma chamada analisando o nome e os argumentos fornecidos a ele. Ele pode se comportar como se o membro tivesse sido declarado pelo tipo da maneira normal, mesmo que os nomes dos membros não pudessem ser conhecidos até o tempo de execução. Por exemplo, considere a seguinte chamada:

```
livros.FindByAuthor("Joshua Bloch")
```

Normalmente, isso exigiria que o membro FindByAuthor fosse declarado pelo designer do tipo envolvido. Em uma camada de dados dinâmica, pode haver um único código inteligente para analisar chamadas como essa. Ele pode detectar que há uma propriedade Autor nos dados associados (sejam de um banco de dados, documento XML , dados codificados ou qualquer outra coisa) e agir de acordo.

---

<sup>1</sup> O artigo da Wikipedia sobre digitação de pato traz mais informações sobre a história do termo: [http://en.wikipedia.org/wiki/Duck\\_ttyping](http://en.wikipedia.org/wiki/Duck_ttyping).

Nesse caso, decidiria que você deseja realizar uma consulta usando o argumento especificado como autor. De certa forma, essa é apenas uma maneira mais complexa de escrever algo assim:

```
livros.Find("Autor", "Joshua Bloch")
```

Mas o primeiro trecho parece mais apropriado; o código de chamada conhece a parte do Autor estaticamente, mesmo que o código de recebimento não conheça. Essa abordagem pode ser usada para imitar linguagens específicas de domínio (DSLs) em algumas situações. Também pode ser usado para criar uma API natural para explorar estruturas de dados, como árvores XML .

Outra característica da programação com linguagens dinâmicas *tende* a ser um estilo experimental de programação usando um interpretador apropriado, como mencionei anteriormente.

Isso não é *diretamente* relevante para o C# 4, mas o fato de que o C# 4 pode interoperar ricamente com linguagens dinâmicas executadas no Dynamic Language Runtime (DLR) significa que se você estiver lidando com um problema que se beneficiaria com esse estilo, você ' Serei capaz de usar os resultados diretamente do C# em vez de ter que portá-los para C# posteriormente.

Examinaremos esses cenários com mais profundidade e alguns exemplos concretos, quando discutirmos os fundamentos das habilidades dinâmicas do C# 4. Vale a pena ressaltar brevemente que, se esses benefícios *não se aplicarem* a você, é mais provável que a digitação dinâmica seja um obstáculo do que uma ajuda. Muitos desenvolvedores não precisarão usar muito a digitação dinâmica em sua codificação diária e, mesmo quando *necessária*, pode ser apenas para uma pequena parte do código. Assim como qualquer recurso, ele pode ser usado em demasia. Na minha opinião, geralmente vale a pena pensar cuidadosamente se algum design alternativo permitiria que a digitação estática resolvesse o mesmo problema com elegância. Mas sou tendencioso por ter experiência em linguagens de tipo estaticamente — vale a pena ler livros sobre linguagens de tipo dinâmico, como Python e Ruby, para ver uma variedade maior de benefícios do que os que apresento neste capítulo.

Você provavelmente está ficando ansioso para ver algum código real agora, então vamos apenas dar uma olhada breve visão geral do que está acontecendo e, em seguida, mergulhe em alguns exemplos.

### 14.1.3 Como o C# 4 fornece digitação dinâmica?

C# 4 introduz um novo tipo chamado dinâmico. O compilador trata esse tipo de maneira diferente de qualquer tipo CLR normal.<sup>2</sup> Qualquer expressão que use um valor dinâmico faz com que o compilador mude radicalmente seu comportamento. Em vez de tentar descobrir *exatamente* o que o código significa, vincular o acesso de cada membro de forma adequada, executar a resolução de sobrecarga e assim por diante, o compilador apenas analisa o código-fonte para descobrir que tipo de *operação* você está tentando executar. nome, quais argumentos estão envolvidos e qualquer outra informação relevante. Em vez de emitir IL para executar o código diretamente, o compilador gera código que chama o DLR com todas as informações necessárias. O restante do trabalho é realizado em tempo de execução.

<sup>2</sup> Na verdade, dinâmico não representa um tipo específico de CLR. Na verdade, é apenas System.Object em conjunto com System.Dynamic.DynamicAttribute. Veremos isso com mais detalhes na seção 14.4, mas por enquanto você pode fingir que é um tipo real.

Em muitos aspectos, isso é semelhante aos diferentes tipos de código gerados pelas conversões de expressões lambda. Isso pode resultar em código para executar as ações necessárias (ao converter para um tipo delegado) ou resultar em código que cria uma descrição das ações necessárias (ao converter para uma árvore de expressão). Você verá mais tarde que as árvores de expressão são extremamente importantes no DLR e muitas vezes o compilador C# usará árvores de expressão para descrever o código. (Nos casos mais simples, onde não há nada além de uma invocação de membro, não há necessidade de uma árvore de expressão.)

Quando o DLR vincula a chamada relevante em tempo de execução, ele passa por um processo complicado para determinar o que deve acontecer. Isso não apenas deve levar em consideração as regras normais do C# para sobrecargas de métodos e assim por diante, mas também a possibilidade de que o próprio objeto queira fazer parte da decisão, como você viu no exemplo `FindBy-Author` anteriormente .

A maior parte disso acontece nos bastidores - o código-fonte que você escreve para usar digitar pode ser muito simples.

## 14.2 O guia de cinco minutos para dinâmica

Você se lembra de quantos novos bits de sintaxe estavam envolvidos quando você aprendeu sobre o LINQ? Bem, a digitação dinâmica é exatamente o oposto: há uma única palavra-chave contextual, dinâmica, que você pode usar na maioria dos lugares onde usaria um nome de tipo. Essa é toda a nova sintaxe necessária, e as principais regras sobre dinâmica são facilmente expressas, se você não se importar em acenar com a mão para começar:

- ÿ Existe uma conversão implícita de quase qualquer tipo de CLR para dinâmico.
- ÿ Existe uma conversão implícita de qualquer expressão do tipo dinâmico para quase qualquer tipo CLR .
- ÿ Expressões que usam um valor do tipo dinâmico geralmente são avaliadas dinamicamente.
- ÿ O tipo estático de uma expressão avaliada dinamicamente é geralmente considerado dinâmico.

As regras detalhadas são mais complicadas, como você verá na seção 14.4, mas por enquanto vamos ficar com a versão simplificada.

A listagem a seguir fornece um exemplo completo demonstrando cada ponto.

### Listagem 14.1 Usando dinâmico para iterar em uma lista, concatenando strings

```
itens dinâmicos = new List<string> { "Primeiro", "Segundo", "Terceiro" }; valor dinâmicoToAdd = "!"; foreach
(item dinâmico em itens) {

    string resultado = item + valorToAdd;
    Console.WriteLine(resultado);
}
```

O resultado da listagem 14.1 não deveria ser uma surpresa: ele imprime Primeiro!, Segundo! e Terceiro!. Você poderia facilmente ter especificado os tipos de itens e variáveis valueToAdd explicitamente neste caso, e tudo teria funcionado normalmente.

maneira, mas imagine que as variáveis estão obtendo seus valores de outras fontes de dados em vez de tê-los codificados. O que aconteceria se você quisesse adicionar um número inteiro em vez de uma string?

A próxima listagem é apenas uma ligeira variação. A declaração de `valueToAdd` não foi mudado; apenas a expressão de atribuição.

#### Listagem 14.2 Adicionando números inteiros a strings dinamicamente

```
itens dinâmicos = new List<string> { "Primeiro", "Segundo", "Terceiro" }; valor dinâmicoToAdd = 2; foreach (item dinâmico em itens) {
```

```
    string resultado = item + valorToAdd;
    Console.WriteLine(resultado);
}
```

string +  
concatenação int

Desta vez, o primeiro resultado é `First2`, que é o que você esperaria. Usando digitação estática, você teria que alterar explicitamente a declaração de `valueToAdd` de string para int. O operador de adição ainda está construindo uma string.

E se você alterasse os itens para números inteiros também? Vamos tentar isso simples alteração, conforme mostrado na listagem a seguir.

#### Listagem 14.3 Adicionando inteiros a inteiros

```
itens dinâmicos = new List<int> { 1, 2, 3 }; valor dinâmicoToAdd = 2;
foreach (item dinâmico em itens) {
```

```
    string resultado = item + valorToAdd;
    Console.WriteLine(resultado);
}
```

int + int adição

Desastre! Você ainda está tentando converter o resultado da adição em uma string. As únicas conversões permitidas são as mesmas que estão presentes normalmente no C#, portanto não há conversão de int para string. O resultado é uma exceção (em tempo de execução, é claro):

Exceção não tratada:

```
Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
Não é possível converter implicitamente o tipo 'int' em 'string' em
    CallSite.Target(Closure , em Site de chamada , Objeto)
        System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet]
            (Site CallSite, T0 arg0)
...
```

A menos que você seja perfeito, provavelmente encontrará muito `RuntimeBinderException` quando começar a usar a digitação dinâmica. É a nova `NullReferenceException` em alguns aspectos; você certamente encontrará isso, mas com alguma sorte será no contexto de testes de unidade, e não em relatórios de bugs do cliente. De qualquer forma, você pode consertar alterando o tipo de resultado para dinâmico, para que a conversão não seja necessária.

Pensando bem, por que se preocupar com a variável de resultado em primeiro lugar? Você poderia basta ligar para `Console.WriteLine` imediatamente. A listagem a seguir mostra as alterações.

#### Listagem 14.4 Adicionando inteiros a inteiros – mas sem exceção

```
itens dinâmicos = new List<int> { 1, 2, 3 }; valor dinâmicoToAdd = 2;
foreach (item dinâmico em itens) {
    Console.WriteLine(item + valorToAdd);
}
```

Sobrecarga de  
chamadas com argumento int

Isso imprime 3, 4 e 5, como seria de esperar. Alterar os dados de entrada agora não apenas alteraria o operador escolhido no tempo de execução, mas também alteraria qual sobrecarga de `Console.WriteLine` foi chamada. Com os dados originais, ele chamaria `Console.WriteLine(string)`; com as variáveis atualizadas, chamaria `Console.WriteLine(int)`. Os dados podem até conter uma mistura de valores, fazendo com que a chamada exata mude a cada iteração!

Você também pode usar dinâmico como tipo declarado para campos, parâmetros e retorno tipos. Isso contrasta fortemente com o uso de var, que é restrito a variáveis locais.

**DIFERENÇAS ENTRE VAR E DYNAMIC** Em muitos dos exemplos até agora, quando você realmente conhece os tipos em tempo de compilação, você poderia ter usado var para declarar as variáveis. À primeira vista, os dois recursos parecem muito semelhantes. Em ambos os casos, parece que você está declarando uma variável sem especificar seu tipo, mas ao usar dynamic você está definindo explicitamente o tipo como dinâmico. Você só pode usar var quando o compilador for capaz de inferir *estaticamente o tipo que você quer dizer*, e o sistema de tipos realmente permanece totalmente estático. Claro, se você usar var para uma variável que é inicializada com uma expressão do tipo dinâmico, a variável acaba sendo digitada (estaticamente) para ser dinâmica também. Dada a confusão que isso pode causar, aconselho fortemente contra isso.

O compilador é inteligente quanto às informações que registra, e o código que usa essas informações em tempo de execução também é inteligente: é um mini compilador C# por si só. Ele usa qualquer informação de tipo estático conhecida em tempo de compilação para fazer o código se comportar da maneira mais intuitiva possível.

Além de alguns detalhes sobre o que você *não pode* fazer com a digitação dinâmica, isso é tudo que você realmente precisa saber para começar a usá-la em seu próprio código. Mais tarde voltaremos a essas restrições, bem como aos detalhes do que o compilador está realmente fazendo, mas primeiro vamos ver a digitação dinâmica fazendo algo genuinamente *útil*.

### 14.3 Exemplos de digitação dinâmica

A digitação dinâmica é um pouco como código inseguro ou interoperabilidade com código nativo usando P/Invoke. Muitos desenvolvedores não precisarão dele ou o usarão uma vez na lua azul. Para outros desenvolvedores – especialmente aqueles que lidam com o Microsoft Office – isso proporcionará um enorme aumento de produtividade, seja simplificando o código existente ou permitindo abordagens radicalmente diferentes para seus problemas.

Esta seção não pretende ser exaustiva de forma alguma. Desde a publicação da segunda edição deste livro, vários projetos de código aberto usaram a digitação dinâmica com grande efeito, incluindo Massive (<https://github.com/robconery/massive>), Dapper (<http://code.google.com/p/dapper-dot-net/>) e Json.NET (<http://json.code-plex.com>). Esses exemplos estão todos nos limites dos dados, seja ao se comunicar com um banco de dados ou ao serializar e desserializar JSON. Isso não quer dizer que a digitação dinâmica só seja útil nos limites dos dados, é claro, e detesto prever que novos usos a comunidade poderá criar no futuro.

Veremos três exemplos aqui: trabalhar com Excel, chamar Python e usando tipos .NET gerenciados normais de uma forma mais flexível.

#### 14.3.1 COM em geral e Microsoft Office em particular

Você já viu a maioria dos novos recursos que o C# 4 traz para a interoperabilidade COM, mas houve um que não pudemos abordar no capítulo 13 porque você ainda não tinha visto a digitação dinâmica. Se você optar por incorporar os tipos de interoperabilidade que está usando no assembly (usando a opção do compilador /l ou definindo a propriedade Embed Interop Types como true), qualquer coisa na API que de outra forma seria declarada como objeto será alterada para dinâmico. Isso torna muito mais fácil trabalhar com APIs de tipo um tanto fraco, como as expostas pelo Office. (Embora o modelo de objeto no Office seja razoavelmente forte por si só, muitas propriedades são expostas como *variantes* porque podem lidar com números, cadeias de caracteres, datas e assim por diante.)

Novamente, mostrarei apenas um pequeno exemplo aqui – um que faz ainda menos do que o exemplo da Palavra no capítulo 13. O aspecto dinâmico é fácil de entender neste cenário. Definiremos as primeiras 20 células da linha superior de uma nova planilha do Excel com os números de 1 a 20. A listagem a seguir mostra um trecho de código inicial digitado estaticamente para conseguir isso.

##### Listagem 14.5 Configurando um intervalo de valores com digitação estática

```
var app = novo aplicativo { Visível = true }; app.Workbooks.Add(); Planilha
planilha = (Planilha)
app.ActiveSheet; Início do intervalo = (intervalo) planilha.Células[1, 1]; Fim do
intervalo = (intervalo) planilha.Células[1, 20]; planilha.Range[início,
fim].Valor = Enumerable.Range(1, 20)
```

Abra o Excel com um  
Planilha ativa **B**

**C** Determinar células iniciais e finais

Preencha o intervalo  
**D** com [1, 20]

Este código depende de uma diretiva using para o namespace Microsoft.Office.Interop.Excel (não mostrado aqui), portanto, desta vez o tipo Application se refere ao Excel, não ao Word. Você ainda está usando os novos recursos do C# 4 ao não especificar um argumento para o parâmetro opcional na chamada Workbooks.Add() enquanto configura as coisas **B** e também ao usar um indexador nomeado **C**.

Quando o Excel estiver instalado e funcionando, você calcula as células inicial e final do intervalo geral. Neste caso, ambos estão na mesma linha, mas você poderia ter criado um intervalo retangular selecionando dois cantos opostos. Você *poderia* ter criado o intervalo

em uma única chamada para Range["A1:T1"], mas pessoalmente acho mais fácil trabalhar apenas com números. Nomes de células como B3 são ótimos para humanos, mas mais difíceis de usar em um programa.

Depois de ter o intervalo, você define todos os valores nele definindo a propriedade Value com uma matriz de números inteiros D. Você pode usar uma matriz unidimensional, pois está definindo apenas uma única linha; para definir um intervalo abrangendo várias linhas, você precisaria usar uma matriz retangular.

Tudo isso funciona, mas você teve que usar três conversões em seis linhas de código. O indexador que você chama por meio de Cells e a propriedade ActiveSheet são declarados para retornar o objeto normalmente. (Vários parâmetros *também* são declarados como tipo objeto, mas isso não importa tanto porque há uma conversão implícita de qualquer tipo que não seja ponteiro para objeto - apenas vindo na direção oposta requer a conversão.) Esse código não fecha o Excel em no final da listagem, só para você ver a planilha aberta no final.

Com o assembly de interoperabilidade primário configurado para incorporar os tipos necessários em seu próprio binário, todos esses exemplos se tornam dinâmicos. Com a conversão implícita de dinâmico para outros tipos, é possível remover todas as conversões, conforme mostrado na listagem a seguir.

#### Listagem 14.6 Usando conversões implícitas de dinâmicas no Excel

```
var app = novo aplicativo { Visível = true }; app.Workbooks.Add(); Planilha
de planilha = app.ActiveSheet;
Início do intervalo = planilha.Cells[1, 1]; Fim do intervalo =
planilha.Cells[1, 20]; planilha.Range[início, fim].Valor =
Enumerable.Range(1, 20)

.ToArray();
```

Este é exatamente o mesmo código da listagem 14.5, mas sem as conversões.

Vale ressaltar que as conversões ainda são verificadas em tempo de execução. Se você alterasse a declaração de início para Planilha, a conversão falharia e uma exceção seria lançada. Claro, você não precisa realizar a conversão. Você poderia simplesmente deixar tudo dinâmico, conforme mostrado na listagem a seguir.

#### Listagem 14.7 Usando dinâmico em todos os lugares

```
var app = novo aplicativo { Visível = true }; app.Workbooks.Add(); planilha
dinâmica = app.ActiveSheet;
início dinâmico = planilha.Células[1, 1]; final dinâmico =
planilha.Células[1, 20]; planilha.Range[início, fim].Valor =
Enumerable.Range(1, 20)

.ToArray();
```

O que é mais claro? Sou um fã da digitação estática antiquada, então prefiro a versão da listagem 14.6. Ele indica os tipos que espero em cada linha, portanto, se houver algum problema, posso descobrir imediatamente, em vez de esperar até tentar usar um valor de uma forma que pode não ser suportada.

Em termos de produtividade no desenvolvimento inicial, existem prós e contras em ambas as abordagens. Usando dinâmico, você não precisa descobrir qual tipo específico você

esperar; você pode simplesmente usar o valor e, desde que todas as operações necessárias sejam suportadas, você estará bem. Por outro lado, usando digitação estática, você pode ver o que está disponível em cada etapa via IntelliSense. Você ainda está usando digitação dinâmica para fornecer o conversão implícita para planilha e intervalo - você está usando-o apenas para uma etapa de cada vez tempo em vez de atacado. A mudança de digitação estática para dinâmica pode não parecer gosto muito para começar, porque o exemplo é relativamente simples, mas como a complexidade do código aumenta, assim como o benefício de legibilidade da remoção de todas essas conversões.

De certa forma, tudo isso foi uma explosão do passado – COM é uma tecnologia relativamente antiga. Agora passaremos à interoperação com algo muito mais recente: IronPython.

### 14.3.2 Linguagens dinâmicas como IronPython

Nesta seção usarei apenas o IronPython como exemplo, mas esse certamente não é o único linguagem dinâmica disponível para o DLR. É sem dúvida o mais maduro, mas há já alternativas como IronRuby e IronScheme. Um dos objetivos declarados do DLR é tornar mais fácil para designers de linguagem iniciantes criarem uma linguagem de trabalho que tem acesso às enormes bibliotecas do .NET Framework, bem como boa interoperabilidade com outras linguagens DLR e as linguagens .NET tradicionais , como C#.

#### POR QUE EU QUERO USAR O IRONPYTHON DE C#?

Há muitos motivos pelos quais você pode querer interoperar com uma linguagem dinâmica, apenas já que tem sido benéfico interoperar com outras linguagens gerenciadas do .NET infância. É claramente útil para um desenvolvedor VB poder usar uma biblioteca de classes escrita em C# e vice-versa, então por que o mesmo não aconteceria com linguagens dinâmicas? Perguntei Michael Foord, um dos autores de *Iron Python in Action* (Manning, 2009), por vir apresente algumas ideias para usar o IronPython em um aplicativo C#. Aqui está a lista dele:

- ÿ Script de usuário
- ÿ Escrevendo uma camada da sua aplicação no IronPython
- ÿ Usando Python como linguagem de configuração
- ÿ Usar Python como mecanismo de regras com regras armazenadas como texto (mesmo em um banco de dados)
- ÿ Usar uma biblioteca que está disponível em Python, mas não tem equivalente em .NET
- ÿ Colocar um intérprete ao vivo em seu aplicativo para depuração

Se você ainda estiver cético, considere que incorporar uma linguagem de script em um aplicativo convencional está longe de ser incomum — *Civilization IV*, de Sid Meier. game3 de computador pode ser programado com Python. Isso não é apenas uma reflexão tardia para modificações – grande parte da jogabilidade principal é escrita em Python. Depois que eles construíram o motor, os desenvolvedores descobriram que era um ambiente de desenvolvimento mais poderoso do que eles originalmente imaginaram.

Neste capítulo trabalharei com o exemplo único de uso do Python como linguagem de configuração. Assim como no exemplo COM , vou mantê-lo simples, mas espero que ele forneça um ponto de partida suficiente para você experimentar mais com ele, se estiver interessado.

<sup>3</sup> Ou modo de vida, dependendo de como você vê o mundo e do seu nível de dependência em jogar.

**COMEÇANDO: INCORPORANDO “OLÁ, MUNDO”**

Existem vários tipos disponíveis se você quiser *hospedar* ou *incorporar* outra linguagem em um aplicativo C#, dependendo do nível de flexibilidade e controle que você deseja alcançar. Usaremos apenas ScriptEngine e ScriptScope aqui, porque nossos requisitos são primitivos. Neste exemplo, você sabe que sempre usará Python, então pode pedir ao framework IronPython para criar um ScriptEngine diretamente; em situações mais gerais, você pode usar um ScriptRuntime para escolher implementações de linguagem dinamicamente por nome. Cenários mais exigentes podem exigir que você trabalhe com ScriptHost e ScriptSource, bem como use mais recursos dos outros tipos também.

Não satisfeito em apenas imprimir hello, world uma vez, este exemplo inicial fará isso *duas vezes*, primeiro usando o texto passado diretamente para o mecanismo como uma string e depois carregando um arquivo chamado HelloWorld.py. A listagem a seguir mostra tudo que você precisa.

**Listagem 14.8 Imprimindo hello, world duas vezes usando Python incorporado em C#**

```
Mecanismo ScriptEngine = Python.CreateEngine();
engine.Execute("imprimir 'olá, mundo'");
motor.ExecuteFile("HelloWorld.py");
```

Você pode achar esta listagem um tanto enfadonha ou muito emocionante, ambos pelo mesmo motivo. É simples de entender, exigindo pouca explicação. Ele faz pouco em termos de saída real... e ainda assim o fato de ser *tão* fácil incorporar código Python em C# é motivo de comemoração. É verdade que o nível de interação é mínimo até agora, mas realmente não poderia ser muito mais fácil do que isso.

**AS MUITAS FORMAS LITERAIS DE STRING DO PYTHON** O arquivo Python contém uma

única linha: `print "hello, world"`. Observe as aspas duplas no arquivo, em comparação com as aspas simples na string literal que foi passada para `engine.Execute()`.

Qualquer um estaria bem em qualquer fonte. Python tem várias representações literais de string, incluindo aspas simples triplas ou aspas duplas triplas para literais multilinhas.

Mencionei isso apenas porque é útil não precisar escapar de aspas duplas sempre que você quiser colocar o código Python em uma string literal C#.

O próximo tipo que veremos é o ScriptScope, que será crucial para o script de configuração.

**ARMAZENANDO E RECUPERANDO INFORMAÇÕES DE UM SCRIPTSCOPE**

Os métodos de execução que usamos possuem sobrecargas com um segundo parâmetro – um escopo. Em termos mais simples, isto pode ser considerado como um dicionário de nomes e valores. As linguagens de script geralmente permitem que variáveis sejam atribuídas sem qualquer declaração explícita e, quando isso é feito no nível superior de um programa (em vez de em uma função ou classe), isso geralmente afeta um escopo global .

Quando uma instância de ScriptScope é passada para um método de execução, ela é usada como escopo global para o script que você solicitou ao mecanismo para executar. O script pode recuperar valores existentes do escopo e criar novos valores, conforme mostrado na listagem a seguir.

### Listagem 14.9 Passando informações entre um host e um script usando ScriptScope

```
string python = @" texto =
'olá' saída = entrada
+ 1 "; Mecanismo ScriptEngine
=
Python.CreateEngine(); Escopo ScriptScope = engine.CreateScope();
escopo.SetVariable("entrada", 10); motor.Execute(python,
escopo); Console.WriteLine(scope.GetVariable("texto"));
Console.WriteLine(scope.GetVariable("entrada"));
Console.WriteLine(scope.GetVariable("saída"));

C Define variável para
B Código Python incorporado
Código Python para usar

D de volta do escopo
A Busca variáveis
```

Neste código, o código-fonte Python é incorporado ao código C# como uma string literal **B**, em vez de colocá-lo em um arquivo, para que seja mais fácil ver todo o código em um só lugar. Não recomendo que você faça isso em código de produção, em parte porque o Python é sensível a espaços em branco — reformatar o código de uma forma aparentemente inofensiva pode fazer com que ele falhe completamente em tempo de execução.

Os métodos SetVariable e GetVariable simplesmente colocam valores no escopo **C** e os recuperam novamente **D** da maneira óbvia. Eles são declarados em termos de objeto e não de dinâmico, como você poderia esperar. Mas GetVariable também permite especificar um argumento de tipo, que atua como uma solicitação de conversão.

Isso não é exatamente o mesmo que converter o resultado do método não genérico, já que o último apenas desempacota o valor, o que significa que você precisa convertê-lo exatamente para o tipo certo. Por exemplo, você pode colocar um número inteiro no escopo, mas recuperá-lo como um duplo:

```
scope.SetVariable("num", 20) double x =
scope.GetVariable<double>("num") double y = (double)
scope.GetVariable("num");

B Converte com sucesso para double
C Unboxing lança exceção
```

A primeira chamada é bem-sucedida: você está informando explicitamente a GetVariable que tipo deseja **B**, para que ele saiba como forçar o valor de maneira adequada. A segunda chamada **C** lançará uma InvalidCastException, assim como faria em qualquer outra situação em que você tentasse desempacotar um valor usando o tipo errado.

O escopo também pode conter funções, que você pode recuperar e depois chamar dinamicamente, passando argumentos e retornando valores. A maneira mais fácil de fazer isso é usar o tipo dinâmico, conforme mostrado na listagem a seguir.

### Listagem 14.10 Chamando uma função declarada em um ScriptScope

```
string python = @" def
sayHello(usuário): print 'Olá %
    (nome)s' % {'nome': usuário}
";
mechanism ScriptEngine = Python.CreateEngine(); escopo ScriptScope
= engine.CreateScope(); motor.Execute (python, escopo); função
dinâmica = escopo.GetVariable("sayHello");
function("Jon");
```

Os arquivos de configuração podem não precisar dessa habilidade com frequência, mas ela pode ser útil em outras situações. Por exemplo, você poderia facilmente usar Python para criar scripts de um programa de desenho gráfico, fornecendo uma função a ser chamada em cada ponto de entrada. Um exemplo simples disso pode ser encontrado no site do livro em <http://mng.bz/6yGi>.

Há diversas situações em que é útil ter algum tipo de avaliador de expressão executando o código do usuário inserido em tempo de execução, como avaliar regras de negócios para descontos, custos de envio e assim por diante. Também pode ser útil poder

altere essas regras em formato de texto sem precisar recompilar ou reimplantar binários. A Listagem 14.10 é bastante inofensiva - outro exemplo no código-fonte para download é entrelaçado e fora das duas línguas de forma bastante mais tortuosa, mostrando que as chamadas podem ir nos dois sentidos: de C# para IronPython, como você viu, e de IronPython para C#.

### COLOCANDO TUDO JUNTO

Agora que você pode inserir valores em seu escopo, está basicamente pronto. Você poderia potencialmente agrupar o escopo em outro objeto, fornecendo acesso por meio de um indexador ou até mesmo acessando os valores dinamicamente usando as técnicas mostradas na seção 14.5. A aplicação o código pode ser algo assim:

```
Configuração estática LoadConfiguration()
{
    Mecanismo ScriptEngine = Python.CreateEngine();
    Escopo ScriptScope = engine.CreateScope();
    engine.ExecuteFile("configuration.py", escopo);
    retornar Configuração.FromScriptScope(escopo);
}
```

A forma exata do tipo de configuração dependerá da sua aplicação, mas é improvável que seja um código terrivelmente emocionante. Forneci um exemplo de implementação dinâmica na fonte completa que permite recuperar valores como propriedades e chamar funções diretamente também. Claro, você não está limitado a usar tipos primitivos em sua configuração: o código Python pode ser arbitrariamente complexo, construindo coleções, conectando componentes e serviços, e assim por diante. Poderia desempenhar muitas das funções de um normal injeção de dependência ou inversão do contêiner de controle.

O importante é que agora você tenha um arquivo de configuração ativo dos tradicionais arquivos XML passivos e .ini. Claro, você poderia ter incorporado seu própria linguagem de programação em arquivos de configuração anteriores, mas o resultado seria provavelmente teria sido menos poderoso e teria exigido muito mais esforço para ser implementado. Como um exemplo de onde isso poderia ser útil em uma situação mais simples do que a injeção de dependência completa, você pode querer configurar o número de threads a serem usados para alguns componente de processamento em segundo plano em seu aplicativo. Você normalmente pode usar como muitos threads, pois você tem processadores no sistema, mas ocasionalmente reduza-os para para ajudar outro aplicativo a funcionar sem problemas no mesmo sistema. O arquivo de configuração simplesmente mudaria de algo assim

```
agenteThreads = System.Environment.ProcessorCount
agentThreadName = 'Agente de processamento'
```

para isso

```
agentThreads = 1  
agentThreadName = 'Agente de processamento (somente thread único)'
```

Essa alteração não exigiria que o aplicativo fosse reconstruído ou reimplantado — você só precisaria editar o arquivo e reiniciar o aplicativo. Aplicações particularmente inteligentes podem até optar por se reconfigurar rapidamente. (Descobri que geralmente essa capacidade é mais difícil de implementar do que o valor extra que ela traz, mas em certos lugares pode fazer uma grande diferença. A capacidade de alterar os níveis de registro para um determinado trecho de código ou mesmo apenas para um código específico usuário que está tendo dificuldades pode tornar a depuração muito mais fácil.)

Além de executar funções, ainda não analisamos o uso do Python de uma forma particularmente dinâmica. Todo o poder do Python está disponível e, usando o tipo dinâmico em seu código C#, você pode aproveitar as vantagens da metaprogramação e de todos os outros recursos dinâmicos. O compilador C# é responsável por representar seu código de maneira apropriada, e o mecanismo de script é responsável por pegar esse código e descobrir o que ele significa para Python. Apenas não sinta que *precisa* fazer algo particularmente inteligente para que valha a pena incorporar o mecanismo de script em seu aplicativo.

É um passo simples em direção a um aplicativo mais poderoso.

**QUANTO PODER VOCÊ QUER DAR AOS SEUS AUTORES DE SCRIPT?** Se você estiver executando código arbitrário, especialmente código inserido por usuários externos do sistema, você deve pensar seriamente sobre segurança e, possivelmente, executar o script em algum tipo de ambiente de área restrita. A discussão deste tópico está fora do escopo deste livro, mas precisa ser considerada cuidadosamente.

Até agora, nossos exemplos têm interoperado com outros sistemas. No entanto, a digitação dinâmica pode fazer sentido mesmo dentro de um sistema puramente gerenciado. Vamos visitar alguns exemplos.

#### 14.3.3 Digitação dinâmica em código puramente gerenciado

É quase certo que você já usou algo como digitação dinâmica no passado, mesmo que não tenha sido seu próprio código que fez o trabalho. A vinculação de dados é o exemplo mais simples disso: sempre que você especifica algo como `ListControl.DisplayMember`, está solicitando à estrutura que encontre uma propriedade em tempo de execução com base em seu nome. Se você já usou reflexão diretamente em seu próprio código, novamente estará usando informações que só estão disponíveis em tempo de execução.

Na minha experiência, a reflexão está sujeita a erros e, mesmo quando funciona, pode ser necessário fazer um esforço extra para otimizá-la. Em alguns casos, a digitação dinâmica pode substituir completamente o código baseado em reflexão; também pode ser mais rápido, dependendo exatamente do que você está fazendo.

É particularmente complicado usar tipos e métodos genéricos de reflexão. Por exemplo, se você tem um objeto que implementa `IList<T>` para algum argumento de tipo T, pode ser difícil descobrir exatamente o que é T. Se a única razão para descobrir T for chamar outro método genérico, você realmente deseja apenas perguntar ao

compilador para chamar o que ele *teria* chamado se você conhecesse o tipo real. Claro, é exatamente isso que a digitação dinâmica faz. Usarei este cenário como nosso primeiro exemplo.

#### INFERÊNCIA DE TIPO DE TEMPO DE EXECUÇÃO

Se você quiser fazer mais do que apenas chamar um único método, geralmente é melhor agrupar todo o trabalho adicional em um método genérico. Você pode então chamar o método genérico dinamicamente, mas escrever todo o restante do código usando digitação estática. A Listagem 14.11 mostra um exemplo simples disso.

Finja que você recebeu uma lista de algum tipo e um novo elemento de alguma outra parte do sistema. Foi prometido que eles são compatíveis, mas você não conhece seus tipos estaticamente. Existem vários motivos pelos quais isso pode acontecer — pode ser o resultado da desserialização em outro lugar, por exemplo. De qualquer forma, seu código pretende adicionar o novo elemento ao final da lista, mas somente se houver menos de 10 elementos na lista no momento. O método retorna se o elemento foi realmente adicionado ou não. Obviamente, na vida real a lógica de negócios seria mais complicada, mas a questão é que você realmente gostaria de poder usar tipos fortes para essas operações.

A listagem a seguir mostra o método digitado estaticamente e a chamada dinâmica para ele.

#### Listagem 14.11 Usando inferência de tipo dinâmico

```
private static bool AddConditionallyImpl<T>(IList<T> lista, T item) {
    if (lista.Count < 10) {
        lista.Add(item);
        return true;
    }
    return false;
}

public static bool AddConditionally(IList dinâmica, object item) {
    return AddConditionallyImpl(dinâmica, item);
}
...
lista de objetos = new Lista<string> { "x", "y" };
object item = "z";
AdicionarCondicionalmente(lista, item);
```

**Normal estaticamente**  
**Código digitado B**

**Chama o método auxiliar**  
**C dinamicamente**

**Eventualmente liga**  
**AddConditionallyImpl<string>**

O método público possui parâmetros dinâmicos; nas versões anteriores do C# talvez fossem necessários `IEnumerable` e `Object`, contando com verificações complicadas com reflexão para descobrir o tipo da lista e então invocar o método genérico com reflexão. Com a digitação dinâmica, você pode simplesmente chamar uma implementação **B** fortemente tipada usando os argumentos dinâmicos **C**, isolando o acesso dinâmico à chamada única no método wrapper. É claro que a chamada ainda pode falhar, mas você poupará o esforço de tentar determinar o argumento de tipo apropriado.

Você também pode expor publicamente o método fortemente tipado para evitar a digitação dinâmica para chamadores que conheciam seus tipos de lista estaticamente. Valeria a pena manter os nomes diferentes nesse caso, para evitar chamar acidentalmente a versão dinâmica devido a um leve

confunda com os tipos estáticos dos argumentos. (Também é muito mais fácil fazer a chamada certa na versão dinâmica quando os nomes são diferentes!)

Como outro exemplo de digitação dinâmica em código puramente gerenciado, já lamentei a falta de suporte a operadores genéricos em C#. Não existe o conceito de especificar uma restrição dizendo “T deve ter um operador que me permita somar dois valores do tipo T”. Você viu isso em nossa demonstração inicial de digitação dinâmica (veja listagem 14.4), então mencioná-lo aqui não deve ser nenhuma surpresa. Vamos pegar o operador de consulta Sum do LINQ e torná-lo dinâmico.

### COMPENSANDO A FALTA DE OPERADORES GENÉRICOS

Você já olhou a lista de sobrecargas de `Enumerable.Sum`? É bem longo.

É certo que metade das sobrecargas se deve a uma projeção, mas mesmo assim existem 10 sobrecargas, cada uma das quais apenas pega uma sequência de elementos e os soma, e isso nem cobre a soma de valores não assinados, ou bytes, ou shorts. Por que não usar a digitação dinâmica para tentar fazer tudo em um único método?

Embora utilizemos digitação dinâmica internamente, o método mostrado na listagem 14.12 é digitado estaticamente. Você poderia declará-lo como um método não genérico somando um `IEnumerable<dinâmico>`, mas isso não funcionaria bem devido às limitações da covariância. Nomeei o método `DynamicSum` em vez de `Sum` para evitar conflito com os métodos em `Enumerable`. O compilador escolherá uma sobrecarga não genérica em vez de uma genérica, onde ambas as assinaturas têm os mesmos tipos de parâmetros e é mais simples evitar a colisão em primeiro lugar.

#### Listagem 14.12 Somando uma sequência arbitrária de elementos dinamicamente

```
public static T DynamicSum<T>(esta fonte IEnumerable<T>) {
    total dinâmico = padrão(T); foreach (elemento
    T na fonte) {
        total = (T) (total + elemento);
    }
    ...
    byte[] bytes = novo byte[] { 1, 2, 3 };
    Console.WriteLine(bytes.DynamicSum());
}
```



**B** Digitado dinamicamente  
para uso posterior

← Escolhe o operador  
de adição dinamicamente

← Impressões 6

O código é basicamente simples; parece quase exatamente igual a qualquer uma das implementações das sobrecargas normais de `Sum`. Ele omite a verificação se `source` é nulo apenas por questões de brevidade, mas a maior parte do resto é bastante simples. Há alguns pontos interessantes.

Primeiro, você usa `default(T)` para inicializar `total`, que é declarado como dinâmico para que você obtenha o comportamento dinâmico desejado B. Você precisa começar com um valor inicial de alguma forma; você poderia tentar usar o primeiro valor da sequência, mas ficaria preso se a sequência estivesse vazia. Para tipos de valor não anuláveis, `default(T)` é quase sempre um valor apropriado: é um zero natural. Para tipos de referência, você acabará

adicionar o primeiro elemento da sequência a nulo, o que pode ou não ser apropriado. Para tipos de valor anuláveis, você acabará tentando adicionar o primeiro elemento ao valor nulo valor para esse tipo, o que certamente *não* será apropriado.

Segundo, você converte o resultado da adição de volta para T, mesmo que seja atribuído para uma variável dinâmica. Isso pode parecer estranho, mas você precisa pensar nos resultados de somando dois bytes juntos. O compilador C# normalmente promoveria cada operando para int antes de realizar a adição. Sem a conversão, a variável total seria acabaria armazenando um valor int , o que causaria uma exceção quando o retorno instrução tentou convertê-lo de volta para byte.

Ambos os pontos levam a questões mais profundas, mas esse não é o objetivo desta seção. Escrevi uma investigação mais detalhada sobre soma dinâmica no site do livro (veja <http://mng.bz/0N37>).

Só para provar que nosso código é capaz de mais do que aritmética com números normais, a listagem 14.13 mostra um exemplo de soma de valores de TimeSpan .

#### Listagem 14.13 Somando uma lista de elementos TimeSpan dinamicamente

```
var tempos = new List<TimeSpan>
{
    2.Horas(), 25.Minutos(), 30.Segundos(),
    45.Segundos(), 40.Minutos()
};
Console.WriteLine(times.DynamicSum());
```

Os valores TimeSpan são criados usando métodos de extensão por conveniência, mas o somatório é totalmente dinâmico, resultando em um intervalo total de 3 horas, 6 minutos e 15 segundos.

#### DIGITAÇÃO DE PATO

Às vezes você sabe que um membro com um nome específico estará disponível no momento da execução, mas você não pode dizer ao compilador exatamente de qual membro você está falando porque vai depender do tipo. De certa forma, este é um exemplo mais geral da mesmo problema que acabamos de resolver, exceto usando métodos e propriedades normais em vez de operadores.

Há uma diferença: normalmente você tentaria capturar o que há de comum em uma interface ou classe base abstrata. Você não pode fazer isso com operadores, mas é a abordagem normal para métodos e propriedades. Infelizmente isso nem sempre funciona – especialmente se várias bibliotecas estiverem envolvidas. O .NET Framework é bastante consistente aqui, mas você já vi um exemplo em que não funciona muito bem. No capítulo 12, examinamos o otimizações disponíveis para contar uma sequência e vi que tanto ICollection quanto ICollection<T> possuem uma propriedade Count , mas não possuem interface de ancestral comum com essa propriedade, portanto, é necessário tratá-los separadamente.

A digitação Duck permite que você acesse Count sem realizar a verificação de tipo self, conforme mostrado na listagem a seguir.

**Listagem 14.14 Acessando uma propriedade Count com digitação duck**

```
estático void PrintCount (coleção Inumerable)
{
    dinâmico d = coleção; contagem
    interna = d.Contagem;
    Console.WriteLine(contagem);
}
...
PrintCount(novo BitArray(10));
PrintCount(new HashSet<int> { 3, 5 });
PrintCount(nova Lista<int> { 1, 2, 3 });
```

O método PrintCount é restrito a implementações de Ienumerable pelo mesmo motivo que os inicializadores de coleção: é uma boa indicação de que a propriedade Count que você acaba usando é apropriada. As coleções de teste são um BitArray (que implementa apenas ICollection), um HashSet<int> (que implementa apenas ICollection<int>) e um List<int> (que implementa ambos). Em todos os casos, a propriedade correta é encontrada em tempo de execução.

**Implementação explícita de interface e dinâmica não combinam bem** Quando

tentei testar esse código pela primeira vez, usei um int[], que é implicitamente conversível para ambas as interfaces envolvidas. Fiquei, portanto, surpreso quando o método PrintCount falhou em tempo de execução... até que pensei mais sobre isso. A ligação em tempo de execução é realizada usando o tipo real do objeto, que neste caso é um int[]. Os tipos de array não expõem publicamente uma propriedade Count — eles usam implementação de interface explícita para isso. Você só pode usar Count ao visualizar um objeto array de uma maneira específica.

Este é apenas um exemplo em que a digitação dinâmica pode se comportar de maneira lógica, mas pode ser inesperada, a menos que você tome cuidado. Estou coletando uma lista contínua dessas esquisitices no site do livro (veja <http://mng.bz/5y7M>); por favor, deixe-me saber se você encontrar algum novo.

Continuaremos com o exemplo de recuperação da contagem de itens, mas desta vez veremos como a resolução de sobrecarga em tempo de execução pode oferecer uma alternativa ao teste de tipo explícito.

**DESPACHO MÚLTIPLO**

Com a digitação estática, o C# usa *despacho único*: na execução, o método exato chamado depende apenas do tipo real do destino da chamada do método, por meio de substituição. A sobrecarga é decidida em tempo de compilação. Ocionalmente, o *despacho múltiplo* é útil para encontrar a implementação mais especializada de um método com base nos tipos de tempo de execução dos argumentos - novamente, é isso que a tipagem dinâmica fornece.

A listagem a seguir demonstra como o despacho múltiplo permitiria uma implementação mais variada e robusta de contagem otimizada.

**Listagem 14.15 Contando diferentes tipos de forma eficiente usando despacho múltiplo**

```

private static int CountImpl<T>(ICollection<T> coleção) {
    retornar coleção.Count;
}

private static int CountImpl(coleção ICollection) {
    retornar coleção.Count;
}

private static int CountImpl(string texto) {
    retornar texto.Comprimento;
}

private static int CountImpl(ICollection Inumerable) {

    contagem interna = 0;
    foreach (item do objeto na coleção) {

        contar++;
    }
    contagem de retorno;
}

public static void PrintCount(coleção Inumerable) {

    dinâmico d = coleção; contagem
    interna = ContagemImpl(d);
    Console.WriteLine(contagem);
}
...
PrintCount(novoBitArray(5)); PrintCount(new
HashSet<int> { 1, 2 });
PrintCount("ABC");
PrintCount("ABCDEF".Where(c
=> c > 'B'));

```

Você sabe que pelo menos uma sobrecarga de CountImpl será apropriada em tempo de execução porque o parâmetro para PrintCount é do tipo IEnumerable. Você depende da digitação dinâmica para executar o mesmo trabalho que o explícito “se for um ICollection<T>, use esta implementação; se for uma ICollection, use esta implementação” etapas que usamos ao escolher um elemento aleatório na listagem 12.17. Como exemplo de como isso é mais do que apenas usar a propriedade Count , se estiver disponível, a listagem 14.15 inclui uma otimização para strings, onde você pode usar a propriedade Length para obter o resultado correto rapidamente.

Mesmo usando despacho múltiplo aqui, você ainda pode ter problemas no momento da execução: e se o tipo real implementasse ICollection<string> e ICollection<int> por meio de implementação de interface explícita? Haveria dois resultados possíveis dependendo de qual implementação de Count foi escolhida. Neste caso, a ligação seria ambígua, levando a uma exceção. Felizmente, esses casos patológicos são provavelmente raros.

Estes são apenas alguns exemplos de áreas em que você *pode* querer usar a digitação dinâmica, mesmo que não esteja tentando interoperar com mais nada. A seguir, nos aprofundaremos em como todos esses efeitos são alcançados, antes de finalizarmos o capítulo implementando nosso próprio comportamento dinâmico.

Devo avisar que as coisas estão prestes a ficar complicadas. Na verdade, é tudo extremamente elegante, mas é complicado porque as linguagens de programação fornecem um rico conjunto de operações, e representar todas as informações necessárias sobre essas operações como dados e depois agir de acordo com elas são tarefas complexas. A boa notícia é que você não precisa entender tudo intimamente. Como sempre, você aproveitará melhor a digitação dinâmica quanto mais familiarizado estiver com o mecanismo por trás dela, mas mesmo que use apenas as técnicas que viu até agora, pode haver situações em que isso o torne muito mais útil. mais produtivo.

## 14.4 Olhando os bastidores

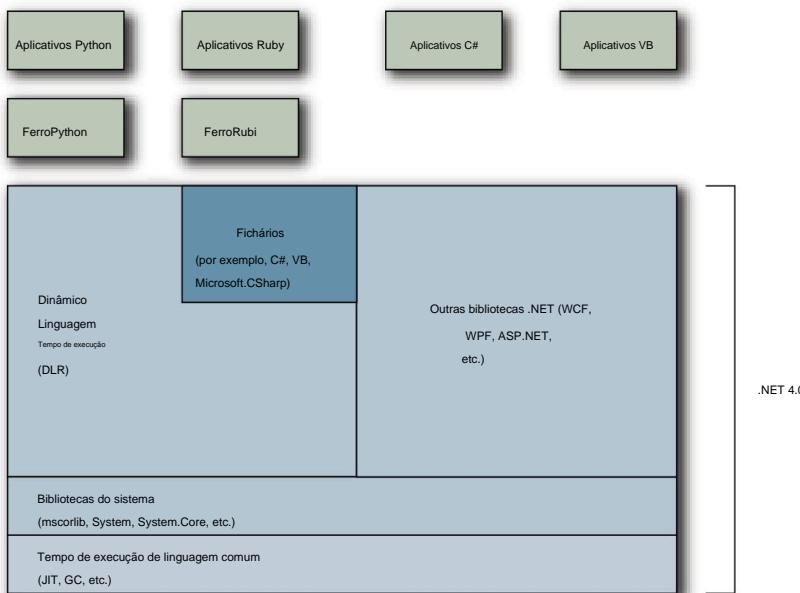
Apesar do aviso do parágrafo anterior, não entrarei em *muitos* detalhes sobre o funcionamento interno da digitação dinâmica. Seria muito terreno a percorrer, tanto no que diz respeito à estrutura como às alterações linguísticas. Não é sempre que eu me esquivo dos detalhes das especificações, mas neste caso eu realmente acredito que não há muito a ganhar aprendendo tudo. Abordarei os pontos mais importantes (e interessantes) e posso recomendar completamente o blog de Sam Ng (<http://blogs.msdn.com/b/samng/>), a especificação da linguagem C# e a página do projeto DLR (<http://mng.bz/0M6A>) para obter mais informações se precisar se aprofundar em um cenário específico.

Meu objetivo final é ajudá-lo a entender o que o compilador C# está fazendo e o código que ele emite para obter vinculação dinâmica em tempo de execução. Infelizmente, nenhum código gerado fará sentido até que você veja o mecanismo que sustenta tudo: o DLR. Você pode gostar de pensar em um programa de digitação estática como uma peça de teatro convencional com um roteiro fixo, e em um programa de digitação dinâmica mais como um show de improvisação. O DLR toma o lugar dos cérebros dos atores que inventam freneticamente algo a dizer em resposta às sugestões do público. Vamos conhecer nossa estrela de raciocínio rápido.

### 14.4.1 Apresentando o tempo de execução de linguagem dinâmica

Já faz algum tempo que uso a sigla *DLR*, expandindo-a ocasionalmente para Dynamic Language Runtime, mas nunca explicando o que é. Isso foi deliberado: tenho tentado entender a natureza da digitação dinâmica e como ela afeta os desenvolvedores, em vez dos detalhes da implementação. Mas essa desculpa nunca duraria até o final do capítulo, então aqui estamos. Em seus termos mais básicos, o Dynamic Language Runtime é uma biblioteca que todas as linguagens dinâmicas e o compilador C# usam para executar código dinamicamente.

Por incrível que pareça, é realmente apenas uma biblioteca. Apesar do nome, ele não está no mesmo nível do CLR (Common Language Runtime) — ele não lida com compilação JIT, empacotamento de API nativa, coleta de lixo e assim por diante. Mas baseia-se em muitos dos



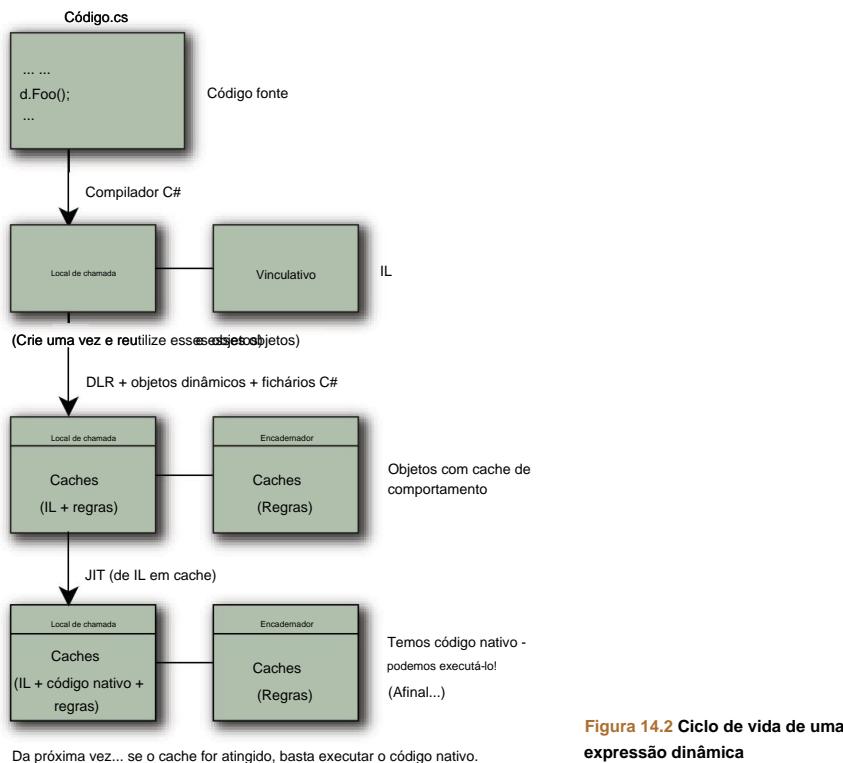
**Figura 14.1** Como os componentes do .NET 4 se encaixam, permitindo que linguagens estáticas e dinâmicas sejam executadas na mesma plataforma subjacente

funcionam no .NET 2.0 e 3.5, principalmente nos tipos DynamicMethod e Expression . O A API da árvore de expressão foi expandida no .NET 4 para permitir que o DLR expresse mais conceitos também. A Figura 14.1 mostra como tudo se encaixa.

Além do DLR, a figura 14.1 mostra outra biblioteca que pode ser nova para você. Um dos assemblies na parte dos fichários do diagrama é Microsoft.CSharp. Ele contém vários tipos que são referenciados pelo compilador C# quando você usa no seu código. Confusamente, isso não inclui o Microsoft.CSharp existente .Compiler e Microsoft.CSharp.CodeDomProvider. (Eles nem estão no mesmo montagem como um ao outro!) Você verá exatamente para que os novos tipos são usados na seção 14.4.2, onde descompilaremos alguns códigos escritos usando dinâmico.

Um outro aspecto importante diferencia o DLR do resto do .NET Framework: é fornecido como código aberto. O código completo está em um projeto CodePlex (<http://dlr.codeplex.com>), então você pode baixá-lo e ver o funcionamento interno. Um dos benefícios desta abordagem é que o DLR não teve que ser reimplementado para Mono (<http://mono-project.com>): o mesmo código é executado no .NET e em seu primo multiplataforma.

Embora o DLR não lide diretamente com o código nativo, você pode pensar nele como se estivesse fazendo um trabalho *semelhante* ao CLR em certo sentido: assim como o CLR converte IL (Linguagem Intermediária) em código nativo, o DLR converte código representado usando fichários, sites de chamada, meta-objetos e vários outros conceitos em árvores de expressão que podem então ser compiladas em IL e, eventualmente, em código nativo pelo CLR. A Figura 14.2 mostra uma visão simplificada do ciclo de vida de uma única avaliação de uma expressão dinâmica.



Como você pode ver, um dos aspectos importantes do DLR é o cache multinível. Isso é crucial por razões de desempenho, mas para entender esse e outros conceitos que já mencionado, precisaremos mergulhar uma camada mais fundo.

#### 14.4.2 Conceitos básicos do DLR

Podemos resumir o objectivo do DLR em termos *muito* gerais, assumindo uma abordagem de alto nível representação de código e execução desse código, com base em diversas informações que só podem ser conhecidas em tempo de execução. Nesta seção apresentarei muitos terminologia para descrever como o DLR funciona, mas tudo contribui para esse mirar.

##### SITES DE CHAMADA

O primeiro conceito de que precisamos é um *sítio de chamada*. Este é uma espécie de átomo do DLR – o menor pedaço de código que pode ser considerado como uma única unidade executável. Uma expressão pode contém muitos sites de chamadas, mas o comportamento é construído de forma natural, avaliando um ligue para o site por vez.

No restante da discussão, consideraremos apenas um único site de chamada. Será útil tenho um pequeno exemplo de site de chamada para consultar, então aqui está um exemplo simples, onde d é uma variável do tipo dinâmico:

```
d.Foo(10);
```

O site de chamada é representado no código como System.Runtime.CompilerServices.Call-Site<T>. Você verá um exemplo completo de como os sites de chamada são criados e usados na próxima seção, quando observarmos o que o compilador C# faz em tempo de compilação, mas aqui está um exemplo do código que pode ser chamado para criar o site do snippet anterior:

```
CallSite<Action<CallSite, objeto, int>>.Create(Binder.InvokeMember(
    CSharpBinderFlags.ResultDiscarded, "Foo", nulo, typeof(Teste),
    novo CSharpArgumentInfo[] {
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, nulo),
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.Constant | 
            CSharpArgumentInfoFlags.UseCompileTimeType,
            nulo) }));

```

Agora que temos um site de chamada, podemos executar o código? Não exatamente.

## RECEPTORES E PASTAS

Assim como um call site, precisamos de algo para decidir o que significa e como executá-lo. No DLR, duas entidades podem decidir isso: o *receptor* de uma chamada e o *fichário*. O receptor de uma chamada é simplesmente o objeto no qual um membro é chamado. Em nossa chamada de exemplo site, o receptor é o objeto ao qual d se refere em tempo de execução. O fichário irá depender do idioma da chamada e faz parte do site da chamada. Nesse caso, você pode ver que o compilador C# emita código para criar um fichário usando Binder.InvokeMember. O A classe Binder , neste caso, é Microsoft.CSharp.RuntimeBinder.Binder, então é realmente Específico para C#. O fichário C# também reconhece COM e executará a ligação COM apropriada se o receptor for um objeto IDispatch .

O DLR sempre dá precedência ao receptor: se for um objeto dinâmico que conhece como lidar com a chamada, então ele usará qualquer caminho de execução fornecido pelo objeto. Um O objeto pode se anunciar como dinâmico implementando a nova interface IDynamicMeta-ObjectProvider . O nome é complicado, mas contém apenas um único membro: GetMetaObject. Você precisará ser um ninja da árvore de expressão e conhecer o DLR muito bem para implementar GetMetaObject corretamente. Mas nas mãos certas, isso pode ser um ferramenta poderosa, proporcionando interação de nível inferior com o DLR e seu cache de execução. Se você precisa implementar comportamento dinâmico com alto desempenho, vale a pena o investimento para aprender os detalhes.

Existem duas implementações públicas de IDynamicMetaObjectProvider incluídas na estrutura para facilitar a implementação de comportamento dinâmico em situações onde o desempenho não é tão crítico. Veremos tudo isso com mais detalhes na seção 14.5, mas por enquanto você só precisa estar ciente da interface em si e de que ela representa a capacidade de um objeto reagir dinamicamente.

Se o receptor não for dinâmico, o fichário decide como o código deve ser executado. No código C#, ele aplicaria regras específicas do C# ao código e descobriria o que fazer. Se você estivesse criando sua própria linguagem dinâmica, você poderia implementar sua própria fichário para decidir como ele deve se comportar em geral (quando o objeto não substitui o comportamento). Isso está muito além do escopo deste livro, mas é uma interessante tópico em si; um dos objetivos do DLR é facilitar a implementação do seu próprias línguas.

## REGRAS E CACHES

A decisão sobre como executar uma chamada é representada como *regra*. Fundamentalmente, isso consiste em dois elementos de lógica: as circunstâncias sob as quais o local de chamada deve comportar dessa maneira e o comportamento em si.

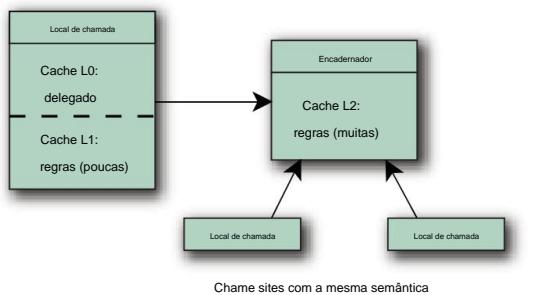
A primeira parte é realmente para otimização. Suponha que você tenha um site de chamada que represente adição de dois valores dinâmicos, e na primeira vez que é avaliado, ambos os valores são de digite byte. O encadernador fez um grande esforço para descobrir que isso significa ambos os operandos devem ser promovidos para int, e o resultado deve ser a soma desses inteiros. Ele pode reutilizar essa operação sempre que os operandos forem bytes.

Verificar a validade de um conjunto de resultados anteriores pode economizar muito tempo. A regra que usei como exemplo (os tipos de operandos devem ser exatamente iguais aos que acabei de ver) é comum, mas o DLR também suporta outras regras.

A segunda parte de uma regra é o código a ser usado quando a regra corresponder e é representada como uma árvore de expressão. Poderia ter sido armazenado como um delegado compilado para chamar, mas manter a representação da árvore de expressão significa que o cache pode ser bastante otimizado. Existem três níveis de cache no DLR: L0, L1 e L2. Os caches armazenam informações de diferentes maneiras e com escopos diferentes. Cada site de chamada tem seu próprio L0 e L1 caches, mas um cache L2 pode ser compartilhado entre vários sites de chamadas semelhantes, como mostrado em figura 14.3.

O conjunto de locais de chamada que compartilham um cache L2 é determinado por seus ligantes – cada um O fichário possui um cache L2 associado a ele. O compilador (ou o que quer que esteja criando o chamar sites) decide quantos fichários deseja usar. Ele só pode usar um fichário para vários sites de chamada que representam código muito semelhante, onde, se o contexto for o mesmo no momento da execução, os sites de chamada deverão ser executados da mesma maneira. Na verdade, o compilador C# não usa esse recurso - ele cria um novo fichário para cada local de chamada,<sup>4</sup> portanto não há muita diferença entre os caches L1 e L2 para desenvolvedores C#. Genuinamente dinâmico linguagens, como IronRuby e IronPython, fazem mais uso dele.

Os próprios caches são executáveis, o que demora um pouco para ser entendido. O C# compilador gera código para simplesmente executar o cache L0 do site de chamada (que é um delegado acessado através da propriedade Target). É isso! O cache L0 tem uma única regra,



**Figura 14.3 Relacionamentos entre caches dinâmicos e sites de chamada**

<sup>4</sup> Muitas informações são específicas para um local de chamada específico, pois as regras vinculativas serão diferentes dependendo coisas como de qual classe ele está sendo chamado.

que verifica quando é chamado. Se a regra corresponder, ela executa o comportamento associado. Se a regra não corresponder (ou se esta for a primeira chamada, então não tem nem uma regra), ele chama o cache L1 , que por sua vez chama o cache L2 . Se o cache L2 não consegue encontrar nenhuma regra correspondente, ele pede ao receptor ou ao fichário para resolver a chamada. O os resultados são então colocados no cache para a próxima vez.

No caso do nosso trecho anterior, a parte de execução seria algo como esse:

```
callSite.Target(callSite, d, 10);
```

Os caches L1 e L2 examinam suas regras de maneira bastante padronizada – cada um tem uma coleção de regras e é perguntado a cada regra se ela corresponde ou não. O cache L0 é um pouco diferente. As duas partes do seu comportamento (verificar a sua regra e delegar o cache L1 ) são combinados em um único método que é então compilado JIT . Atualizando o cache L0 consiste em reconstruir o método a partir da nova regra.

O resultado de tudo isso é que sites de chamada típicos que veem contextos semelhantes repetidamente são muito rápidos; o mecanismo de despacho é tão enxuto quanto você poderia fazer se você você mesmo codificou os testes manualmente. É claro que isto tem de ser ponderado em relação ao custo de todos a geração dinâmica de código envolvida, mas o cache multinível é complicado precisamente porque tenta alcançar um equilíbrio entre vários cenários.

Agora que você conhece um pouco sobre o maquinário do DLR, será capaz de entender entenda o que o compilador C# faz para colocar tudo em movimento.

#### 14.4.3 Como o compilador C# lida com dinâmica

As principais tarefas do compilador C# quando se trata de código dinâmico são descobrir quando comportamento dinâmico é necessário e capturar todo o contexto necessário para que o fichário e o receptor possuem informações suficientes para resolver a chamada em tempo de execução.

##### **SE USA DINÂMICO, É DINÂMICO!**

Uma situação é obviamente dinâmica: quando o alvo de uma chamada de membro é dinâmico. O compilador não tem como saber como isso será resolvido. Pode ser um processo verdadeiramente dinâmico objeto que executará a resolução em si ou pode acabar com o fichário C# resolvendo-o com reflexão posteriormente. De qualquer forma, não há oportunidade para a chamada ser resolvida estaticamente.

Mas quando o valor dinâmico está sendo usado como *argumento* para a chamada, existem algumas situações em que você *pode* esperar que a chamada seja resolvida estaticamente - principalmente se houver uma sobrecarga adequada que tenha um tipo de parâmetro dinâmico. A regra é que se qualquer parte de uma chamada é dinâmica, a chamada se torna dinâmica e resolverá a sobrecarga com o tipo de tempo de execução do valor dinâmico. A listagem a seguir demonstra isso usando um método com duas sobrecargas e invocando-o de várias maneiras diferentes.

**Listagem 14.16 Experimentando sobrecarga de métodos e valores dinâmicos**

```
static void Executar(string x) {
    Console.WriteLine("Sobrecarga de string");
}

static void Executar(dinâmico x) {
    Console.WriteLine("Sobrecarga dinâmica");
}
...
texto dinâmico = "texto";
Executar(texto); número
dinâmico = 10;
Executar(número);
```

← Imprime "Sobrecarga de string"

← Imprime "Sobrecarga dinâmica"

Ambas as chamadas para Executar são vinculadas dinamicamente. No tempo de execução, eles são resolvidos usando os tipos dos valores reais, ou seja, string e int. O parâmetro do tipo dynamic é tratado como se tivesse sido declarado com o tipo object em todos os lugares, exceto dentro do próprio método — se você observar o código compilado, verá que é *um* parâmetro do tipo object, apenas com um atributo extra aplicado. Isso também significa que você não pode ter dois métodos cujas assinaturas diferem apenas pelos tipos de parâmetros dinâmicos/objetos .

Esse é um exemplo de resolução de chamadas de método, mas há muitas outras expressões a serem consideradas. Às vezes a situação não é tão simples como eu fiz você acreditar...

**É DINÂMICO... EXCETO QUANDO NÃO É**

Quando introduzi a dinâmica no 14.2, tive que tomar cuidado para não generalizar muito, porque há exceções para quase todas as regras. Embora você deva saber sobre isso, não precisa se preocupar com eles – é improvável que lhe causem problemas.

Vamos tirá-los do caminho rapidamente.

***Conversões entre tipos CLR e dinâmicos***

conversões entre tipos CLR e dinâmicos são restritas da mesma forma que não é possível converter de *todos* os tipos CLR para objeto; as exceções são tipos como ponteiros e System.TypedReference. Dado que a dinâmica é apenas um objeto no nível CLR , não é surpreendente que esses tipos sejam excluídos.

Você também deve ter notado que escrevi sobre uma conversão “de uma expressão do tipo dinâmico” para um tipo CLR , não uma conversão do tipo dinâmico em si. Essa sutileza ajuda durante a inferência de tipos e outras situações que precisam considerar conversões implícitas entre tipos; em geral, a vida fica desagradável quando existem dois tipos com conversões implícitas nos dois sentidos. Basicamente limita as situações em que a conversão é considerada. Por exemplo, considere esta matriz digitada implicitamente:

```
dinâmico d = 0; string
x = "texto"; var matriz = novo[]
{ d, x };
```

Qual deve ser o tipo inferido de array? Se houvesse uma conversão implícita de dinâmico para string, então poderia ser string[] ou dinâmico[], então você acabaria com ambiguidade e um erro em tempo de compilação. Mas como a conversão só existe a partir de uma expressão dinâmica, o compilador vê uma conversão de string para dinâmica, mas não a de outra forma, e array é do tipo dinâmico[]. Provavelmente é melhor não se preocupar com isso sutiliza, a menos que você esteja tentando trabalhar em um cenário específico com a especificação ao seu lado.

#### ***Expressões que usam dinâmica nem sempre são avaliadas dinamicamente***

Existem alguns casos em que o CLR é bastante capaz de avaliar uma expressão usando os caminhos normais de execução estática, mesmo se uma das subexpressões for dinâmica. Para por exemplo, considere o operador as:

```
dinâmico d = GetValueDynamically();
string x = d como string;
```

Não há nada que possa acontecer dinamicamente aqui – ou o valor de d é uma referência a uma string ou não é. As conversões definidas pelo usuário não são aplicadas quando o operador as é usado, portanto, o compilador C# pode usar exatamente o mesmo IL que usaria se a variável eram do tipo objeto.

#### ***Expressões avaliadas dinamicamente nem sempre são do tipo dinâmico***

Em alguns casos, o compilador não sabe exatamente como avaliará uma expressão, mas conhece o tipo exato do resultado (assumindo que uma exceção não seja lançada). Para Por exemplo, considere fazer uma chamada de construtor usando um valor dinâmico como argumento:

```
dinâmico d = GetValueDynamically();
SomeType x = new SomeType(d);
```

A própria chamada do construtor deve ser avaliada dinamicamente. Pode haver diversas sobrecargas a serem resolvidas em tempo de execução, mas o resultado sempre será um SomeType referência. A atribuição a x pode, portanto, acontecer sem uma conversão dinâmica.

Existem alguns outros casos como este; usando um índice de array dinâmico em um estaticamente array digitado só pode resultar em um valor do tipo de elemento do array, por exemplo. Mas você não deve presumir que isso sempre acontecerá onde você espera. Você poderia ter várias sobrecargas de um método, todas com o mesmo tipo de retorno estático, mas o tipo dessa expressão de invocação de método ainda será dinâmica.

Já basta quando a avaliação dinâmica *não* acontece ou não resulta em um valor dinâmico — vamos voltar às situações em que isso acontece e ver o que o C# compilador faz para fazer tudo funcionar.

#### ***CRIANDO SITES DE CHAMADA E BINDERS***

Você não precisa saber os detalhes exatos do que o compilador faz com dinâmica expressões para usá-las, mas pode ser instrutivo ver o que o compilado código se parece. Em particular, se você precisar descompilar seu código por qualquer outro motivo, você não ficará surpreso com a aparência das partes dinâmicas. Minha ferramenta preferida para esse tipo de trabalho é o Reflector (veja <http://mng.bz/pMXJ>), mas você poderia usar ildasm se você quiser ler o IL diretamente.

Veremos apenas um único exemplo — tenho certeza de que poderia preencher um capítulo inteiro examinando os detalhes da implementação, mas a idéia é apenas dar a você a essência do que o compilador está fazendo. Se você achar este exemplo interessante, você pode experimentar mais por conta própria. Apenas lembre-se de que os detalhes exatos são específicos da implementação; eles podem mudar em versões futuras do compilador, desde que o comportamento seja equivalente.

Aqui está o trecho de exemplo, que existe em um método Main da maneira normal para Snippy:

```
string text = "texto para recortar"; startIndex
dinâmico = 2; string substring =
text.Substring(startIndex);
```

Muito simples, certo? Mas na verdade ele contém duas operações dinâmicas – uma para chamar Substring e outra (implícita) para converter dinamicamente o resultado (que é apenas dinâmico em tempo de compilação) em uma string. A Listagem 14.17 mostra o código descompilado para a classe Snippet.5 Omiti a própria declaração de classe e o construtor implícito sem parâmetros para economizar espaço e reformatei o código com menos espaços em branco pelo mesmo motivo.

#### Listagem 14.17 Os resultados da compilação de código dinâmico

```
[CompilerGenerated] private
static class <Main>o__SiteContainer0 { public static CallSite<Func<CallSite,
object, string>> <>p__Site1; public static CallSite<Func<CallSite, string, object, object>> <>p__Site2;
}

privado estático void Main() {
    string text = "texto para recortar"; objeto
    startIndex = 2; if
        (<Principal>o__SiteContainer0.<>p__Site1 == null) {
            <Principal>o__SiteContainer0.<>p__Site1 =
                CallSite<Func<CallSite, objeto, string>>.Create( new
                    CSharpConvertBinder(typeof(string),
                    CSharpConversionKind.ImplicitConversion, false));
        } if (<Main>o__SiteContainer0.<>p__Site2 == nulo) {
            <Principal>o__SiteContainer0.<>p__Site2 =
                CallSite<Func<CallSite, string, objeto, objeto>>.Create( new
                    CSharpInvokeMemberBinder(CSharpCallFlags.None, "Substring", typeof(Snippet),
                    null, new CSharpArgumentInfo[] { new CSharpArgumentInfo(
                        Preserva o tipo de texto
                        E
                        CSharpArgumentInfoFlags.UseCompileTimeType, null),
                        novo CSharpArgumentInfo(
                            CSharpArgumentInfoFlags.None, null ))));
        }
    } string substring =
```

- B Armazenamento de locais de chamada
- C Cria site de chamada de conversão
- D Cria site de chamada de substring
- E Preserva o tipo de texto
- F Invocação de ambas as chamadas

<sup>5</sup> Só para lembrar, Snippet é a classe gerada pelo Snippy automaticamente.

```

<Principal>o__SiteContainer0.<>p__Site1.Target.Invoke( <Principal>o__SiteContainer0.<>p__Site1,
<Principal>o__SiteContainer0.<>p__Site2.Target.Invoke(
}

<Principal>o__SiteContainer0.<>p__Site2, text, startIndex));
}

```

Não sei sobre você, mas estou feliz por nunca ter que escrever ou encontrar códigos como esse, a não ser com o propósito de aprender sobre o que está acontecendo. Porém, não há nada de novo nisso – o código gerado para blocos iteradores, árvores de expressão e funções anônimas também pode ser bastante horrível.

Uma classe estática aninhada é usada para armazenar todos os sites de chamada **B** para o método, pois eles só precisam ser criados uma vez. (Se eles fossem criados todas as vezes, o cache seria inútil!) É possível que os sites de chamada *sejam* criados mais de uma vez devido ao multithreading, mas se isso acontecer, será apenas um pouco ineficiente e significa que a criação lenta é alcançada sem nenhum bloqueio. Realmente não importa se uma instância do site de chamada for substituída por outra. Cada método que usa ligação dinâmica possui um contêiner de site separado; esse *deve* ser o caso dos métodos genéricos, pois o site da chamada precisa variar com base nos argumentos de tipo. Outra implementação de compilador poderia optar por usar um contêiner de site para todos os métodos não genéricos, um para todos os métodos genéricos com um único parâmetro de tipo e assim por diante.

Após a criação dos sites de chamada (**C** e **D**), eles são invocados. A chamada de Substring é invocada primeiro (leia o código da parte mais interna da instrução para fora) e então a conversão é invocada no resultado **F**. Neste ponto, você tem um valor digitado estaticamente novamente, para que possa atribuí-lo à substring variável.

Gostaria de destacar mais um aspecto do código: a forma como algumas informações do tipo estático são preservadas no site de chamada. As informações de tipo em si estão presentes na assinatura do delegado usado para o argumento de tipo do site de chamada (`Func<CallSite, string, object, object>`), e um sinalizador no `CSharpArgumentInfo` correspondente indica que essas informações de tipo devem ser usadas no fichário **E**. (Mesmo que este seja o alvo do método, ele é representado como um argumento; os métodos de instância são tratados como métodos estáticos com um primeiro parâmetro implícito disso.) Esta é uma parte crucial para fazer o fichário se comportar como se fosse apenas recompilando seu código em tempo de execução. Vejamos por que isso é tão importante.

#### 14.4.4 O compilador C# fica ainda mais inteligente

O C# 4 permite ultrapassar o limite estático/dinâmico não apenas tendo parte do seu código vinculada estaticamente e parte dinamicamente, mas também combinando as duas ideias em uma única ligação. Ele se lembra de tudo o que precisa saber no site da chamada e, em seguida, mescla essas informações de maneira inteligente com os tipos de valores dinâmicos em tempo de execução.

##### PRESERVANDO O COMPORTAMENTO DO COMPILADOR NO MOMENTO DE EXECUÇÃO

O modelo ideal para descobrir como o fichário deve se comportar é imaginar que, em vez de ter um valor dinâmico em seu código-fonte, você tem um valor exatamente igual a

tipo certo: o tipo do valor real em tempo de execução.<sup>6</sup> Isso se aplica *apenas* a valores dinâmicos dentro da expressão; quaisquer tipos conhecidos em tempo de compilação ainda serão usados para pesquisas, como resolução de membros. Darei dois exemplos de onde isso faz diferença.

A listagem a seguir mostra um método simples sobre carregado em um único tipo.

#### Listagem 14.18 Resolução de sobrecarga dinâmica dentro de um único tipo

```
static void Execute(dinâmico x, string y) {
    Console.WriteLine("dinâmico, string");
}

static void Execute(dinâmico x, objeto y) {
    Console.WriteLine("dinâmico, objeto");
}
...
objeto texto = "texto"; dinâmico d =
10;
Executar(d, texto);
```



Imprime "dinâmico, objeto"

A variável importante aqui é o texto. Seu tipo *em tempo de compilação* é objeto, mas em *tempo de execução* seu valor é uma referência de string. A chamada para Execute é dinâmica porque você está usando a variável dinâmica d como um dos argumentos, mas a resolução de sobrecarga usa o tipo de texto estático, portanto o resultado é dinâmico, objeto. Se a variável de texto tivesse sido declarada como dinâmica, ela teria usado a outra sobrecarga.

A próxima listagem é semelhante, mas desta vez o que importa é o receptor da chamada.

#### Listagem 14.19 Resolução de sobrecarga dinâmica dentro de uma hierarquia de classes

```
classe Base
{
    public void Executar(objeto x) {
        Console.WriteLine("objeto");
    }
}

classe Derivada: Base
{
    public void Executar(string x) {
        Console.WriteLine("string");
    }
}
...
Receptor base = new Derived(); dinâmico d =
"texto"; receptor.Execute(d);
```



Imprime "objeto"

<sup>6</sup> É um pouco mais complicado que isso – e se o tipo real for interno a outro assembly? Você não gostaria que isso fosse usado como argumento de tipo de um método genérico por meio de inferência de tipo, por exemplo. O arquivo tem a noção de “melhor tipo acessível” com base no contexto de chamada e no tipo real.

Na listagem 14.19, o tipo de receptor é Derived em tempo de execução, então você poderia esperar que a sobrecarga introduzida em Derived fosse chamada. Mas o tipo de receptor em tempo de compilação é Base, então o arquivo restringe o conjunto de métodos que considera apenas aqueles que *estariam* disponíveis se você estivesse vinculando o método estaticamente.

Apesar de todas essas decisões que devem ser tomadas posteriormente, algumas verificações em tempo de compilação estão disponíveis, mesmo para códigos que serão totalmente vinculados em tempo de execução.

#### ERROS EM TEMPO DE COMPILAÇÃO PARA CÓDIGO DINÂMICO

Como eu disse no início deste capítulo, uma das desvantagens da digitação dinâmica é que alguns erros que normalmente seriam detectados pelo compilador são adiados até o tempo de execução, momento em que uma exceção é lançada. Há muitas situações em que o compilador precisa apenas esperar que você saiba o que está fazendo, mas quando puder *ajudá-lo*, ele o fará.

O exemplo mais simples disso é quando você tenta chamar um método com um receptor de tipo estaticamente (ou um método estático) e nenhuma das sobrecargas pode ser válida, qualquer que seja o tipo que o valor dinâmico tenha no tempo de execução. A listagem a seguir mostra três exemplos de chamadas inválidas, duas das quais são capturadas pelo compilador.

#### Listagem 14.20 Capturando erros em chamadas dinâmicas em tempo de compilação

```
string text = "me corte"; guid dinâmico =
    Guid.NewGuid(); texto.Substring(guid);
    text.Substring("x", guid);
    text.Substring(guid, guid);
```

Aqui você tem três chamadas para `string.Substring`. O compilador conhece o conjunto exato de possíveis sobrecargas, porque conhece o tipo de texto estaticamente. Ele não reclama na primeira chamada, porque não consegue dizer que tipo de `guid` será – se for um número inteiro, tudo ficará bem. Mas as duas linhas finais geram erros – não há sobrecargas que usam uma `string` como primeiro argumento e não há sobrecargas com três parâmetros. O compilador pode *garantir* que eles falharão em tempo de execução, portanto, é razoável que falhem em tempo de compilação.

Um exemplo um pouco mais complicado é a inferência de tipos. Se um valor dinâmico for usado para inferir um argumento de tipo em uma chamada para um método genérico, o argumento de tipo real não será conhecido até o tempo de execução e nenhuma validação poderá ocorrer antecipadamente. Mas qualquer argumento de tipo que seria inferido sem o uso de valores dinâmicos pode causar falha na inferência de tipo em tempo de compilação. A listagem a seguir mostra um exemplo disso.

#### Listagem 14.21 Inferência de tipo genérico com valores mistos estáticos e dinâmicos

```
void Execute<T>(T primeiro, T segundo, string outro) onde T : struct
{
}
...
guid dinâmico = Guid.NewGuid(); Executar(10,
0, guia);
```

```
Executar(10, falso, guid);
Execute("olá", "olá", guid);
```

Novamente, a primeira chamada é compilada, mas falharia em tempo de execução. A segunda chamada não será compilada porque T não pode ser int e bool e não há conversões entre os dois. A terceira chamada não será compilada porque infere-se que T seja uma string, o que viola a restrição de que deve ser um tipo de valor.

O compilador é conservador: ele só falhará com um erro se puder dizer que algum código não pode ter sucesso e só executa testes relativamente simples nesse aspecto. Existem algumas situações em que pode ser óbvio (e provável) para um ser humano que o código não funcionará, mas onde o compilador permite a passagem do código. É claro que, se uma determinada linha de código nunca funcionar, então um único teste de unidade que a executa falhará; portanto, a natureza simplista da verificação do compilador não importa se você tiver uma boa cobertura de código. Pense nisso como um bônus nos casos em que detecta um problema.

Isso cobre os pontos mais importantes em termos do que o compilador *pode* fazer por você. Mas você não pode usar a dinâmica em todos os lugares. Existem limitações, algumas das quais são dolorosas, mas a maioria é bastante obscura.

#### 14.4.5 Restrições ao código dinâmico

Você pode usar *principalmente* dinâmico sempre que normalmente usaria um nome de tipo e, em seguida, escrever C# normal. Mas existem algumas exceções. Esta não é uma lista exaustiva, mas abrange os casos que você provavelmente encontrará.

##### MÉTODOS DE EXTENSÃO NÃO SÃO RESOLVIDOS DINAMICAMENTE

O compilador emite *parte* do contexto da chamada para o site da chamada, como você já viu. Em particular, o site conhece os tipos estáticos que o compilador conhecia. Mas nas versões atuais do C#, ele *não* sabe quais diretivas using ocorreram no arquivo de origem que contém a chamada. Isso significa que ele não sabe quais métodos de extensão estão disponíveis em tempo de execução.

Isso não significa apenas que você não pode chamar métodos de extensão *em* valores dinâmicos — significa que você também não pode passá-los para métodos de extensão como argumentos. Existem duas soluções alternativas, ambas sugeridas de forma útil pelo compilador. Se você souber qual sobrecarga deseja, poderá converter o valor dinâmico para o tipo correto na chamada do método. Caso contrário, supondo que você saiba qual classe estática contém o método de extensão, você poderá chamá-lo como um método estático normal. A listagem a seguir mostra um exemplo de chamada com falha e ambas as soluções alternativas.

##### Listagem 14.22 Chamando métodos de extensão com argumentos dinâmicos

```
tamanho dinâmico = 5;
var números = Enumerable.Range(10, 10); var erro =
números.Take(tamanho); var solução alternativa1
= números.Take((int) tamanho); var workaround2 =
Enumerable.Take(números, tamanho);
```

 Erro em tempo de compilação

Ambas as abordagens funcionarão se você quiser chamar o método de extensão com o valor dinâmico como o valor implícito também , embora a conversão se torne bastante feia nesse caso.

#### DELEGAR RESTRIÇÕES DE CONVERSÃO COM DINÂMICO

O compilador precisa saber o tipo exato de delegado (ou expressão) envolvido ao converter uma expressão lambda, um método anônimo ou um grupo de métodos. Você não pode atribuir nenhum deles a um delegado simples ou a uma variável de objeto sem conversão, e o mesmo se aplica ao dinâmico. Mas um elenco é suficiente para manter o compilador feliz. Isto pode ser útil em algumas situações se você quiser executar o delegado dinamicamente mais tarde. Você também pode usar um delegado com um tipo dinâmico como um de seus parâmetros, se isso for útil.

A Listagem 14.23 mostra alguns exemplos que serão compilados e outros que não.

#### Listagem 14.23 Tipos dinâmicos e expressões lambda

```
badMethodGroup dinâmico = Console.WriteLine; goodMethodGroup
dinâmico = (Action<string>) Console.WriteLine;

dinâmico badLambda = y => y + 1; goodLambda
dinâmico = (Func<int, int>) (y => y + 1);

dinâmico muitoDynamic = (Func<dinâmico, dinâmico>) (d => d.SomeMethod());
```

Observe que, devido à forma como a resolução de sobrecarga funciona, isso significa que você não pode usar expressões lambda em chamadas vinculadas dinamicamente sem conversão - mesmo que o único método que possa ser invocado tenha um tipo de delegado conhecido em tempo de compilação. Por exemplo, este código não será compilado:

```
Método void(Ação<string> ação, valor da string) {
    ação(valor);
}
...
texto dinâmico = "erro"; Método(x =>
Console.WriteLine(x, texto);
```

 **Erro em tempo**

**de compilação** Vale ressaltar que nem tudo está perdido em termos de LINQ e interação dinâmica . Você pode ter uma coleção fortemente tipada com um tipo de elemento dinâmico; nesse ponto, você ainda pode usar métodos de extensão, expressões lambda e até mesmo expressões de consulta. A coleção pode conter objetos de diferentes tipos e eles se comportarão adequadamente em tempo de execução, conforme mostrado na listagem a seguir.

#### Listagem 14.24 Consultando uma coleção de elementos dinâmicos

```
var lista = new Lista<dinâmica> { 50, 5m, 5d };
var consulta = do número na
lista
    onde número> 4
    selecione (número/20) * 10;

foreach (var item na consulta) {
    Console.WriteLine(item);
}
```

Isso imprime 20, 2,50 e 2,5. Dividi deliberadamente por 20 e depois multipliquei por 10 para mostrar a diferença entre decimal e duplo: o tipo decimal controla a precisão sem normalizar, e é por isso que 2,50 é exibido em vez de 2,5. O primeiro valor é um número inteiro, portanto a divisão inteira é usada; daí o valor de 20 em vez de 25.

#### **CONSTRUTORES E MÉTODOS ESTÁTICOS**

Você pode chamar construtores e métodos estáticos dinamicamente, no sentido de que pode especificar argumentos dinâmicos, mas não pode resolver um construtor ou método estático em relação a um tipo dinâmico. Simplesmente não há como especificar a que tipo você se refere.

Se você se deparar com uma situação em que *deseja* fazer isso dinamicamente de alguma forma, tente pensar em maneiras de usar métodos de instância, como criando um tipo de fábrica. Você pode descobrir que pode obter o comportamento dinâmico desejado usando polimorfismo ou interfaces simples, mas dentro da digitação estática.

#### **DECLARAÇÕES DE TIPO E PARÂMETROS DE TIPO GENÉRICO**

Você não pode declarar que um tipo possui uma classe base dinâmica. Você também não pode usar dinâmico em uma restrição de parâmetro de tipo ou como parte do conjunto de interfaces que seu tipo implementa. Você *pode* usá-lo como um argumento de tipo para uma classe base ou quando estiver especificando uma interface para uma declaração de variável. Por exemplo, estas declarações são inválidas:

```
ÿ classe BaseTypeOfDynamic: dinâmico ÿ classe  
DynamicTypeConstraint<T> onde T: dinâmico ÿ classe  
DynamicTypeConstraint<T> onde T: Lista<dinâmico> ÿ classe  
DynamicInterface: IEnumerable<dinâmico>
```

Mas estes são válidos:

```
ÿ classe GenericDynamicBaseClass : List<dynamic> ÿ  
IEnumerable<dynamic> variável;
```

A maioria dessas restrições em torno dos genéricos é o resultado do tipo dinâmico não existir realmente como um tipo .NET . O CLR não sabe disso – qualquer uso em seu código é traduzido em objetos com DynamicAttribute aplicado apropriadamente. (Para tipos como List<dynamic> ou Dictionary<string, dynamic>, o atributo indica exatamente quais partes do tipo são dinâmicas.) DynamicAttribute só é aplicado quando a natureza dinâmica precisa ser representada em metadados; variáveis locais não requerem o atributo, pois nada precisa inspecioná-las após a compilação para detectar sua natureza dinâmica.

Todo o comportamento dinâmico é alcançado através da inteligência do compilador ao decidir como o código-fonte deve ser traduzido e da inteligência *da biblioteca* em tempo de execução. Essa equivalência entre dinâmico e objeto é evidente em vários lugares, mas talvez seja mais óbvia se você observar typeof(dynamic) e typeof(object), que retornam a mesma referência. Em geral, se você achar que não consegue fazer o que deseja com o tipo dinâmico , lembre-se de como ele se parece com o CLR e veja se isso explica o problema. Pode não sugerir uma solução, mas pelo menos você ficará melhor em prever o que funcionará com antecedência.

Esses são todos os detalhes que darei sobre como o C# 4 trata a dinâmica, mas há outro aspecto da imagem da digitação dinâmica que realmente precisamos observar para obter uma visão completa do tópico: reagir dinamicamente. Uma coisa é poder *chamar* código dinamicamente, outra é poder *responder* dinamicamente a essas chamadas.

É claro que, se você estiver apenas chamando código de terceiros dinamicamente — ou mesmo usando técnicas como despacho múltiplo, mostradas anteriormente — você não precisa se preocupar com isso. Eu entendo se você acha que já está farto de digitação dinâmica, pelo menos por enquanto; já cobrimos muito terreno. Você pode pular a próxima seção com segurança e voltar a ela em outra ocasião — nada no resto do livro depende disso. Por outro lado, é divertido.

## 14.5 Implementando comportamento dinâmico

A linguagem C# não oferece nenhuma ajuda específica na implementação de comportamento dinâmico, mas a estrutura oferece. Um tipo precisa implementar `IDynamicMetaObjectProvider` para reagir dinamicamente, mas há duas implementações integradas que podem eliminar muito trabalho em muitos casos. Veremos ambos, bem como uma implementação *muito* simples de `IDynamicMetaObjectProvider`, apenas para mostrar o que está envolvido.

Essas três abordagens são realmente diferentes e começaremos com a mais simples delas: `ExpandoObject`.

### 14.5.1 Usando `ExpandoObject`

`System.Dynamic.ExpandoObject` parece uma fera engraçada à primeira vista. Seu único construtor público não possui parâmetros. Ele não possui métodos públicos, a menos que você conte a implementação explícita de várias interfaces – principalmente, `IDynamicMetaObject-Provider` e `IDictionary<string, object>`. (As outras interfaces que ele implementa são todas devido à extensão de `IDictionary<,>` outras interfaces.) Ah, e é selado, então não é uma questão de derivar dele para implementar um comportamento útil. Não, `ExpandoObject` só é útil se você se referir a ele via dinâmica ou uma das interfaces que ele implementa.

#### CONFIGURANDO E RECUPERANDO PROPRIEDADES INDIVIDUAIS

A interface do dicionário dá uma dica sobre sua finalidade – é basicamente uma forma de armazenar objetos por meio de nomes. Mas esses nomes também podem ser usados como propriedades por meio de digitação dinâmica. A listagem a seguir mostra isso funcionando nos dois sentidos.

#### Listagem 14.25 Armazenando e recuperando valores com `ExpandoObject`

```
expansão dinâmica = new ExpandoObject(); IDictionary<string,
objeto> dicionário = expando; expando.First = "valor definido dinamicamente";
Console.WriteLine(dicionário["Primeiro"]);

dicionário["Segundo"] = "valor definido com dicionário";
Console.WriteLine(expandindo.Second);
```

A Listagem 14.25 apenas usa strings como valores por conveniência — você pode usar qualquer objeto, como seria de esperar com um `IDictionary<string, object>`. Se você especificar um delegado como o valor, poderá chamar o delegado como se fosse um método no expando, como segue.

#### Listagem 14.26 Falsificação de métodos em um `ExpandoObject` com delegados

```
expansão dinâmica = new ExpandoObject(); expando.AddOne =
(Func<int, int>)(x => x + 1); Console.WriteLine(expandindo.AddOne(10));
```

Embora pareça um acesso de método, você também pode pensar nisso como um acesso de propriedade que retorna um delegado e, em seguida, uma invocação do delegado. Se você criou uma classe de tipo estático com uma propriedade `AddOne` do tipo `Func<int, int>`, você poderia usar exatamente a mesma sintaxe. O C# gerado para chamar `AddOne` de fato usa uma operação de “invocação de membro” em vez de tentar acessá-lo como uma propriedade e depois invocá-lo, mas `ExpandoObject` sabe o que fazer. Você também pode acessar a propriedade para recuperar o delegado, se desejar.

Vamos passar para um exemplo um pouco maior, embora ainda não vamos fazer nada particularmente complicado.

#### CRIANDO UMA ÁRVORE DOM

Criaremos uma árvore de expansos que espelha uma árvore XML DOM. Esta é uma implementação bastante rudimentar, projetada para simplicidade de demonstração, em vez de uso no mundo real. Em particular, pressupõe que não temos nenhum namespace XML com que nos preocupar.

Cada nó na árvore tem dois pares nome/valor que sempre estarão presentes: `XElement`, que armazena o elemento LINQ to XML original usado para criar o nó, e `ToXml`, que armazena um delegado que apenas retorna o nó como uma string XML. Você poderia simplesmente chamar `node.XElement.ToString()`, mas desta forma dá outro exemplo de como os delegados trabalham com `ExpandoObject`. Um ponto a ser mencionado é que usaremos `ToXml` em vez de `ToString`, porque definir a propriedade `ToString` em um expando *não* substitui o método `ToString` normal. Isso pode levar a erros confusos, então usaremos um nome diferente.

A parte interessante não são os nomes fixos; são aqueles que dependem do XML real. Ignoraremos completamente os atributos, mas quaisquer *elementos* no XML original que sejam filhos do elemento original serão acessíveis por meio de propriedades de mesmo nome. Por exemplo, considere o seguinte XML:

```
<raiz>
  <ramo>
    <folha /> </
  branch> </root>
```

Assumindo uma variável dinâmica chamada `root` representando o elemento `raiz`, você poderia acessar o nó `folha` com dois acessos simples à propriedade, que podem ocorrer em um único declaração:

```
folha dinâmica = root.branch.leaf;
```

Se um elemento ocorrer mais de uma vez dentro de um pai, a propriedade se referirá ao primeiro elemento com esse nome. Para tornar os demais elementos acessíveis, cada elemento também será exposto por meio de uma propriedade utilizando o nome do elemento com sufixo *List*, que retorna um *List<dynamic>* contendo cada um dos elementos com aquele nome na ordem do documento. Em outras palavras, o acesso também poderia ser representado como *root.branchList[0].leaf*, ou talvez *root.branchList[0].leafList[0]*. Observe que o indexador aqui está sendo aplicado à lista – você não pode definir seu próprio comportamento de indexador para expandos.

A implementação de tudo isso é extremamente simples, com um único método recursivo fazendo todo o trabalho, conforme mostrado na listagem a seguir.

#### Listagem 14.27 Implementando uma conversão XML DOM simplista com *ExpandoObject*

```

público estático dinâmico CreateDynamicXml (elemento XElement)
{
    expansão dinâmica = new ExpandoObject(); expando.XElement
    = elemento; expando.Xml =
    (Func<string>)element.ToString();

    IDictionary<string, objeto> dicionário = expando; foreach (XElement subElement
    em element.Elements()) {

        subNode dinâmico = CreateDynamicXml(subElement); string nome =
        subElement.Name.LocalName; string nomedelista = nome + "Lista"; if
        (dicionário.ContainsKey(nome)) {

            ((Lista<dinâmica>) dicionário[nomedelista]).Add(subNode);
        }
        outro
        {
            dicionário[nome] = subNode;
            dicionário[nomedelista] = new Lista<dinâmica> ( subNode );
        }
    }

    } return expansão;
}

```

Sem o tratamento da lista, a listagem 14.27 teria sido ainda mais simples. Você define as propriedades *XElement* e *ToXml* dinamicamente (**B** e **C**), mas não pode fazer isso para os elementos ou suas listas, porque não conhece os nomes em tempo de compilação.<sup>7</sup> Em vez disso, você usa a representação de dicionário (**E** e **F**), que também permite verificar facilmente elementos repetidos. Você não pode saber se um expando contém um valor para uma chave específica apenas acessando-o como uma propriedade; qualquer tentativa de acessar uma propriedade que ainda não tenha sido definida resulta em uma exceção. A manipulação recursiva de subelementos é tão simples em código dinâmico quanto seria em código digitado estaticamente; vo-

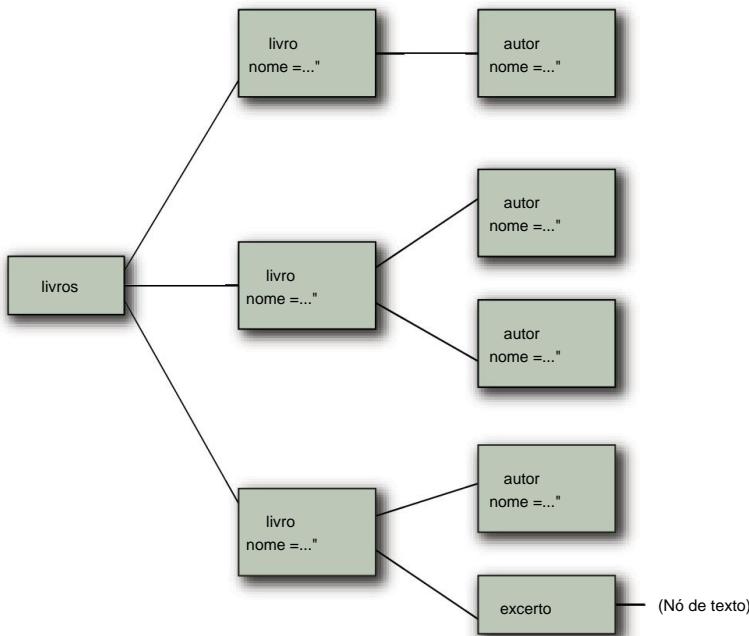
<sup>7</sup> Há uma certa ironia aqui: os nomes que você conhece estaticamente podem ser definidos dinamicamente, mas os nomes que você conhece dinamicamente precisam usar digitação estática.

chame o método **D** recursivamente com cada subelemento, usando seu resultado para preencher as propriedades apropriadas.

Você precisará de algum XML para usar como exemplo, mas é útil representá-lo graficamente e também em seu formato bruto. Usearemos uma estrutura simples representando livros. Cada livro possui um único nome representado como um atributo, podendo ter múltiplos autores, cada um com seu elemento próprio. A Figura 14.4 mostra o arquivo inteiro como uma árvore; o texto a seguir é o XML bruto:

```
<livros>
  <nome do livro="Motores Mortais">
    <author name="Philip Reeve" /> </book> <book
      name="O
      Talismã">
      <nome do autor="Stephen King" /> <nome do
        autor="Peter Straub" /> </book> <nome do
        livro="Rose">

      <author name="Holly Webb" /> <excerpt>
        Rose estava
        se lembrando das ilustrações de Contos Moralmente Instrutivos para
        o Berçário. </excerpt> </book> </books>
```



**Figura 14.4** Estrutura em árvore do arquivo XML de amostra

A listagem a seguir mostra um breve exemplo de como o código expando pode ser usado com este documento XML , incluindo as propriedades ToXml e XElement . O arquivo books.xml contém o documento XML mostrado na figura.

#### Listagem 14.28 Usando um DOM dinâmico criado a partir de expandos

```
XDocument doc = XDocument.Load("livros.xml"); raiz dinâmica =
CreateDynamicXml(doc.Root);
Console.WriteLine(root.book.author.ToXml());
Console.WriteLine(root.bookList[2].excerpt.XElement.Value);
```

A Listagem 14.28 não deve trazer surpresas, a menos que você não esteja familiarizado com a propriedade XElement .Value , que simplesmente retorna o texto dentro de um elemento. A saída da listagem é a esperada:

```
<author name="Philip Reeve" /> Rose estava
se lembrando das ilustrações de Morally Instructive Tales for the
Nursery.
```

Tudo isso é bom, mas existem alguns problemas com o DOM que usamos:

- ÿ Ele não lida com atributos.
- ÿ Duas propriedades são necessárias para cada nome de elemento, devido à necessidade de representar listas enviadas.
- ÿ Seria bom substituir ToString() em vez de adicionar uma propriedade extra. ÿ O resultado é mutável – não há nada que impeça o código de adicionar suas próprias propriedades posteriormente.
- ÿ Embora o expando seja mutável, ele não refletirá nenhuma alteração no XElement subjacente (que também é mutável). ÿ Existem muitas oportunidades para nomear conflitos, como um nó contendo elementos Foo e FooList, ou elementos chamados XElement ou ToXml.
- ÿ A árvore inteira é preenchida antecipadamente, o que é muito trabalhoso se você precisar apenas de um poucos nós.

A correção desses problemas requer mais controle do que apenas definir propriedades. Digite DynamicObject.

### 14.5.2 Usando DynamicObject

DynamicObject é uma maneira mais poderosa de interagir com o DLR do que usar ExpandoObject, mas é muito mais simples do que implementar IDynamicMetaObjectProvider . Embora *na verdade* não seja uma classe abstrata, você realmente precisa derivar dela para fazer algo útil — e o único construtor é protegido, então pode muito bem ser abstrato para todos os propósitos práticos.

Existem quatro tipos de métodos que você pode querer substituir:

- ÿ Métodos de invocação TryXXX() , representando chamadas dinâmicas para o objeto ÿ GetDynamicMemberNames(), que pode retornar uma lista dos membros disponíveis

- ÿ Os métodos normais Equals(), GetHashCode() e ToString() , que podem ser substituído como de costume
- ÿ GetMetaObject(), que retorna o metaobjeto usado pelo DLR

Veremos todos, exceto o último, para melhorar a representação XML DOM e discutiremos metaobjetos na próxima seção, quando implementarmos IDynamicMetaObjectProvider do zero. Além disso, pode ser útil criar novos membros em um tipo derivado, mesmo que os chamadores provavelmente utilizem instâncias como valores dinâmicos. Antes de tomarmos qualquer uma dessas medidas, precisaremos de uma classe para acomodar todos esses membros.

#### COMEÇANDO

Como derivamos de DynamicObject em vez de apenas chamar métodos nele, precisamos começar com uma declaração de classe. A listagem a seguir mostra o esqueleto básico que iremos desenvolver.

#### Listagem 14.29 Esqueleto de Dynamic XElement

```
classe pública Dynamic XElement: DynamicObject {
    elemento XElement somente leitura privado;
    private Dynamic XElement (elemento XElement) {
        este.elemento = elemento;
    }
    public static dynamic CreateInstance (elemento XElement) {
        retornar novo Dynamic XElement(elemento);
    }
}
```

**B** XElement que este quebras de instância

**C** Construtor privado impedindo direto Instanciação

**D** Método público para criar Instâncias

A classe Dynamic XElement apenas envolve um XElement B. Esse será todo o estado que você terá, o que por si só é uma decisão de design significativa. No ExpandoObject anterior, você recorreu à sua estrutura e preencheu uma árvore espelhada inteira. Você realmente tinha que fazer isso, porque não seria possível interceptar acessos de propriedade com código personalizado posteriormente. Obviamente, isso é mais caro do que a abordagem Dynamic XElement , onde você só agrupará os elementos da árvore quando realmente for necessário. Além disso, significa que quaisquer alterações no XElement após a criação do expando serão efetivamente perdidas; se você adicionar mais subelementos, por exemplo, eles não aparecerão como propriedades porque não estavam presentes quando você tirou o instantâneo. A abordagem de empacotamento leve está sempre “ativa” – quaisquer alterações feitas na árvore

A desvantagem disso é que você não fornece mais a mesma ideia de identidade que tinha antes. Com o expando, a expressão root.book.author seria avaliada como a mesma referência se você a usasse duas vezes. Usando Dynamic XElement, cada vez que a expressão for avaliada, ela criará novas instâncias para agrupar os subelementos. Você poderia implementar algum tipo de cache inteligente para contornar isso, mas isso poderia acabar ficando muito complicado, muito rapidamente.

Na listagem 14.29 o construtor de DynamicXElement é private C, e há um método estático público para criar instâncias D. O método tem um tipo de retorno dinâmico, porque é assim que você espera que os desenvolvedores usem a classe. Uma pequena alternativa teria sido criar uma classe estática pública separada com um método de extensão para XElement, e para manter o próprio DynamicXElement interno. A classe em si é uma implementação detalhe; não faz muito sentido usá-lo, a menos que você esteja trabalhando dinamicamente.

Com esse esqueleto instalado, você pode começar a adicionar recursos. Começaremos com realmente simples coisas: adicionar métodos e indexadores como se fosse uma classe normal.

#### SUPORTE DYNAMICOBJECT PARA MEMBROS SIMPLES

Quando discutimos expandos, sempre adicionamos dois membros: o ToXml método e a propriedade XElement. Desta vez você não precisa de um novo método para converter o objeto para uma representação de string; você pode substituir o método ToString() normal.

Você também pode fornecer a propriedade XElement como se estivesse escrevendo qualquer outra classe.

Uma das coisas boas do DynamicObject é que quando algum comportamento não funciona precisa ser verdadeiramente dinâmico, você não precisa implementá-lo dinamicamente. Antes do metaobjeto associado usar qualquer um dos métodos TryXXX, ele verifica se o membro já existe como um membro direto do CLR. Se isso acontecer, esse membro será chamado. Isso torna a vida significativamente mais simples.

Teremos também dois indexadores em DynamicXElement, para fornecer acesso aos atributos e substituir as listas de elementos. A listagem a seguir mostra o novo código a ser adicionado ao classe.

#### Listagem 14.30 Adicionando membros não dinâmicos ao DynamicXElement

```
string de substituição pública ToString()
{
    retornar elemento.ToString();
}

XElement público XElement
{
    obter {elemento de retorno; }
}

público XAttribute isto[nome do XName]
{
    obter {retornar elemento.Atributo(nome); }
}

dinâmico público este[índice int]
{
    pegar
    {
        XElement pai = elemento.Parent;
        if (pai == nulo)
        {
            se (índice! = 0)
            {
                lançar novo ArgumentOutOfRangeException();
            }
        }
    }
}
```

**A** Substitui ToString()  
**B** normalmente  
**C** Devoluções envolto Elemento C  
**D** Indexador recuperando um atributo  
**E** Indexador recuperando um Elemento irmão E  
**F** Este é um elemento raiz?

```

        devolva isso;
    }
    Irmão XElement = parent.Elements(element.Name)
        .ElementAt(indice);
    elemento de retorno == irmão? esse
        : novo Dynamic XElement(irmão);
}
}

```

**G** Encontre o  
irmão apropriado

Há uma boa quantidade de código na listagem 14.30, mas a maior parte é simples. Você substitui `ToString()` fazendo proxy da chamada para o `XElement` e, se quiser implementar a igualdade de valores, poderá fazer algo semelhante para `Equals()` e `GetHashCode()`. A propriedade que retorna o elemento subjacente **C** e o indexador para os atributos **D** também são simples, embora seja importante notar que você só precisa usar um `XName` para o parâmetro do indexador de atributos; se você fornecer uma string em tempo de execução, `DynamicObject` se encarregará de chamar a conversão implícita para `XName` para você.

A parte mais complicada do código é entender o que o indexador com o parâmetro `int` **E** deve fazer. Provavelmente é mais fácil explicar isso em termos de uso esperado. A idéia é evitar ter a propriedade de lista extra, fazendo com que um elemento atue tanto como um elemento único quanto como uma lista de elementos filhos com o mesmo nome. A Figura 14.5 mostra o exemplo XML anterior com algumas expressões para alcançar diferentes nós dentro dele.

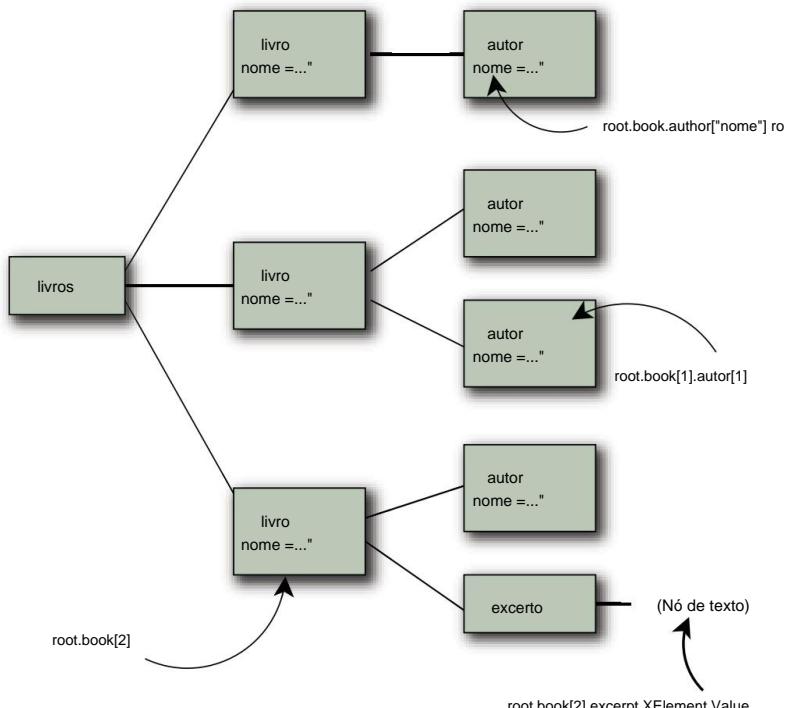


Figura 14.5 Selecionando dados usando `Dynamic XElement`

Depois de entender o que o indexador pretende fazer, a implementação é bastante simples, complicado apenas pela possibilidade de você já estar no topo da árvore F. Caso contrário, você só precisa perguntar ao elemento todos os seus irmãos e depois escolher aquele que lhe foi pedido G.

Até agora não analisamos nada dinâmico, exceto em termos do tipo de retorno de `CreateInstance()` — nenhum desses exemplos funcionará porque você não escreveu o código para buscar subelementos. Vamos consertar isso agora.

## SUBSTITUINDO MÉTODOS TRYXXX

No DynamicObject você responde às chamadas dinamicamente substituindo um dos TryXXX métodos. São 12, representando diferentes tipos de operações, conforme mostrado na tabela 14.1.

**Tabela 14.1** Métodos TryXXX virtuais em DynamicObject

Nome	Tipo de chamada representada (onde x é o objeto dinâmico)
ExperimenteBinaryOperation	Operação binária, como x + y
TryConvert	Conversões, como (Meta) x
TryCreateInstance	Expressões de criação de objetos; nenhum equivalente em C#
TryDeleteIndex	Operação de remoção de indexador; nenhum equivalente em C#
TryDeleteMember	Operação de remoção de bens; nenhum equivalente em C#
TryGetIndex	Getter indexador, como x[10]
TryGetMember	Getter de propriedade, como x.Property
TryInvoke	Invocação direta tratando x como um delegado, como x(10)
TryInvokeMembro	Invocação de um membro, como x.Method()
TrySetIndex	Configurador de indexador, como x[10] = 20
TrySetMember	Configurador de propriedades, como x.Property = 10
ExperimenteUnaryOperation	Operação unária, como !x ou -x

Cada um desses métodos possui um tipo de retorno booleano para indicar se a ligação foi bem-sucedido. Cada um usa um fichário apropriado como primeiro parâmetro e, se a operação tiver argumentos logicamente (por exemplo, os argumentos para um método ou os índices para um indexador), estes são representados como um objeto[]. Finalmente, se a operação puder ter um valor de retorno (que inclui tudo, exceto as operações set e delete), há um parâmetro out do tipo object para capturar esse valor.

O tipo exato de fichário depende da operação; há um fichário diferente tipo para cada uma das operações. Por exemplo, a assinatura completa de TryInvokeMember é

bool virtual público TryInvokeMember (fichário InvokeMemberBinder,  
object[] args, resultado do objeto)

Você só precisa substituir os métodos que representam as operações que você suporta dinamicamente. Nesse caso, você tem propriedades dinâmicas somente leitura (para os elementos), portanto, é necessário substituir TryGetMember(), conforme mostrado na listagem a seguir.

#### Listagem 14.31 Implementando uma propriedade dinâmica com TryGetMember()

substituição pública bool TryGetMember (Fichário GetMemberBinder, resultado do objeto fora)

```
{
    XElement subElement = element.Element(binder.Name); if (subElement != nulo) {
        ← Encontre a primeira correspondência
        ← Subelemento B
        resultado = novo DynamicXElement(subElement);
        retornar verdadeiro;
    } return base.TryGetMember(fichário, resultado final);
}
```

**Caso contrário, use a implementação base**

**Se encontrado, construa um novo elemento dinâmico C**

A implementação na listagem 14.31 é simples. O fichário contém o nome da propriedade solicitada, então você procura o subelemento apropriado na árvore B. Se houver um, você cria um novo DynamicXElement com ele, atribui-o ao parâmetro de saída result e retorna true para indicar que a chamada foi vinculada com sucesso C. Se não houver nenhum subelemento com o nome correto, basta chamar a implementação base de TryGetMember() D. A implementação básica de cada um dos métodos TryXXX apenas retorna false e define o parâmetro de saída como nulo, se houver. Você poderia facilmente ter feito isso explicitamente, mas teria duas instruções separadas: uma para definir o parâmetro de saída e outra para retornar falso. Se você preferir um código um pouco mais longo, não há razão para não escrevê-lo — as implementações básicas são apenas um pouco convenientes em termos de fazer tudo o que for necessário para indicar que a ligação falhou.

Evitei um pouco de complexidade: o fichário tem outra propriedade (Ignore-Case) que indica se a propriedade deve ser vinculada sem distinção entre maiúsculas e minúsculas.

Por exemplo, o Visual Basic não faz distinção entre maiúsculas e minúsculas, portanto, sua implementação de fichário retornaria true para essa propriedade, enquanto a do C# retornaria false. Nesta situação, é um pouco estranho. Não só seria mais trabalhoso para o TryGetMember encontrar o elemento sem distinção entre maiúsculas e minúsculas (“mais trabalho” é sempre desagradável, mas não é uma boa razão para não implementá-lo), mas há o problema mais filosófico do que acontece quando você usa o indexador (por número) para selecionar irmãos. O objeto deve lembrar se diferencia maiúsculas de minúsculas e selecionar irmãos da mesma maneira mais tarde? Você pode facilmente entrar em situações em que o comportamento seja difícil de prever e de explicar na documentação. Esse tipo de incompatibilidade de impedância provavelmente ocorrerá também em outras situações semelhantes. Se você almeja a perfeição, é provável que se atrapalhe. Em vez disso, encontre uma solução pragmática que você tenha certeza de poder implementar e manter e, em seguida, documente as restrições.

Com tudo isso implementado, você pode testar DynamicXElement, conforme mostrado na listagem a seguir.

**Listagem 14.32 Testando DynamicXElement**

```
XDocument doc = XDocument.Load("livros.xml"); raiz dinâmica =
Dynamic XElement.CreateInstance(doc.Root); Console.WriteLine(root.book[2]["nome"]);
Console.WriteLine(root.book[1].autor[1]); Console.WriteLine(root.book);
```

Você poderia adicionar mais complexidade à classe, é claro. Você pode adicionar uma propriedade Parent para voltar à árvore ou pode querer alterar o código para usar métodos para acessar subelementos e fazer com que o acesso à propriedade represente atributos. O princípio seria exatamente o mesmo: onde você conhece o nome com antecedência, implemente-o como um membro normal da classe. Se você precisar que seja dinâmico, substitua o método DynamicObject apropriado .

Há mais um polimento a ser aplicado ao DynamicXElement antes de deixá-lo. É hora de anunciar o que você tem a oferecer.

**SUBSTITUINDO GETDYNAMICMEMBERNAMES**

Algumas linguagens, como Python, permitem perguntar a um objeto quais nomes ele conhece. Por exemplo, você pode usar a função dir em Python para gerar uma lista. Essas informações são úteis em um ambiente REPL e também podem ser úteis ao depurar em um IDE. O DLR disponibiliza essas informações por meio do método GetDynamicMemberNames() de DynamicObject e DynamicMetaObject (você conhecerá o último em um minuto). Tudo que você precisa fazer é substituir esse método para fornecer uma sequência de nomes de membros dinâmicos, e as propriedades do seu objeto serão mais detectáveis. A listagem a seguir mostra a implementação de DynamicXElement.

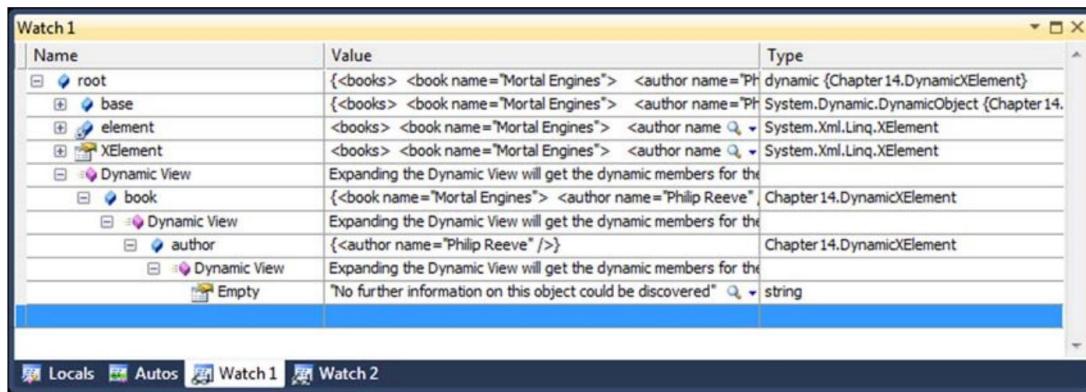
**Listagem 14.33 Implementando GetDynamicMemberNames em DynamicXElement**

```
substituição pública IEnumerable<string> GetDynamicMemberNames() {
    retornar elemento.Elements()
        .Select(x => x.Nome.NomeLocal)
        .Distinto()
        .OrderBy(x => x);
}
```

Como você pode ver, tudo que você precisa é de uma consulta LINQ simples . Nem *sempre* será esse o caso, mas suspeito que muitas implementações dinâmicas poderão usar o LINQ dessa maneira.

Você precisa ter certeza de não retornar o mesmo valor mais de uma vez se houver mais de um elemento com um nome específico, e os resultados serão classificados para consistência. No depurador do Visual Studio 2010, você pode expandir a Visualização Dinâmica de um objeto dinâmico e ver os nomes e valores das propriedades, conforme mostrado na figura 14.6.

Você pode detalhar o objeto dinâmico, mostrando a Visualização Dinâmica em cada nível. Para a figura 14.6, detalhei desde o documento até o primeiro livro e até o autor. A Visualização Dinâmica do autor mostra que não há mais informações na hierarquia.



**Figura 14.6** Visual Studio 2010 exibindo propriedades dinâmicas de um Dynamic XElement

Terminamos agora a classe Dynamic XElement , pelo menos até onde vamos levá-la neste livro. Acredito que o DynamicObject atinge um ponto ideal entre controle e simplicidade: é bastante fácil acertar e tem muito menos restrições do que o Expando-Object. Mas se você realmente precisa de controle total sobre a vinculação, precisará implementar IDynamicMetaObjectProvider diretamente.

### 14.5.3 Implementando IDynamicMetaObjectProvider

Não vou entrar em muitos detalhes aqui, mas quero muito mostrar pelo menos um exemplo de comportamento dinâmico de baixo nível. A parte difícil de implementar IDynamicMetaObjectProvider não é a interface em si - é criar o DynamicMetaObject para retornar do único método da interface. DynamicMetaObject é um pouco como DynamicObject em que ele contém muitos métodos e você substitui os métodos individuais para afetar o comportamento; onde você substituiu DynamicObject.TryGetMember anteriormente, você substituiria DynamicMetaObject.BindGetMember. Mas dentro dos métodos substituídos, em vez disso de tomar a ação necessária directamente, a ideia é construir uma árvore de expressão *descrevendo a ação* necessária e as circunstâncias em que essa ação deve ser tomada.

Esse nível extra de indireção é o motivo pelo qual é um *metaobjeto*.

Irei direto para um exemplo e depois saltarei com apenas uma breve explicação. Eu realmente quero entender a diferença no nível de interação aqui – é um pouco como mexer nas entradas do compilador JIT . A maioria dos desenvolvedores C# não precisará saber o detalhes, e se você precisar fazer isso, provavelmente significa que você está tentando escrever uma biblioteca que responde dinamicamente, mas também precisa ter um bom desempenho. Alternativamente, pode significar que você está tentando construir sua própria linguagem dinâmica. Se for esse o caso, então boa sorte—e encontre um recurso mais abrangente do que este escasso exemplo.

O exemplo não pretende ser inteligente; é um tipo Rumpelstiltskin. Criaremos um instância de Rumpelstiltskin com um determinado nome (armazenado em uma string perfeitamente comum variável) e chamamos métodos no objeto até chamarmos um método com o nome correto.

O objeto escreverá respostas apropriadas com base em nossas suposições.<sup>8</sup> Apenas para tornar isso concreto, a listagem a seguir mostra o código que você eventualmente executará.

#### Listagem 14.34 O objetivo final: chamar métodos dinamicamente até encontrar o nome certo

```
dinâmico x = novo Rumpelstiltskin("Hermione"); x.Harry(); x.Ron();
x.Hermione();
```

O objeto não se chamará Rumpelstiltskin – isso seria óbvio demais. Em vez disso, você usará alguns outros mágicos, embora nenhum deles seja particularmente famoso pela alquimia. O objetivo é que as duas primeiras chamadas de método resultem em negações e a terceira admita a derrota. Você também fará com que suas chamadas de método retornem um valor booleano para indicar se a estimativa foi bem-sucedida, mas, por questões de brevidade, não usaremos o resultado aqui.

Vejamos primeiro o tipo Rumpelstiltskin . Não se esqueça de que este *não* é o meta-objeto - isso virá mais tarde. A próxima listagem mostra o código completo.

#### Listagem 14.35 O tipo Rumpelstiltskin , sem seu código de metaobjeto

```
classe pública selada Rumpelstiltskin: IDynamicMetaObjectProvider
{
    nome da string somente leitura privada; público
    Rumpelstiltskin(nome da string) ← B Constrói uma nova instância
    {
        este.nome = nome;
    }

    public DynamicMetaObject GetMetaObject (expressão de expressão)
    {
        retornar novo MetaRumpelstiltskin(expressão, isto); ← C Expõe
    } ← dinâmica
    Comportamento C

    objeto privado RespondToWrongGuess (string palpite)
    {
        Console.WriteLine("Não, não sou {0}! (Sou {1}).",
            adivinhe, nome);
        retorna falso;
    }

    objeto privado RespondToRightGuess()
    {
        Console.WriteLine("Maldições! Falha de novo!"); retornar verdadeiro;
    }
}
```

← D Responde
 para suposições

Existem três aspectos nesta aula. Existe a construção B, que é perfeitamente comum. Há a implementação do único método C do IDynamicMetaObjectProvider e há dois métodos que você usará para realizar o trabalho real D.

<sup>8</sup> Se você não está familiarizado com o conto de fadas de Rumpelstiltskin, consulte o artigo da Wikipedia (<http://en.wikipedia.org/wiki/Rumpelstiltskin>). O exemplo fará mais sentido depois!

O metaobjeto construído em **C** precisa saber a qual instância está respondendo e qual árvore de expressão se refere à instância no código de chamada. Você é dado a árvore de expressão como um parâmetro, e você conhece sua própria instância por meio da referência `this`, então basta passá-la no construtor.

**POR QUE OS MÉTODOS RETORNAM OBJETO?** Você pode estar se perguntando por que os métodos são declarados para retornar objeto em vez de `bool`. Minha implementação original na verdade tinha métodos vazios, mas infelizmente métodos dinâmicos espera-se que as invocações retornem algo, e o fichário sempre espera objeto, na minha experiência. (Há uma propriedade `ReturnType` que você pode verificar.) Isso faz com que uma chamada para um método `void` lance uma exceção em tempo de execução, e o mesmo se aplica a um método `bool`; você mesmo precisa realizar o boxe para fazer com que os tipos correspondam corretamente. Você poderia construir o boxe na árvore de expressão, mas isso é mais doloroso do que alterar o tipo de retorno do método. Esses são os tipos de sutilezas com as quais você precisará lidar se algum dia implementar `IDynamicMetaObjectProvider` na vida real.

A rigor, você não *precisa* dos dois métodos de resposta. Quando você constrói o comportamento para reagir às chamadas de método recebidas, você *poderia* expressar essa lógica diretamente em uma árvore de expressão. Mas seria relativamente doloroso fazer isso, comparado a apenas retornar uma árvore de expressão que chama o método correto. Mais especificamente, porém, é não seria muito difícil neste caso; em outras situações poderia ser muito pior. Você vai efetivamente criar uma ponte entre os mundos estático e dinâmico, respondendo a chamadas de métodos dinâmicos, redirecionando-os para métodos estáticos com argumentos apropriados. Isso leva a um código mais simples no metaobjeto.

Falando nisso, vamos finalmente dar uma olhada no código do `MetaRumpelstiltskin` - está em a listagem a seguir e é uma classe privada aninhada dentro de `Rumpelstiltskin`.

#### Listagem 14.36 As verdadeiras entradas dinâmicas de Rumpelstiltskin – seu metaobjeto

```
classe privada MetaRumpelstiltskin: DynamicMetaObject
{
    privado estático somente leitura MethodInfo RightGuessMethod =
        typeof(Rumpelstiltskin).GetMethod("RespondToRightGuess",
            BindingFlags.Instance | BindingFlags.NonPublic);

    private static readonly MethodInfo WrongGuessMethod =
        typeof(Rumpelstiltskin).GetMethod("RespondToWrongGuess",
            BindingFlags.Instance | BindingFlags.NonPublic);

    MetaRumpelstiltskin interno
        (Expressão de expressão, criador de Rumpelstiltskin)
        : base (expressão, BindingRestrictions.Empty, criador)
    {}

    substituição pública DynamicMetaObject BindInvokeMember
        (Fichário InvokeMemberBinder, DynamicMetaObject[] args)
    {
        Rumpelstiltskin targetObject = (Rumpelstiltskin)base.Value;
        Expressão self = Expression.Convert(base.Expression,
```

Recorda  
o Real"  
objeto **E**

Obtém  
métodos por  
Reflexão **B**

**C** Delega a  
construção  
à classe base

**D** Responde  
à invocação  
do membro

```

        typeof(Rumpelstiltskin));

Expressão targetBehavior; if (binder.Name
== targetObject.name) {

    targetBehavior = Expression.Call(self, RightGuessMethod);

} outro
{
    targetBehavior = Expression.Call(self, WrongGuessMethod, Expression.Constant(binder.Name));

}

var restrições = BindingRestrictions.GetInstanceRestriction (self, targetObject); retornar novo
    DynamicMetaObject(targetBehavior,
restrições);
}
}

```

**Responde com**  
Comportamento  
**G e restrições**



**F** Determina  
comportamento apropriado

Enquanto digito isso, quase posso ver seus olhos vidrados. A Listagem 14.36 é um código denso e parece muito trabalhoso realizar uma tarefa simples. Apenas lembre-se: é improvável que você precise fazer isso, então relaxe e deixe o sabor geral do código penetrar enquanto os detalhes tomam conta de você.

A primeira metade do código é genuinamente fácil. Você armazena o MethodInfo para os dois métodos de resposta nas variáveis estáticas **B** (eles não mudam para instâncias diferentes) e declara um construtor que não faz nada além de passar seus parâmetros para a classe base **C**. Todo o trabalho real é feito em BindInvokeMember **D**, que precisa resolver duas coisas: como o objeto deve reagir à chamada do método e as circunstâncias nas quais essa decisão é válida.

Você deseja reagir chamando RespondToRightGuess ou RespondToWrongGuess com base no fato de o nome da chamada do método ser igual ao nome do objeto.

O metaobjeto sabe qual é a instância real, porque você a passou para o construtor. Você o acessa novamente usando a propriedade Value e lembra-se dele usando a variável targetObject **E**. Você também precisa da árvore de expressão que foi originalmente usada para criar o metaobjeto, para que possa vincular a chamada de método apropriada inteiramente dentro das árvores de expressão. O método Expression.Convert é o equivalente em árvore de expressão da conversão na linha anterior.

Depois de conhecer o objeto real, você pode verificar seu nome em relação à chamada de método que está vinculando, que está disponível por meio da propriedade InvokeMemberBinder.Name . Você cria uma chamada para o método apropriado usando Expression.Call, passando o nome do método como um argumento se a estimativa estiver errada **F**. Novamente, gostaria de enfatizar que neste ponto você não está realmente chamando o método — você está descrevendo a chamada do método.

As restrições neste caso são simples: esta chamada será sempre vinculada da mesma maneira se estiver chamando o mesmo argumento, mas será vinculada de forma diferente se for chamada em um objeto diferente, porque poderia ter um nome diferente. GetInstanceRestriction retorna uma restrição apropriada; se você quisesse sempre se comportar da mesma maneira

independentemente de qual instância o método foi chamado, você pode usar GetType-Restriction para indicar que a chamada seria tratada da mesma maneira para qualquer instância de Rumpelstiltskin. O código-fonte completo inclui uma implementação alternativa que faz exatamente isso, sempre passando o nome real do método, colocando o teste de condição dentro do método normal.

Finalmente, você cria um novo DynamicMetaObject representando os resultados da ligação G. É bastante confuso que o resultado seja do mesmo tipo que o objeto que está realizando a ligação, mas é assim que o DLR funciona.

Neste ponto, você terminou: cruze os dedos, execute o código e veja se funciona...

Em seguida, depure-o algumas vezes para descobrir exatamente o que está errado, se você for como eu. Como eu disse, isso não é algo que a maioria dos desenvolvedores precisará assumir — é um pouco como o LINQ, em que muito mais pessoas usarão o LINQ do que implementarão seu próprio provedor LINQ baseado em IQueryable . É útil dar uma olhada em como tudo funciona em vez de tratar isso como mágica, mas na maioria das vezes você pode simplesmente sentar e aproveitar o trabalho duro da equipe DLR .

## 14.6 Resumo

Parece que percorremos um longo caminho desde o C# convencional e de tipo estaticamente. Vimos algumas situações em que a digitação dinâmica pode ser útil, como o C# 4 torna isso possível (tanto em termos do código que você escreve quanto de como ele funciona sob a superfície) e como responder dinamicamente às chamadas. Ao longo do caminho, você viu um pouco de COM, um pouco de Python, alguma reflexão e aprendeu um pouco sobre o Dynamic Language Runtime.

Este *não* foi um guia completo sobre como o DLR funciona, ou mesmo como o C# opera com ele. A verdade é que este é um tema profundo com muitos cantos obscuros. Muitos dos problemas são obscuros o suficiente para que você não os encontre — e a maioria dos desenvolvedores nem mesmo usa os cenários simples com frequência. Tenho certeza de que livros inteiros serão escritos sobre o DLR, mas espero ter fornecido detalhes suficientes aqui para permitir que 99% dos desenvolvedores C# continuem com seus trabalhos sem precisar de mais informações. Se você quiser saber mais, a documentação no site do DLR é um bom ponto de partida (veja <http://mng.bz/OM6A>).

Se você nunca usa o tipo dinâmico , pode ignorar completamente a digitação dinâmica. Recomendo que você faça exatamente isso para a maior parte do seu código — em particular, eu não usaria isso como uma muleta para evitar a criação de interfaces, classes base apropriadas e assim por diante. Onde você *precisa* de digitação dinâmica, eu a usaria com a maior moderação possível. Não tome a atitude: “Estou usando dinâmico neste método, então é melhor tornar *tudo* dinâmico”.

Não quero parecer muito negativo. Se você se encontrar em uma situação em que a digitação dinâmica é útil, tenho certeza de que ficará grato por ela estar presente no C# 4. Mesmo que você nunca precise dela para código de produção, recomendo que você experimente. a diversão disso - achei fascinante aprofundar. Você também pode achar o DLR útil sem realmente usar digitação dinâmica; a maior parte do exemplo Python deste capítulo não usou nenhum

recursos de digitação dinâmica, mas usou o DLR para executar o script Python contendo os dados de configuração.

Entre este capítulo e o anterior, ele cobre todos os novos recursos do C# 4.

Em seguida, vem o C# 5, que tem um foco ainda mais restrito do que o C# 4 com tipagem dinâmica.

É realmente *tudo* uma questão de assincronia...

## Parte 5

# C# 5: Assincronia simplificada

Isso é simples de descrever o C# 5: ele possui exatamente um grande recurso (funções assíncronas) e dois pequenos.

O Capítulo 15 é sobre assincronia. O objetivo do recurso de funções assíncronas (geralmente chamado apenas de `async/await`, para abreviar) é tornar a programação assíncrona mais fácil...ou pelo menos mais fácil do que era antes. Não tenta remover a complexidade *inerente* à assincronia; você ainda precisa considerar as consequências da conclusão das operações em uma ordem inesperada ou do usuário pressionar outro botão antes da conclusão da primeira operação, mas isso elimina grande parte da complexidade incidental. Isso permite que você veja a madeira das árvores e construa soluções robustas e legíveis para essas complexidades inerentes.

No passado, o código assíncrono muitas vezes se transformava em espaguete, com o caminho de execução lógica saltando de método para método à medida que uma chamada assíncrona era concluída e iniciava outra. Com funções assíncronas, você pode escrever código que parece síncrono, com estruturas de controle familiares, como loops e blocos `try/catch/finally`, mas com um fluxo de execução assíncrono acionado por uma nova palavra-chave contextual (`await`). A diferença na legibilidade é simplesmente impressionante, na minha experiência. Iremos nos aprofundar neste tópico, não apenas em termos de como a linguagem se comporta, mas como ela é implementada pelo compilador C# da Microsoft.

Isso deixa apenas os dois recursos abordados no capítulo 16: uma ligeira mudança no comportamento irritante do `foreach` que você viu no capítulo 5 e alguns novos atributos que funcionam com o recurso de parâmetros opcionais do C# 4 para permitir o número da linha,

nome do membro e arquivo fonte de um trecho de código a ser fornecido automaticamente pelo compilador. Em seguida, encerrarei esta edição do livro da maneira habitual, com algumas reflexões finais.

Você pode ser perdoado por pensar que isso não parece muito, principalmente porque desconsiderarei deliberadamente os recursos abordados no capítulo 16. Não se deixe enganar; funções assíncronas são realmente importantes, principalmente se você estiver escrevendo aplicativos da Windows Store usando WinRT. A API exposta pelo WinRT é construída em torno da assincronia, para combater interfaces de usuário que não respondem. Sem funções assíncronas, seria muito difícil usá-las. Com os recursos do C# 5, você ainda precisa pensar, mas o código pode ser tão claro quanto posso imaginar que o código assíncrono possa se tornar. Então, em vez de mais descrições de como isso é maravilhoso, vamos conhecer o recurso...

# 15

## Assincronia com assíncrono/aguardar

### Este capítulo cobre

- ÿ Os objetivos fundamentais da assincronia
- ÿ Escrevendo métodos assíncronos e delegados
- ÿ Transformações do compilador para assíncrono
- ÿ O padrão assíncrono baseado em tarefas
- ÿ Assincronia no WinRT

A assincronia tem sido uma pedra no sapato dos desenvolvedores há anos. É conhecido por ser *útil* como uma forma de evitar amarrar um tópico enquanto espera por alguma tarefa arbitrária para ser concluído, mas também tem sido uma dor de cabeça implementá-lo corretamente.

Mesmo dentro do .NET Framework (que ainda é relativamente jovem no grande esquema das coisas), tivemos três modelos diferentes para tentar tornar as coisas mais simples:

- ÿ A abordagem BeginFoo / EndFoo do .NET 1.x, usando IAsyncResult e AsyncCallback para propagar resultados ÿ
- O padrão assíncrono baseado em eventos do .NET 2.0, conforme implementado por BackgroundWorker e WebClient
- ÿ A Task Parallel Library (TPL) introduzida no .NET 4 e expandida no .NET 4.5

Apesar de seu design geralmente excelente, escrever código assíncrono robusto e legível com o TPL era difícil. Embora o suporte ao paralelismo tenha sido grande, existem alguns aspectos da assincronia geral que são muito melhor fixados em uma linguagem do que puramente em bibliotecas.

**ASYNC/AWAIT VAI BALANÇAR SEU MUNDO** A lista introdutória de tópicos pode fazer este capítulo parecer um tanto enfadonho. É uma lista precisa, mas não consegue transmitir o entusiasmo que sinto por esse recurso. Tenho jogado com `async/await` há cerca de dois anos e ainda me faz sentir como um estudante tonto. Acredito firmemente que isso fará com a assincronia o que o LINQ fez com o tratamento de dados quando o C# 3 foi lançado — exceto que lidar com a assincronia era um problema muito mais difícil. Para obter o efeito adequado, leia este capítulo com uma voz mental superexcitada. Espero contagiar você com meu entusiasmo pelo recurso ao longo do caminho.

O principal recurso do C# 5 baseia-se no TPL para que você possa escrever código de aparência síncrona que usa assincronia quando apropriado. Já se foi o espaguete de retornos de chamada, assinaturas de eventos e tratamento fragmentado de erros; em vez disso, o código assíncrono expressa suas intenções de forma clara e de uma forma que se baseia nas estruturas com as quais os desenvolvedores já estão familiarizados. Uma nova construção de linguagem permite “aguardar” uma operação assíncrona. Essa “espera” se parece muito com uma chamada de bloqueio normal, pois o resto do seu código não continuará até que a operação seja concluída, mas consegue fazer isso sem bloquear o thread em execução no momento. Não se preocupe se essa afirmação parecer completamente contraditória — tudo ficará claro no decorrer do capítulo.

O .NET Framework adotou a assincronia de todo o coração na versão 4.5, expondo versões assíncronas de muitas operações, seguindo um *padrão assíncrono baseado em tarefas* recentemente documentado para fornecer uma experiência consistente em diversas APIs. Além disso, a nova plataforma Windows Runtime<sup>1</sup> usada para criar aplicativos da Windows Store no Windows 8 impõe assincronia para todas as operações de longa execução (ou potencialmente de longa execução). Resumindo, o futuro é assíncrono e seria tolo se não aproveitasse as vantagens dos novos recursos da linguagem ao tentar gerenciar a complexidade adicional. Mesmo se você não estiver usando o .NET 4.5, a Microsoft criou um pacote NuGet (`Microsoft.Bcl.Async`) que permite usar os novos recursos ao direcionar o .NET 4, Silverlight 4 ou 5 ou Windows Phone 7.5 ou 8.

Só para deixar claro, o C# não se tornou onisciente, adivinhando onde você pode querer executar operações de forma simultânea ou assíncrona. O compilador é inteligente, mas não tenta remover a complexidade *inerente* à execução assíncrona. Você ainda precisa pensar com cuidado, mas a beleza do C# 5 é que todo o código padrão tedioso e confuso que costumava ser necessário desapareceu. Para começar, sem a distração de todas as complicações necessárias para tornar seu código assíncrono, você pode se concentrar nas partes difíceis.

---

<sup>1</sup> Isso é comumente conhecido como WinRT; não deve ser confundido com o Windows RT, que é a versão do Windows 8 executada em processadores ARM.

Uma palavra de advertência: este tópico está razoavelmente avançado. Ele tem a infeliz propriedade de ser incrivelmente importante (realisticamente, mesmo os desenvolvedores iniciantes precisarão ter uma compreensão superficial dele em alguns anos), mas também é bastante complicado de entender, para começar. Assim como no restante deste livro, não fugirei da complexidade — veremos o que está acontecendo com bastante detalhe.

É possível que eu quebre um pouco o seu cérebro, e espero que ele volte a ser montado mais tarde. Se tudo começar a parecer um pouco maluco, não se preocupe — não é só você; a perplexidade é uma reação inteiramente natural. A boa notícia é que quando você *usa* C# 5, tudo faz sentido superficialmente. Somente quando você tenta pensar exatamente no que está acontecendo nos bastidores é que as coisas ficam difíceis. É claro que faremos exatamente isso mais tarde — e também veremos como usar o recurso de maneira eficaz.

Vamos começar.

## 15.1 Apresentando funções assíncronas

Até agora afirmei que o C# 5 torna o assíncrono mais fácil, mas forneci apenas uma pequena descrição dos recursos envolvidos. Vamos consertar isso e depois ver um exemplo.

C# 5 introduz o conceito de *função assíncrona*. Este é sempre um método ou uma função anônima<sup>2</sup> declarada com o modificador `async` e pode incluir expressões de espera. Essas expressões de espera são os pontos onde as coisas ficam interessantes do ponto de vista da linguagem: se o valor que a expressão está aguardando ainda não estiver disponível, a função assíncrona retornará imediatamente e continuará de onde parou (em um thread apropriado) quando o valor estiver disponível. O fluxo natural de “não execute a próxima instrução até que esta esteja concluída”.

pleted” ainda é mantido, mas sem bloqueio.

Vou dividir essa descrição confusa em termos e comportamentos mais concretos mais tarde em diante, mas você realmente precisa ver um exemplo disso antes que faça algum sentido.

### 15.1.1 Primeiros encontros do tipo assíncrono

Vamos começar com algo muito simples, mas que demonstra a assincronia de forma prática. Muitas vezes amaldiçoamos a latência da rede por causar atrasos em nossas aplicações reais, mas a latência torna mais fácil mostrar por que a assincronia é tão importante. Dê uma olhada na listagem a seguir.

#### Listagem 15.1 Exibindo um comprimento de página de forma assíncrona

```
classe AsyncForm: Formulário {
    Etiqueta de etiqueta;
    Botão botão;
    public AsyncForm() {
        rótulo = novo rótulo { Localização = novo ponto (10, 20),
            Texto = "Comprimento" };
    }
}
```

<sup>2</sup> Apenas como lembrete, uma função anônima é uma expressão lambda ou um método anônimo.

```

botão = novo botão { Localização = novo ponto (10, 50),
    Texto = "Clique" };
botão.Click += DisplayWebSiteLength; AutoSize = verdadeiro;
Controls.Add(rótulo);
Controls.Adicionar(botão);

}

async void DisplayWebSiteLength(objeto remetente, EventArgs e) {

    label.Text = "Buscando..."; usando (cliente
    HttpClient = new HttpClient() {

        Começa C
        buscando
        a página
    }
}
...
Application.Run(novo AsyncForm());

```

**C**omeça buscando a página

**D** atualiza a IU

**Fios**  
Manipulador de eventos [B](#)

A primeira parte da listagem 15.1 simplesmente cria a UI e conecta um manipulador de eventos para o botão de maneira direta B. É o método `DisplayWebSiteLength` que interessa aqui. Quando você clica no botão, o texto da página inicial do livro é buscado C, e o rótulo é atualizado para exibir o comprimento do HTML em caracteres D. O `HttpClient` também é disposto adequadamente, independentemente de a operação ser bem-sucedida ou falhar - algo que seria muito fácil de esquecer se você estivesse escrevendo código assíncrono semelhante em C# 4.

**DESCARTE DE TAREFAS** Tenho o cuidado de descartar o `HttpClient` quando termino de usá-lo, mas não estou descartando a tarefa retornada por `GetStringAsync`, mesmo que `Task` implemente `IDisposable`. Felizmente, você realmente não precisa se desfazer de tarefas em geral. O pano de fundo disso é um tanto complicado, mas Stephen Toub explica isso em uma postagem de blog dedicada ao tópico: <http://mng.bz/E6L3>.

Eu poderia ter escrito um programa de exemplo menor como um aplicativo de console, mas espero que a listagem 15.1 seja uma demonstração mais convincente. Em particular, se você remover as palavras-chave `async` e `await` contextuais, alterar `HttpClient` para  `WebClient` e alterar `GetStringAsync` para `DownloadString`, o código ainda será compilado e funcionará... mas a UI irá congelar enquanto busca o conteúdo da página.<sup>3</sup> Se você executar a versão assíncrona (de preferência em uma conexão de rede lenta), verá que a interface do usuário responde – você ainda pode mover a janela enquanto a página da web está sendo buscada.

<sup>3</sup> `HttpClient` é, em certo sentido, o  `WebClient` “novo e aprimorado” – é a API HTTP preferida para .NET 4.5 em diante e contém apenas operações assíncronas. Se você estiver escrevendo um aplicativo da Windows Store, não terá nem a opção de usar o  `WebClient`.

A maioria dos desenvolvedores está familiarizada com as duas regras de ouro do threading no Windows Desenvolvimento de formulários:

- ÿ Não execute nenhuma ação demorada no thread de UI .
- ÿ Não acesse nenhum controle de UI *além* do thread de UI .

Estas são mais fáceis de declarar do que obedecer. Como exercício, você pode tentar algumas maneiras diferentes de criar código semelhante à listagem 15.1 sem usar os novos recursos do C# 5. Para este exemplo extremamente simples, na verdade não é tão ruim usar o WebClient baseado em eventos.

`DownloadStringAsync`, mas assim que um controle de fluxo mais complexo (tratamento de erros, espera pela conclusão de várias páginas e assim por diante) entra na equação, o código “legado” rapidamente se torna difícil de manter, enquanto o código C# 5 pode ser modificado de uma forma natural.

Neste ponto, o método `DisplayWebSiteLength` parece um tanto mágico: você sabe que ele faz o que você precisa, mas não tem ideia de como. Vamos desmontá-lo um pouquinho, guardando os detalhes realmente sangrentos para mais tarde.

### 15.1.2 Detalhando o primeiro exemplo

Começaremos expandindo um pouco o método - dividindo a chamada para `HttpClient.GetStringAsync` da expressão `await` para destacar os tipos envolvidos:

```
async void DisplayWebSiteLength(object remetente, EventArgs e) {
    label.Text = "Buscando..."; usando (cliente
        HttpClient = new HttpClient()) {

        Tarefa<string> tarefa =
            cliente.GetStringAsync("http://csharpinprofundidade.com");
        string text = aguarda tarefa; rótulo.Text =
            text.Length.ToString();
    }
}
```

Observe como o tipo de tarefa é `Task<string>`, mas o tipo da expressão da tarefa `await` é apenas `string`. Nesse sentido, uma expressão `await` executa uma operação de “desembrulhamento” – pelo menos quando o valor que está sendo esperado é `Task<TResult>`. (Você também pode esperar outros tipos, como verá, mas `Task<TResult>` é um bom ponto de partida.) Esse é um aspecto do `await` que não parece diretamente relacionado à assincronia, mas facilita a vida.

O principal objetivo do `await` é evitar o bloqueio enquanto você espera a conclusão de operações demoradas. Você pode estar se perguntando como tudo isso funciona em termos concretos de threading. Você está definindo `label.Text` no início e no final do método, então é razoável assumir que ambas as instruções são executadas no thread da UI ... e ainda assim você claramente *não* está bloqueando o thread da UI enquanto espera pelo página da web para download.

O truque é que o método realmente retorna assim que você pressiona a expressão `await`. Até esse ponto, ele é executado de forma síncrona no thread da UI , assim como qualquer outro manipulador de eventos faria. Se você colocar um ponto de interrupção na primeira linha e acertá-lo no

depurador, você verá que o rastreamento de pilha mostra que o botão está ocupado gerando seu evento `Click`, incluindo o método `Button.OnClick`. Quando você atinge o `await`, o código verifica se o resultado já está disponível e, se não estiver (o que quase certamente será o caso), ele agenda uma *continuação* a ser executada quando a operação web for concluída. Neste exemplo, a continuação executa o resto do método, efetivamente saltando para o final da expressão `await`, de volta ao thread da UI, exatamente como você deseja para manipular a UI.

**CONTINUAÇÕES** Uma continuação é efetivamente um retorno de chamada a ser executado quando uma operação assíncrona (ou mesmo qualquer tarefa) for concluída. Num método assíncrono, a continuação mantém o estado de controle do método; assim como um encerramento mantém seu ambiente em termos de variáveis, uma continuação lembra onde chegou, para que possa continuar a partir daí quando for executada. A classe `Task` possui um método específico para anexar continuações: `Task.ContinueWith`.

Se você colocar um ponto de interrupção no código *após* a expressão `await`, verá que o rastreamento de pilha não contém mais o método `Button.OnClick` (assumindo que a expressão `await` é necessária para agendar a continuação). Esse método terminou de ser executado há muito tempo. A pilha de chamadas agora será efetivamente o loop de eventos do Windows Forms, com algumas camadas de infraestrutura assíncrona no topo. A pilha de chamadas será muito semelhante ao que você veria se chamassem `Control.Invoke` de um thread em segundo plano para atualizar a UI adequadamente, mas tudo foi feito para você. A princípio pode ser enervante notar que a pilha de chamadas muda drasticamente sob seus pés, mas é absolutamente necessário que a assincronia seja eficaz.

Caso você esteja se perguntando, tudo isso é tratado pelo compilador criando uma máquina de estado complicada. Esse é um detalhe de implementação, e é instrutivo examiná-lo para entender melhor o que está acontecendo, mas primeiro precisamos de uma descrição mais concreta do que estamos tentando alcançar e do que a linguagem realmente específica.

## 15.2 Pensando em assincronia

Se você pedir a um desenvolvedor para descrever a execução assíncrona, é provável que ele comece a falar sobre multithreading. Embora essa seja uma parte importante dos usos *típicos* da assincronia, ela não é realmente necessária para a execução assíncrona. Para apreciar totalmente como funciona o recurso assíncrono do C# 5, é melhor abandonar qualquer pensamento sobre threading e voltar ao básico.

### 15.2.1 Fundamentos da execução assíncrona

A assincronia atinge o cerne do modelo de execução com o qual os desenvolvedores de C# estão familiarizados. Considere um código simples como este:

```
Console.WriteLine("Primeiro");
Console.WriteLine("Segundo");
```

Você espera que a primeira chamada seja concluída e, em seguida, a segunda chamada seja iniciada. Fluxos de execução de uma declaração para a próxima, em ordem. Mas um modelo de execução assíncrona não funciona assim. Em vez disso, é tudo uma questão de *continuações*. Quando você começa a fazer alguma coisa, você diz a essa operação o que deseja que aconteça quando essa operação for concluída. Você pode ter ouvido (ou usado) o termo *retorno de chamada* para a mesma ideia, mas isso um significado mais amplo do que aquele que buscamos aqui. No contexto da assíncronia, estou usando o termo para se referir a retornos de chamada que preservam o estado de controle do programa—não retornos de chamada arbitrários para outros fins, como manipuladores de eventos GUI .

As continuações são naturalmente representadas como delegados no .NET e normalmente são ações que recebem os resultados da operação assíncrona. É por isso que, para usar o métodos assíncronos no WebClient anteriores ao C# 5, você conectaria vários eventos para dizer qual código deve ser executado em caso de sucesso, falha e assim por diante. O problema é que criar todos esses delegados para uma sequência complicada de etapas acaba sendo muito complicado, mesmo com o benefício das expressões lambda. É ainda pior quando você tenta ter certeza de que o tratamento de erros está correto. (Em um dia bom, eu posso estar razoavelmente confiante de que os caminhos de sucesso do código assíncrono manuscrito são correto. Normalmente tenho menos certeza de que ele reage da maneira certa em caso de falha.)

Essencialmente, tudo o que await em C# faz é pedir ao compilador para construir uma continuação para você. Para uma ideia que pode ser expressa de forma tão simples, entretanto, as consequências para a legibilidade e a sanidade do desenvolvedor são notáveis.

Minha descrição anterior de assíncronia foi idealizada. A realidade no padrão assíncrono baseado em tarefas é um pouco diferente. Em vez de a continuação ser passado para a operação assíncrona, a operação assíncrona inicia e retorna um token que você pode usar para fornecer a continuação posteriormente. Representa a operação em andamento, que pode ter sido concluída antes de retornar ao código de chamada, ou ainda pode estar em andamento. Esse token é então usado sempre que você quiser expressar esta ideia: "Eu não pode prosseguir até que esta operação seja concluída." Normalmente o token é na forma de Task ou Task<TResult>, mas não precisa ser assim.

O fluxo de execução em um método assíncrono em C# 5 normalmente segue estes linhas:

- 1** Faça algum trabalho.
- 2** Inicie uma operação assíncrona e lembre-se do token que ela retorna.
- 3** Possivelmente faça mais algum trabalho. (Muitas vezes você não pode fazer nenhum progresso adicional até a operação assíncrona foi concluída e, nesse caso, esta etapa está vazia.)
- 4** Aguarde a conclusão da operação assíncrona (por meio do token).
- 5** Faça mais algum trabalho.
- 6** Concluir.

Se você não se importasse exatamente com o significado da parte "esperar", você poderia fazer tudo isso em C# 4. Se você quiser  *bloquear* até que a operação assíncrona seja concluída, o token normalmente fornecerá uma maneira de fazer isso. Para uma tarefa, você poderia simplesmente chamar Wait(). Nesse ponto, porém, você está utilizando um recurso valioso (um thread) e não fazendo

qualquer trabalho útil. É um pouco como telefonar pedindo uma entrega de pizza e depois ficar parado na porta da frente até que ela chegue. O que você realmente quer é fazer outra coisa, ignorando a pizza até que ela chegue. É aí que entra a espera.

Quando você “espera” por uma operação assíncrona, você está realmente dizendo: “Já fui o mais longe que pude. Continue quando a operação for concluída.” Mas se você não vai bloquear o tópico, o que pode fazer? Muito simplesmente, você pode retornar ali mesmo. Você mesmo continuará de forma assíncrona. E se você quiser que seu chamador saiba quando seu método assíncrono for concluído, você passará um token de volta para ele, que ele poderá bloquear se quiser ou (mais provavelmente) usar com outra continuação. Muitas vezes você acabará com uma pilha inteira de métodos assíncronos chamando uns aos outros — é quase como se você entrasse em um “modo assíncrono” para uma seção de código. Não há nada na linguagem que afirme que isso deva ser feito dessa maneira, mas o fato de que o mesmo código que consome operações assíncronas também se comporta como uma operação assíncrona certamente incentiva isso.

#### Contextos de sincronização

Mencionei anteriormente que uma das regras de ouro do código da UI é que você não deve atualizar a interface do usuário a menos que esteja no thread certo. No exemplo “verificar o comprimento da página da web” (listagem 15.1), você precisa garantir que o código após a expressão await seja executado no thread da UI. As funções assíncronas retornam ao thread certo usando SynchronizationContext — uma classe que existe desde o .NET 2.0 e é usada por outros componentes, como BackgroundWorker. Um Contexto de Sincronização generaliza a ideia de executar um delegado “em um thread apropriado”; suas mensagens Post (assíncronas) e Send (síncronas) são semelhantes a Control.BeginInvoke e Control.Invoke no Windows Forms.

Diferentes ambientes de execução utilizam contextos diferentes; por exemplo, um contexto pode permitir que qualquer thread do pool de threads execute a ação fornecida. Há mais informações contextuais do que apenas o contexto de sincronização, mas se você começar a se perguntar como os métodos assíncronos conseguem ser executados exatamente onde você deseja, tenha esta barra lateral em mente.

Para obter mais informações sobre SynchronizationContext, leia o artigo de Stephen Cleary na revista MSDN sobre o assunto (<http://mng.bz/5cDw>). Em particular, preste muita atenção se você for um desenvolvedor ASP.NET: o contexto do ASP.NET pode facilmente fazer com que desenvolvedores incautos criem impasses dentro de um código que pareça bom.

Com a teoria fora do caminho, vamos examinar mais de perto os detalhes concretos dos métodos assíncronos. Funções anônimas assíncronas se enquadram no mesmo modelo mental, mas é muito mais fácil falar sobre métodos assíncronos.

### 15.2.2 Modelagem de métodos assíncronos

Acho muito útil pensar em métodos assíncronos como mostra a figura 15.1.

Aqui você tem três blocos de código (os métodos) e dois limites (os tipos de retorno de método). Como um exemplo muito simples, você pode ter um código como este:

```
Tarefa assíncrona estática<int> GetPageLengthAsync(string url)
{
    usando (cliente HttpClient = novo HttpClient())
    {
        Tarefa<string> fetchTextTask = client.GetStringAsync(url);
        comprimento interno = (aguarde fetchTextTask).Length;
        comprimento de retorno;
    }
}

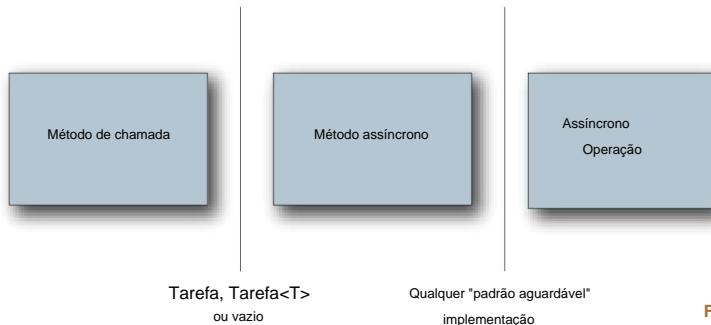
vazio estático PrintPageLength()
{
    Tarefa<int> comprimentoTask =
        GetPageLengthAsync("http://csharpinprofundidade.com");
    Console.WriteLine(comprimentoTask.Result);
}
```

As cinco partes da figura 15.1 correspondem ao código anterior assim:

- ÿ O método de chamada é PrintPageLength. ÿ O
- método assíncrono é GetPageLengthAsync. ÿ A
- operação assíncrona é HttpClient.GetStringAsync. ÿ O limite entre o método de chamada e o método assíncrono é Task<int>. ÿ A fronteira entre o método assíncrono e a operação assíncrona é
- Tarefa<string>.

Estamos interessados principalmente no método assíncrono em si, mas incluí os outros métodos para que você possa ver como todos eles interagem. Em particular, você definitivamente precisa saber sobre os tipos válidos nos limites do método.

Vou me referir a esses bloqueios e limites repetidamente no restante deste capítulo, então tenha em mente a Figura 15.1 enquanto continua lendo.



**Figura 15.1** Modelo assíncrono

### 15.3 Sintaxe e semântica

Finalmente estamos prontos para ver como escrever métodos assíncronos e como eles se comportarão.

Há muito o que abordar aqui, já que “o que você pode fazer” e “o que acontece quando você faz” se misturam em grande parte.

Existem apenas duas novas partes de sintaxe: `async` é um modificador usado ao declarar um método assíncrono e expressões `await` consomem operações assíncronas. Mas acompanhar como as informações são transferidas entre as diferentes partes do seu programa fica complicado muito rapidamente, especialmente quando você precisa considerar o que acontece quando as coisas dão errado. Tentei separar os diferentes aspectos, mas seu código lidará com tudo de uma vez. Se você se perguntar: “Mas e...?” enquanto lê esta seção, continue lendo – é provável que sua dúvida seja respondida em breve.

Vamos começar com a declaração do método em si – essa é a parte mais fácil...

#### 15.3.1 Declarando um método assíncrono

A sintaxe para uma declaração de método assíncrono é exatamente a mesma de qualquer outro método, exceto que deve incluir a palavra-chave contextual `async`. Isso pode aparecer em qualquer lugar antes do tipo de retorno. Tudo isso é válido:

```
public static async Task<int> FooAsync() { ... } public async static Task<int>
FooAsync() { ... } async public Task<int> FooAsync() { ... } public async
virtual Task<int > FooAsync() {...}
```

Minha preferência pessoal é manter o modificador `async` logo antes do tipo de retorno, mas não há razão para você não criar sua própria convenção. Como sempre, discuta isso com sua equipe e tente ser consistente em uma base de código.

Agora, a palavra-chave contextual `async` tem um segredo sujo: os designers da linguagem realmente não precisaram incluí-la. Assim como o compilador entra em uma espécie de “modo de bloco iterador” quando você tenta usar retorno de rendimento ou quebra de rendimento em um método com um tipo de retorno adequado, o compilador poderia apenas ter detectado o uso de `await` dentro de um método e usado isso para entrar no “modo assíncrono”. Mas estou pessoalmente satisfeito com o fato de o `async` ser necessário, pois facilita muito a leitura do código escrito usando métodos assíncronos. Ele define suas expectativas imediatamente, então você está procurando ativamente por expressões de espera - e pode procurar ativamente por quaisquer chamadas de bloqueio que devam ser transformadas em uma chamada assíncrona e em uma expressão de espera.

O fato de o modificador assíncrono não ter representação<sup>4</sup> no código gerado é importante. No que diz respeito ao método de chamada, é apenas um método normal, possivelmente retornando uma tarefa. Você pode alterar um método existente (com uma assinatura apropriada) para usar `async`, ou você pode ir na outra direção – é uma mudança compatível em termos de origem e binário.

---

<sup>4</sup> Bem, mais ou menos. Na prática existe um atributo aplicado, como você verá mais adiante, mas não faz parte da assinatura do método e pode ser ignorado no que diz respeito aos humanos. É realmente usado para ajudar as ferramentas a identificar para onde foi o código “real”.

### 15.3.2 Tipos de retorno de métodos assíncronos

A comunicação entre o chamador e o método assíncrono ocorre efetivamente em termos do valor retornado. As funções assíncronas estão limitadas aos seguintes tipos de retorno:

- ÿ nulo
- ÿ Tarefa
- ÿ Task<TResult> (para algum tipo TResult, que pode ser um parâmetro de tipo)

Os tipos .NET 4 Task e Task<TResult> representam uma operação que pode ainda não ter sido concluída; Task<TResult> deriva de Task. A diferença entre os dois é essencialmente que Task<TResult> representa uma operação que retorna um valor do tipo T, enquanto Task não precisa produzir nenhum resultado. Porém, ainda é útil retornar uma Task, pois permite que o chamador anexe suas próprias continuações à tarefa retornada, detecte quando a tarefa falhou ou foi concluída e assim por diante. Em certo sentido, você pode pensar em Task como sendo um tipo Task<void>, se tal coisa fosse válida.

A capacidade de retornar void de um método assíncrono foi projetada para compatibilidade com manipuladores de eventos. Por exemplo, você pode ter um manipulador de cliques no botão da UI como este:

```
private async void LoadStockPrice(object remetente, EventArgs e) {  
  
    string ticker = tickerInput.Text; preço decimal =  
    aguarda stockPriceService.FetchPriceAsync(ticker); preçoDisplay.Text = preço.ToString("c");  
  
}
```

Este é um método assíncrono, mas o código de chamada (o método OnClick do botão ou qualquer parte do código da estrutura que esteja gerando o evento) realmente não se importa. Ele não precisa saber quando você realmente terminou de lidar com o evento – quando você carregou o preço das ações e atualizou a interface do usuário. Ele apenas chama o manipulador de eventos que lhe foi fornecido. O fato de que o código gerado pelo compilador terminará com uma máquina de estado anexando uma continuação ao que quer que seja retornado por FetchPriceAsync é efetivamente um detalhe de implementação.

Você pode assinar um evento com o método anterior como se fosse qualquer outro manipulador de eventos:

```
loadStockPriceButton.Click += LoadStockPrice;
```

Afinal (e sim, estou trabalhando nisso deliberadamente), é apenas um método normal no que diz respeito à chamada de código. Ele possui um tipo de retorno void e parâmetros do tipo object e EventArgs, o que o torna adequado como ação para uma instância delegada do EventHandler.

A assinatura do evento é praticamente a única vez que recomendo retornar void de um método assíncrono. Qualquer outro momento em que você não precise retornar um valor específico, é melhor declarar o método para retornar Task. Dessa forma, o chamador pode aguardar a conclusão da operação, detectar falhas e assim por diante.

Uma restrição adicional em torno da assinatura de um método assíncrono: nenhum dos parâmetros pode usar os modificadores out ou ref. Isso faz sentido, pois esses modificadores são

para comunicar informações de volta ao código de chamada; porque alguns dos assíncronos pode não ter sido executado quando o controle de tempo retorna ao chamador, o valor do parâmetro por referência pode não ter sido definido. Na verdade, pode ficar mais estranho do que isso: imagine passar uma variável local como argumento para um parâmetro ref - o assíncrono método pode acabar tentando definir essa variável depois que o método de chamada já tiver concluído. Não faz muito sentido tentar fazer isso, então o compilador proíbe.

Depois de declarar o método, você pode começar a escrever o corpo e aguardar outras operações assíncronas.

### 15.3.3 O padrão aguardável

Um método assíncrono pode basicamente conter quase tudo que um método C# regular pode conter, além de expressões de espera . Você pode usar todos os tipos de fluxo de controle – loops, exceções, usando declarações, qualquer coisa. O código se comportará normalmente. O único interessante bits são o que as expressões de espera fazem e como os valores de retorno são propagados.

#### Restrições de espera

Assim como o retorno de rendimento, existem restrições sobre onde você pode usar o wait expressões. Você não pode usá-los em blocos catch ou finalmente , anônimos não assíncronos funções,<sup>5</sup> o corpo de uma instrução de bloqueio ou código inseguro.

Essas restrições são para sua segurança – principalmente as restrições em torno de fechaduras. Se você já quis manter um bloqueio enquanto uma operação assíncrona é concluída, você deve redesenhar seu código. Não contorne a restrição do compilador chamando Monitor.TryEnter e Monitor.Exit manualmente com um bloco try/finally— altere seu código para não precisar do bloqueio durante a operação. Se isso for realmente, realmente estranho na sua situação, considere usar o SemaphoreSlim , com seu Método WaitAsync .

Uma expressão de espera é muito simples – é apenas esperar antes de outra expressão. Mas há limites para o que você pode esperar, é claro. Só para lembrar, estamos falando sobre o segundo limite da figura 15.1 – como o método assíncrono interage com outra operação assíncrona. Informalmente, você só pode esperar por algo que descreve uma operação assíncrona. Em outras palavras, algo que fornece a você com os meios de

↳ Informar se já está concluído ou não ↳ Anexar uma continuação se ainda não tiver terminado ↳ Obter o resultado, que pode ser um valor de retorno, mas pelo menos é uma indicação de sucesso ou fracasso

---

<sup>5</sup> Expressões lambda e métodos anônimos que não são declarados com `async` — portanto, qualquer função anônima declaração que seria válida em C# 4. Você verá funções anônimas assíncronas na seção 15.4.

Você pode esperar que isso seja expresso por meio de interfaces, mas (na maior parte) não é. Há apenas uma interface envolvida e ela cobre apenas a parte de “anexar uma continuação”. Mesmo isso é muito simples – e você quase nunca precisará lidar com isso diretamente. Está no namespace `System.Runtime.CompilerServices` e tem a seguinte aparência:

```
// Interface real em System.Runtime.CompilerServices public interface INotifyCompletion
{
    void OnCompleted(Continuação da ação);
}
```

A maior parte do trabalho é expressa por meio de padrões, um pouco como consultas `foreach` e LINQ . Para tornar a forma do padrão mais clara, vou apresentá-lo brevemente *como se houvesse interfaces envolvidas*, mas na verdade não há. Vou cobrir a realidade em um momento. Vamos dar uma olhada nas interfaces imaginárias:

```
// Aviso: estes não existem realmente
// Interfaces imaginárias para operações assíncronas retornando valores public interface IAwaitable<T> {
```

```
    IAwaiter<T> GetAwaiter();
}

interface pública IAwaiter<T> : INotifyCompletion {

    bool IsCompleted {obter; }
    T GetResult();

    // Herdado de INotifyCompletion // void OnCompleted(Action
    continuação);
}

// Interfaces imaginárias para operações assíncronas "void" public interface IAwaitable {
```

```
    IAwaiter GetAwaiter();
}

interface pública IAwaiter: INotifyCompletion {

    bool IsCompleted {obter; } void GetResult();

    // Herdado de INotifyCompletion // void OnCompleted(Action
    continuação);
}
```

Provavelmente eles lembram `IEnumerable<T>` e `IEnumerator<T>`. Para iterar uma coleção em um loop `foreach` , o compilador gera código que chama `Get-Enumerator()` primeiro e depois usa `MoveNext()` e `Current`. Da mesma forma, em métodos assíncronos, sempre que você escreve uma expressão de espera , o compilador irá gerar um código que primeiro chama `GetAwaiter()` e, em seguida, usa os membros do awaiter para aguardar o resultado de forma adequada.

O compilador C# exige que o aguardador implemente `INotifyCompletion`. Isto ocorre principalmente por razões de eficiência; algumas versões de pré-lançamento do compilador não tinham interface alguma.

Todos os outros membros são verificados pelo compilador apenas por assinatura. É importante ressaltar que o método `GetAwaiter()` em si não precisa ser um método de instância normal. Pode ser um método de extensão com o que você deseja usar uma expressão de espera. Os membros `IsCompleted` e `GetResult` precisam ser membros reais de qualquer tipo retornado de `GetAwaiter()`, mas não precisam ser públicos — eles só precisam estar acessíveis ao código que contém a expressão `await`.

O texto anterior descreve o que é necessário para que uma expressão seja usada como alvo da palavra-chave `await`, mas toda a expressão em si também tem um tipo interessante: se `GetResult()` retornar `void`, então o tipo geral da expressão `await` será nada — a expressão `await` deve ser uma instrução independente. Caso contrário, o tipo geral é igual ao tipo de retorno de `GetResult()`.

Por exemplo, `Task<TResult>.GetAwaiter()` retorna um `TaskAwaiter<TResult>`, que possui um método `GetResult()` retornando `TResult`. (Nenhuma surpresa nisso, espero.)

A regra sobre o tipo da expressão de espera é o que nos permite escrever isto:

```
usando (var cliente = new HttpClient()) {
```

```
    Tarefa<string> tarefa = client.GetStringAsync(...); string resultado = aguarda
    tarefa;
}
```

Compare isso com o método estático `Task.Yield()`, que retorna um `YieldAwaitable`.

Isso, por sua vez, possui um método `GetAwaiter()` retornando um `YieldAwaitable.Yield-Awaiter`, que possui um método `GetResult` retornando `void`. Isso significa que você só pode usá-lo assim:

```
aguarde Task.Yield();
```

Ou se você realmente quisesse dividir as coisas - por mais estranho que fosse:

```
YieldAwaitable rendimento = Task.Yield(); aguarde o rendimento;
```

A expressão `await` aqui não retorna nenhum tipo de valor, então você não pode atribuí-la a uma variável, ou passá-la como um argumento de método, ou fazer qualquer outra coisa que você possa fazer com expressões classificadas como valores.

Um ponto importante a ser observado é que, como `Task` e `Task<TResult>` implementam o padrão `awaitable`, você pode chamar um método assíncrono de outro, e assim por diante:

```
tarefa assíncrona pública<int> FooAsync() {

    string bar = aguarda BarAsync(); // Obviamente
    isso normalmente seria mais complicado... return bar.Length;

}
```

```
tarefa assíncrona pública<string> BarAsync() {
    // Algun código assíncrono que poderia chamar mais métodos assíncronos...
}
```

Essa capacidade de compor operações assíncronas é um dos aspectos do recurso assíncrono que realmente o faz brilhar. Quando você estiver no modo assíncrono, é muito fácil permanecer lá, escrevendo código que flui naturalmente.

Mas estou me adiantando. Descrevi o que o compilador precisa para para você esperar por algo, mas não pelo que realmente faz.

#### 15.3.4 O fluxo de expressões de espera

Um dos aspectos mais curiosos do recurso assíncrono no C# 5 é como o await pode ser simultaneamente intuitivo e extremamente confuso. Se você não pensar muito sobre isso, é muito simples. Se você simplesmente aceitar que ele fará o que você quer, sem realmente definir exatamente o que você quer começar, você provavelmente ficará bem... pelo menos até que algo aconteça.  
dá errado.

Depois que você começa a tentar descobrir exatamente o que deve acontecer para alcançar o efeito desejado, as coisas ficam um pouco mais complicadas. Dado que você está lendo um livro com "Em profundidade" no título, presumo que você queira saber sobre esses detalhes. No longo prazo, prometo que isso permitirá que você use o wait com mais confiança e de forma mais eficaz.

Mesmo assim, recomendo que você tente desenvolver a capacidade de ler código assíncrono em dois níveis diferentes, dependendo do seu contexto: quando não precisar pensar nas etapas individuais listadas aqui, deixe-as passar por você. Leia o código quase como se fosse síncrono, apenas anotando onde o código espera de forma assíncrona pela conclusão de uma operação ou outra. Então, quando você ficar preso em algum problema espinhoso em que o código não está se comportando como esperado, você pode mudar para o modo mais forense, descobrindo quais threads estarão envolvidos, onde e em que posição a pilha de chamadas estará. qualquer momento. (Não estou dizendo que isso será simples, mas compreender o maquinário pelo menos tornará isso mais viável.)

#### EXPANDINDO EXPRESSÕES COMPLEXAS

Vamos começar simplificando um pouco as coisas. Às vezes, wait é usado com o resultado de uma chamada de método ou ocasionalmente com uma propriedade, como esta:<sup>6</sup>

```
string pageText = aguarda novo HttpClient().GetStringAsync(url);
```

Isso faz parecer que await pode modificar o significado de toda a expressão. A verdade é que await opera apenas com um único valor. A linha anterior é equivalente a esta:

```
Tarefa<string> tarefa = new HttpClient().GetStringAsync(url); string pageText = aguarda
tarefa;
```

---

<sup>6</sup> Este exemplo é um pouco artificial, já que você normalmente usaria uma instrução using para o HttpClient, mas espero que você me perdoe por não descartar recursos apenas desta vez.

Da mesma forma, o resultado de uma expressão de espera pode ser usado como argumento de método ou dentro de alguma outra expressão. Novamente, ajuda se você puder separar a parte específica do wait de todo o resto.

Imagine que você tem dois métodos, `GetHourlyRateAsync()` e `GetHoursWorked-Async()`, retornando `Task<decimal>` e `Task<int>`, respectivamente. Você pode ter esta declaração complicada:

```
AddPayment(aguarda funcionário.GetHourlyRateAsync() * aguarda  
timeSheet.GetHoursWorkedAsync(employee.Id));
```

As regras normais de avaliação de expressão C# se aplicam, e o operando esquerdo do operador \* deve ser completamente avaliado antes que o operando direito seja avaliado, portanto, a instrução anterior pode ser expandida da seguinte maneira:

```
Tarefa<decimal> hourlyRateTask = funcionário.GetHourlyRateAsync(); decimal hourlyRate =  
aguardar hourlyRateTask; Task<int> hoursWorkedTask =  
timeSheet.GetHoursWorkedAsync(employee.Id); int horasWorked = aguardar horasWorkedTask;  
AddPayment(taxahora * horas trabalhadas);
```

Essa expansão revela uma potencial ineficiência na instrução original — você poderia introduzir paralelismo nesse código iniciando *ambas* as tarefas (chamando ambos os métodos `Get...Async`) antes de aguardar qualquer uma delas.

No momento, o resultado mais útil é que você só precisa examinar o comportamento de `await` no contexto de um *valor*. Mesmo que esse valor tenha vindo originalmente de uma chamada de método, você pode ignorar essa chamada de método com o propósito de falar sobre assincronia.

#### COMPORTAMENTO VISÍVEL

Quando a execução atinge a expressão `await`, há duas possibilidades: a operação assíncrona que você está aguardando já foi concluída ou não.

Se a operação já foi concluída, o fluxo de execução é realmente simples – ele continua. Se a operação falhar e for capturada uma exceção para representar essa falha, a exceção será lançada. Caso contrário, qualquer resultado da operação será obtido — por exemplo, extrair a string de uma `Task<string>` — e você passará para a próxima parte do programa. Tudo isso é feito sem qualquer troca de contexto de thread ou anexação de continuações a nada.

Você deve estar se perguntando por que uma operação concluída imediatamente seria representada com assincronia em primeiro lugar. É um pouco como chamar o método `Count()` em uma sequência no LINQ: no caso geral, você pode precisar iterar sobre cada item da sequência, mas em algumas situações (como quando a sequência acaba sendo um `List<T>`) há uma otimização fácil disponível. É útil ter uma única abstração que cubra ambos os cenários, mas sem pagar um preço pelo tempo de execução. Como exemplo real no caso da API assíncrona, considere a leitura assíncrona de um fluxo associado a um arquivo no disco. Todos os dados que você deseja ler podem já ter sido buscados do disco para a memória, talvez como parte de uma solicitação de chamada `ReadAsync` anterior, portanto, faz sentido usá-los imediatamente, sem passar por todos os outros mecanismos assíncronos.

O cenário mais interessante é aquele em que a operação assíncrona ainda está em andamento. Nesse caso, o método aguarda de forma assíncrona a conclusão da operação e depois continua em um contexto apropriado. Essa “espera assíncrona” realmente significa que o método não está sendo executado. Uma continuação é anexada à operação assíncrona e o método retorna. Cabe à operação assíncrona garantir que o método seja retomado no thread correto - normalmente um thread de pool de threads (onde não importa qual thread é usado) ou o thread de UI onde isso faz sentido.

Do ponto de vista do desenvolvedor, parece que o método está apenas pausado enquanto a operação assíncrona é concluída. O compilador garante que todas as variáveis locais usadas no

método têm os mesmos valores que tinham antes da continuação - assim como acontece com os blocos iteradores.

Tentei capturar esse fluxo na Figura 15.2, embora os fluxogramas clássicos não tenham sido projetados tendo em mente o comportamento assíncrono.

Você poderia pensar na linha pontilhada como sendo outra linha que chega ao topo do fluxograma como alternativa. Observe que estou assumindo que o alvo da expressão await tem um resultado. Se você está apenas aguardando uma tarefa simples ou algo semelhante, “buscar resultado” na verdade significa “verificar se a operação foi concluída com sucesso”.

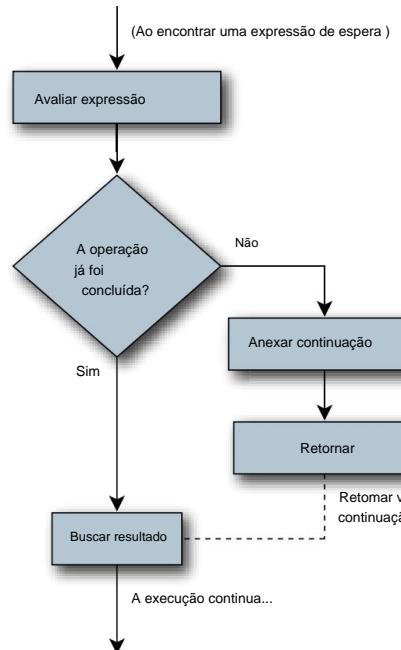
Vale a pena parar para pensar brevemente sobre o que significa “retornar” de uma situação assíncrona. método cronológico. Novamente, existem duas possibilidades:

ÿ Esta é a primeira expressão de espera pela qual você realmente teve que esperar, então você ainda tem o chamador original em algum lugar da sua pilha. (Lembre-se de que até que você realmente precise esperar, o método será executado de forma síncrona.)

ÿ Você já esperou por outra coisa, então está em uma continuação que foi chamada por *alguma coisa*. Sua pilha de chamadas quase certamente terá mudado significativamente em relação àquela que você viu quando entrou no método primeira vez.

No primeiro caso, você geralmente retornará um Task ou Task<T> ao chamador. Obviamente você ainda não tem o resultado real do método — mesmo que não haja nenhum valor para retornar como tal, você não sabe se o método será concluído sem exceções.

Por causa disso, a tarefa que você retornará deve estar incompleta.



**Figura 15.2** Modelo de tratamento de espera visível ao usuário

Neste último caso, o “algo” que o chama de volta depende do seu contexto. Por exemplo, em uma interface de usuário do Windows Forms, se você iniciasse seu método assíncrono no thread de interface do usuário e não mudasse deliberadamente dele, todo o método seria executado no thread de interface do usuário. Na primeira parte do método, você estará em algum manipulador de eventos ou outro – o que quer que tenha iniciado o método assíncrono. Mais tarde, porém, você será chamado de volta pela bomba de mensagens diretamente, como se estivesse usando `Control.BeginInvoke(continuação)`. Aqui, o código de chamada – seja a bomba de mensagens do Windows Forms, parte do mecanismo do pool de threads ou qualquer outra coisa – não se importa com sua tarefa.

Observe que até você atingir a primeira expressão de espera verdadeiramente assíncrona, o método será executado de forma totalmente síncrona. Chamar um método assíncrono não é como iniciar uma nova tarefa em um thread separado, e cabe a você garantir que sempre escreva métodos assíncronos para que eles retornem rapidamente. É certo que isso depende do contexto em que você está escrevendo o código, mas geralmente você deve evitar realizar trabalhos longos em um método assíncrono.

Separe-o em outro método para o qual você pode criar uma tarefa.

#### O USO DE MEMBROS DO PADRÃO AGUARDÁVEL

Agora que você entende o que precisa alcançar, é razoavelmente fácil ver como os membros do grupo esperam

padrão adequado são usados. A Figura 15.3 é realmente igual à figura 15.2, mas desenvolvida para incluir as chamadas ao padrão.

Quando está escrito assim, você deve estar se perguntando por que tanto alarido – por que vale a pena ter suporte a idiomas? Anexar uma continuação é mais complexo do que você imagina. Em casos muito simples, quando o fluxo de controle é inteiramente linear (fazer algum trabalho, aguardar alguma coisa, fazer mais algum trabalho, aguardar outra coisa), é muito fácil imaginar como seria a continuação como uma expressão lambda, mesmo que fosse não seja muito agradável. Contudo, assim que o código contém loops ou condições e você deseja

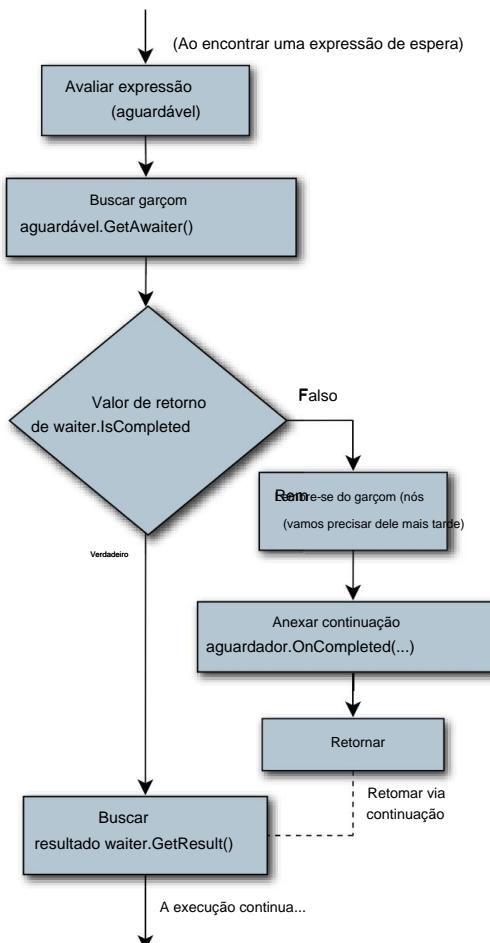


Figura 15.3 Tratamento de espera por meio do padrão aguardável

manter o código dentro de um método, a vida se torna muito mais complicada. É aqui que os benefícios do C# 5 realmente aparecem. Embora você possa argumentar que o compilador está apenas aplicando açúcar sintático, há uma enorme diferença na legibilidade entre criar manualmente as continuações e fazer com que o compilador faça isso para você.

Ao contrário de transformações simples, como propriedades implementadas automaticamente, o código gerado pelo compilador é bem diferente do que você provavelmente escreveria manualmente, mesmo quando o método assíncrono em si é quase trivial. Veremos um pouco dessa transformação em uma seção posterior, mas você já pode ver um pouco do “homem por trás da cortina” – esperamos que os métodos assíncronos estejam parecendo um pouco menos misteriosos agora.

Já descrevi as limitações dos tipos de retorno do método assíncrono e você viu como uma expressão await desembrulha os resultados da operação assíncrona por meio do método Get-Result() , mas não falei sobre a ligação entre os dois, ou como você pode retornar valores de métodos assíncronos.

### 15.3.5 Retornando de um método assíncrono

Você já viu um exemplo que retornou dados, mas vamos analisá-lo novamente, desta vez focando apenas no aspecto do retorno:

```
tarefa assíncrona estática<int> GetPageLengthAsync(string url) {  
  
    usando (cliente HttpClient = new HttpClient()) {  
  
        Tarefa<string> fetchTextTask = client.GetStringAsync(url); comprimento interno = (aguarde  
        fetchTextTask).Length; comprimento de retorno;  
  
    }  
}
```

Você pode ver que o tipo de comprimento é int, mas o tipo de retorno do método é Task<int>. O código gerado cuida do empacotamento para você, para que o chamador receba um Task<int>, que eventualmente terá o valor retornado do método quando ele for concluído. Um método que retorna apenas Task é como um método void normal — ele não precisa de uma instrução return e qualquer instrução return que ele *tenha* deve ser simplesmente return; em vez de tentar especificar um valor. Em ambos os casos, a tarefa também propagará qualquer exceção lançada no método assíncrono.

Esperançosamente, agora você já deve ter uma boa intuição sobre por que esse empacotamento é necessário: é quase certo que você retornará ao chamador antes de acertar a instrução return e precisará propagar a informação para esse chamador de alguma forma. Uma Task<T> (geralmente conhecida como *futuro* na ciência da computação) é a promessa de um valor — ou uma exceção — em um momento posterior.

Assim como acontece com o fluxo de execução normal, se a instrução return ocorrer dentro do escopo de um bloco try que possui um bloco finalmente associado (inclusive quando tudo isso acontece devido a uma instrução using ), a expressão usada para calcular o valor de retorno é *avaliado* imediatamente, mas não se torna o resultado da tarefa até que tudo tenha

foi limpo. Isso significa que se o bloco finalmente lançar uma exceção, você não consiga uma tarefa que seja bem-sucedida e fracasse – a coisa toda falhará.

Para reiterar um ponto que mencionei anteriormente, é a combinação de empacotamento automático e desembrulhar que faz com que o recurso assíncrono funcione tão bem com a composição. Você pode pensar nisso como sendo um pouco como LINQ: você escreve operações em cada *elemento* de um sequência no LINQ, e agrupar e desembrulhar significa que você pode aplicar essas operações para sequências e recuperar sequências. Em um mundo assíncrono, você raramente precisa lidar explicitamente com uma tarefa - em vez disso, você espera que a tarefa a consuma e produz um resultado automaticamente como parte do mecanismo do método assíncrono.

### 15.3.6 Exceções

É claro que as coisas nem sempre funcionam bem, e a forma idiomática de representar falhas no .NET são por meio de exceções. Assim como retornar um valor ao chamador, o tratamento de exceções requer suporte extra da linguagem. Quando você deseja lançar uma exceção, o chamador original do método assíncrono pode não estar na pilha; e quando você aguardar uma operação assíncrona que falhou, ela pode não ter sido executada no mesmo thread, então você precisa de uma maneira de organizar a falha. Se você pensa em fracasso como apenas outro tipo de resultado, faz sentido que exceções e valores de retorno sejam tratados de forma semelhante.

Nesta seção, veremos como as exceções cruzam ambos os limites em figura 15.1. Vamos começar com a fronteira entre o método assíncrono e a operação assíncrona que ele aguarda.

#### DESEMBRULHANDO EXCEÇÕES AO ESPERAR

Assim como o método `GetResult()` de um waiter destina-se a buscar o valor de retorno se existe um, ele também é responsável por propagar quaisquer exceções da operação assíncrona de volta ao método. Isto não é tão simples quanto parece, porque em um mundo assíncrono, uma única tarefa pode representar múltiplas operações, levando a múltiplas falhas. Embora outras implementações de padrões aguardáveis estejam disponíveis, é vale a pena considerar `Task` especificamente, pois é o tipo que você provavelmente estará aguardando pelo grande maioria das vezes.

A tarefa indica exceções de várias maneiras:

- ÿ O status de uma tarefa torna-se `Com Falha` quando a operação assíncrona falhou (e `IsFaulted` retorna verdadeiro). ÿ A propriedade `Exception` retorna uma `AggregateException` contendo todos os exceções (potencialmente múltiplas) que causaram a falha da tarefa ou nulas se a tarefa não tem culpa.
- ÿ O método `Wait()` lançará uma `AggregateException` se a tarefa terminar em um estado falho.
- ÿ A propriedade `Result` de `Task<T>` (que também aguarda a conclusão) também será lançada uma `AggregateException`.

Além disso, as tarefas suportam a ideia de cancelamento, via CancellationTokenSource e CancellationToken. Se uma tarefa for cancelada, o método Wait() e as propriedades Result lançarão uma AggregateException contendo uma OperationCanceledException (na prática, uma TaskCanceledException, que deriva de OperationCanceled-Exception), mas o status se tornará Canceled em vez de Faulted.

Quando você aguarda uma tarefa, se ela apresentar falha ou for cancelada, uma exceção será lançada, mas não a AggregateException. Em vez disso, por conveniência (na maioria dos casos), a primeira exceção *dentro* de AggregateException é lançada. Na maioria dos casos, isso é realmente o que você deseja. Está no espírito do recurso assíncrono permitir que você escreva código assíncrono que se parece muito com o código síncrono que você escreveria de outra forma.

Por exemplo, considere algo assim:

```
async Task<string> FetchFirstSuccessfulAsync(IEnumerable<string> urls) {  
  
    // TODO: Validar se realmente temos alguns URLs... foreach (string url in urls) {  
  
        tentar  
        {  
            usando (var cliente = new HttpClient()) {  
  
                retornar aguardar cliente.GetStringAsync(url);  
            }  
  
        } catch (exceção WebException) {  
  
            // TODO: registro, atualização de estatísticas etc.  
        }  
  
        } throw new WebException("Nenhum URL bem-sucedido");  
    }  

```

Por enquanto, ignore o fato de que você está perdendo todas as exceções originais e buscando todas as páginas sequencialmente. O que estou tentando enfatizar é que capturar WebException é o que você esperaria aqui: você está tentando uma operação assíncrona com um HttpClient e, se algo falhar, lançará uma WebException. Você quer pegar e lidar com isso... certo? Isso certamente parece o que você gostaria de fazer - mas, é claro, GetStringAsync() não pode lançar uma WebException para um erro como o tempo limite do servidor esgotado porque o método apenas *inicia* a operação. Tudo o que ele pode fazer é retornar uma tarefa com falha *contendo* uma WebException. Se você simplesmente chamassem Wait() na tarefa, uma AggregateException seria lançada, contendo a WebException dentro dela. O método GetResult do aguardador de tarefas apenas lança WebException e é capturado no código anterior.

Claro, isso *pode* perder informações. Se houver várias exceções em uma tarefa com falha, GetResult poderá lançar apenas uma delas e usará arbitrariamente a primeira. Talvez você queira reescrever o código anterior para que, em caso de falha, o chamador possa capturar uma AggregateException e examinar *todas* as causas da falha. É importante ressaltar que alguns

métodos de estrutura como `Task.WhenAll()` fazem exatamente isso — `WhenAll()` é um método que aguardará de forma assíncrona a conclusão de múltiplas tarefas (especificadas na chamada do método). Se alguma delas falhar, o resultado será uma falha que conterá as exceções de todas as tarefas com falha. Mas se você apenas aguardar a tarefa retornada por `WhenAll()`, verá apenas a primeira exceção.

Felizmente, não é preciso muito trabalho para corrigir isso. Você pode usar seu conhecimento do padrão `awaitable` e escrever um método de extensão em `Task<T>` para criar um `awaitable` especial que lançará a `AggregateException` original de uma tarefa. O código completo é um pouco complicado para a página impressa, mas sua essência é mostrada na listagem a seguir.

#### Listagem 15.2 Reempacotando múltiplas exceções de falhas de tarefas

```
public static AggregatedExceptionAwaitable WithAggregatedExceptions(
    Task tarefa) {
    return novo AggregatedExceptionAwaitable(tarefa);
}

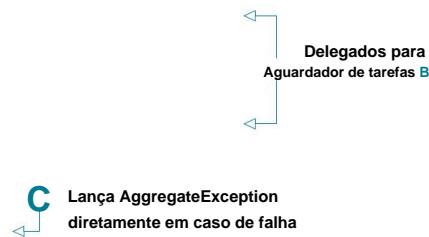
// Em AggregatedExceptionAwaitable public
AggregatedExceptionAwaiter GetAwaiter() {

    return novo AggregatedExceptionAwaiter(tarefa);
}

// Em AggregatedExceptionAwaiter public bool
IsCompleted {
    obter {retornar tarefa.GetAwaiter().IsCompleted; }

} public void OnCompleted(Continuação da ação) {
    task.GetAwaiter().OnCompleted(continuação);

} public void GetResult() {
    tarefa.Espere();
}
```



Você provavelmente desejará uma abordagem semelhante para `Task<T>`, usando `return task.Result;` em `GetResult()` em vez de chamar `Wait()`. O ponto importante é que você delegue ao aguardador normal da tarefa os bits que você não deseja tratar B, mas evite o comportamento usual de `GetResult()`, que é onde ocorre o desembrulhamento da exceção. No momento em que `GetResult` é chamado, você *sabe* que a tarefa está em um estado terminal, então a chamada `Wait()` C será concluída imediatamente — isso não viola a assincronia que você está tentando alcançar.

Para utilizar o código, basta chamar o método de extensão e aguardar o resultado, conforme mostrado a seguir.

### Listagem 15.3 Capturando múltiplas exceções como AggregateException

```
tarefa estática assíncrona privada CatchMultipleExceptions()
{
    Tarefa task1 = Task.Run(() => { throw new Exception("Mensagem 1"); });
    Tarefa task2 = Task.Run(() =>

    { throw new Exception("Mensagem 2"); });

    aguarde Task.WhenAll(task1, task2).WithAggregatedExceptions();
}

catch (AggregateException e) {

    Console.WriteLine("Capturadas {0} exceções: {1}",
        e.InnerExceptions.Count, string.Join(",",
        ", e.InnerExceptions.Select(x
            => x.Message)));
}
}
```

WithAggregatedExceptions() retorna seu aguardável personalizado; GetAwaiter() disso, por sua vez, fornece o aguardador personalizado, que suporta as operações que o compilador C# requer para aguardar o resultado. Observe que você poderia ter unido o awaitable e o awaiter - não há nada que diga que eles *precisam* ser de tipos diferentes - mas parece um pouco mais limpo separá-los.

Aqui está o resultado da listagem 15.3:

```
Capturou 2 exceções: Mensagem 1, Mensagem 2
```

É relativamente raro que você queira fazer isso – suficientemente raro que a Microsoft não inclua nenhum suporte para isso na estrutura – mas vale a pena conhecer essa opção.

Isso é tudo que você precisa saber sobre o tratamento de exceções para o segundo limite, pelo menos por enquanto. Mas e quanto ao primeiro limite, entre o método assíncrono e o chamador?

#### EXCEÇÕES DE EMBALAGEM AO JOGAR

Você pode muito bem ser capaz de prever o que está por vir: métodos assíncronos *nunca* lançam exceções diretamente quando chamados. Em vez disso, para métodos assíncronos que retornam Task ou Task<T>, quaisquer exceções lançadas dentro do método (incluindo aquelas propagadas de outras operações, sejam síncronas ou assíncronas) são simplesmente transferidas para a tarefa, como você já viu. Se o chamador aguardar a tarefa diretamente, ele receberá uma AggregateException contendo a exceção, mas se o chamador usar await, a exceção será desembrulhada da tarefa. Os métodos assíncronos que retornam void reportarão a exceção ao SynchronizationContext original – como isso lida com isso depende do contexto.<sup>7</sup>

---

<sup>7</sup> Discutiremos os contextos com mais detalhes na seção 15.6.4.

A menos que você realmente se importe com o empacotamento e desempacotamento de um contexto específico, você pode simplesmente capturar a exceção que o método assíncrono aninhado lançou. A listagem a seguir demonstra como isso parece familiar:

#### Listagem 15.4 Manipulando exceções assíncronas em um estilo familiar

```
tarefa assíncrona estática MainAsync() {
    Task<string> task = ReadFileAsync("arquivo lixo");
    tentar
    {
        string texto = aguarda tarefa;
        Console.WriteLine("Conteúdo do arquivo: {0}", texto);
    }
    catch (IOException e) {
        Console.WriteLine("IOException capturada: {0}", e.Message);
    }
}

tarefa assíncrona<string> ReadFileAsync(string nome do arquivo) {
    usando (var leitor = File.OpenText(nome do arquivo)) {
        retornar aguardar leitor.ReadToEndAsync();
    }
}
```

- A** Inicia o Leitura assíncrona **B**
- C** Espera por **C** o conteúdo
- D** Lida com falhas de E/S
- E** Abre o arquivo **E** sincronicamente

Aqui você obterá uma `IOException` na chamada `File.OpenText` **E** (a menos que você crie um arquivo chamado "arquivo lixo"), mas verá o mesmo caminho de execução se a tarefa retornada por `ReadToEndAsync` falhar. Dentro de `MainAsync`, a chamada para `ReadFileAsync` **B** acontece *antes de* você inserir o bloco `try`, mas é somente quando você aguarda a tarefa **C** que a exceção é vista pelo chamador e capturada pelo bloco `catch` **D**, assim como com o exemplo Web-Exception anteriormente. Novamente, ele se comporta de maneira muito familiar — exceto talvez pelo momento da exceção.

Assim como os blocos iteradores, isso é um pouco complicado em termos de validação de argumentos. Suponha que você queira fazer algum trabalho em um método assíncrono depois de validar que os parâmetros não possuem valores nulos. Se você validar os parâmetros como faria em um código síncrono normal, o chamador não terá nenhuma indicação do problema até que a tarefa seja aguardada. A listagem a seguir dá um exemplo disso.

#### Listagem 15.5 Validação de argumento quebrado em um método assíncrono

```
tarefa assíncrona estática MainAsync() {
    Task<int> tarefa = ComputeLengthAsync(null); Console.WriteLine("Buscou
    a tarefa"); comprimento interno = aguarda tarefa;
    Console.WriteLine("Comprimento: {0}",
    comprimento);
}
```

- A** Passa deliberadamente um argumento ruim
- B** aguarda o resultado

```
tarefa assíncrona estática<int> ComputeLengthAsync(string texto) {
    if (texto == nulo) {
        lançar novo ArgumentNullException("texto");
    }
    } aguardar Task.Delay(500);
    retornar texto.Comprimento;
}
```

Listagem 15.5 saídas Buscou a tarefa antes que ela falhasse. Na verdade, a exceção foi lançada de forma síncrona antes da saída ser gravada, pois não há expressões de espera antes da validação C, mas o código de chamada não a verá até aguardar a tarefa retornada B. Geralmente, para validação de argumento que pode ser sensata feito antecipadamente, sem levar muito tempo (ou incorrer em outras operações assíncronas), seria melhor se a falha fosse relatada imediatamente, antes que o sistema pudesse ter mais problemas. Como exemplo disso, HttpClient.GetStringAsync lançará uma exceção imediatamente se você passar uma referência nula.

Existem duas abordagens para forçar a exceção a ser lançada “avidamente” em C# 5.

A primeira é separar a validação do argumento da implementação, da mesma forma que você fez para os blocos iteradores na Listagem 6.9. A listagem a seguir mostra uma versão fixa do ComputeLengthAsync.

#### Listagem 15.6 Dividindo a validação de argumentos da implementação assíncrona

```
tarefa estática<int> ComputeLengthAsync(string texto) {
    if (texto == nulo) {
        lançar novo ArgumentNullException("texto");
    }
    } return ComputeLengthAsyncImpl(texto);
}

tarefa assíncrona estática<int> ComputeLengthAsyncImpl(string text) {
    aguarde Task.Delay(500); // Simula trabalho assíncrono real return text.Length;
}
```

Na listagem 15.6, ComputeLengthAsync em si não é um método assíncrono no que diz respeito à linguagem – ele não possui o modificador `async`. Ele é executado usando o fluxo de execução normal, portanto, se a validação do argumento no início do método lançar uma exceção, ele realmente lançará uma exceção. Se isso for aprovado, no entanto, a tarefa retornada será aquela criada pelo método `ComputeLengthAsyncImpl`, que é onde ocorre o trabalho real. Em um cenário mais real, `ComputeLengthAsync` provavelmente seria um método público ou interno, e `ComputeLengthAsyncImpl` deveria ser privado, porque pressupõe que a validação do argumento já foi executada.

A outra abordagem para validação antecipada é usar funções anônimas assíncronas—revisitaremos esse exemplo quando examinarmos os da seção 15.4.

Há um outro tipo de exceção que é tratado de maneira diferente no sistema assíncrono. métodos nous: cancelamento.

#### TRATAMENTO DE CANCELAMENTO

A Task Parallel Library (TPL) introduziu um modelo de cancelamento uniforme no .NET 4 usando dois tipos: `CancellationTokenSource` e `CancellationToken`. A ideia é que você pode criar um `CancellationTokenSource` e, em seguida, solicitar um `CancellationToken`, que é passado para uma operação assíncrona. Você só pode realizar o cancelamento na fonte, mas isso é refletido no token. (Isso significa que você pode distribuir o mesmo token para várias operações e não se preocupe com a possibilidade de elas interferirem umas nas outras.) Existem várias maneiras de usar o token de cancelamento, mas a mais idiomática abordagem é chamar `ThrowIfCancellationRequested`, que lançará `OperationCanceledException` se o token tiver sido cancelado e não fará nada caso contrário. O a mesma exceção é lançada por chamadas síncronas (como `Task.Wait`) se forem canceladas.

Como isso interage com métodos assíncronos não está documentado na especificação C# 5. De acordo com a especificação, se o corpo de um método assíncrono lançar *qualquer* exceção, a tarefa retornada pelo método estará em estado de falha. O significado exato de “com falha” é específico da implementação, mas na realidade, se um método assíncrono lança uma `OperationCanceledException` (ou um tipo de exceção derivado, como `TaskCanceledException`), a tarefa retornada terminará com o status `Canceled`. A listagem a seguir prova que realmente é uma exceção que faz com que a tarefa seja cancelada.

#### Listagem 15.7 Criando uma tarefa cancelada lançando `OperationCanceledException`

```
Tarefa assíncrona estática ThrowCancellationException()
{
    lançar nova OperationCanceledException();
}
...
Tarefa tarefa = ThrowCancellationException();
Console.WriteLine(tarefa.Status);
```

Isso gera `Cancelado` em vez de `Falha` que você esperaria da especificação. Se você `Wait()` na tarefa ou solicitar seu resultado (no caso de `Task<T>`), a exceção ainda é lançada dentro de uma `AggregateException`, então não é como se você precisasse começar explicitamente a verificar o cancelamento em todas as tarefas que você usa.

**VAI PARA AS CORRIDAS?** Você deve estar se perguntando se há uma condição de corrida na listagem 15.7. Afinal, você está chamando um método assíncrono e esperando imediatamente que o status seja corrigido. Se você estivesse realmente começando um novo tópico, isso seria perigoso... mas você não é. Lembre-se que antes do primeira expressão `await`, um método assíncrono é executado de forma síncrona - ainda executa o agrupamento de resultados e exceções, mas o fato de estar em um método assíncrono não significa necessariamente que haja mais threads envolvidos. O método `ThrowCancellationException` não contém nenhuma espera expressões, para que todo o método (toda uma linha dele) seja executado de forma síncrona; você saiba que você terá um resultado quando ele retornar. Visual Studio, na verdade

avisa sobre um método assíncrono sem nenhuma expressão de espera em isso, mas neste caso é exatamente o que você deseja.

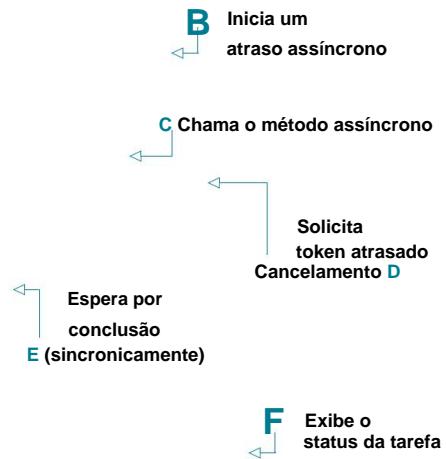
É importante ressaltar que se você aguardar uma operação cancelada, a Operation-CanceledException original será lançada. Isto significa que, a menos que você tome qualquer ação direta, o a tarefa retornada do método assíncrono também será cancelada - o cancelamento é propagado de forma natural.

A listagem a seguir fornece um exemplo um pouco mais realista de cancelamento de tarefa.

#### Listagem 15.8 Cancelamento de um método assíncrono através de um atraso cancelado

```
Tarefa assíncrona estática DelayFor30Seconds (token CancellationToken)
{
    Console.WriteLine("Aguardando 30 segundos...");
    aguarde Task.Delay(TimeSpan.FromSeconds(30), token);
}

...
var fonte = new CancellationTokenSource();
var tarefa = DelayFor30Seconds(source.Token);
source.CancelAfter(TimeSpan.FromSeconds(1));
Console.WriteLine("Status inicial: {0}", tarefa.Status);
tentar
{
    tarefa.Espere();
}
pegar (AggregateException e)
{
    Console.WriteLine("Capturado {0}", e.InnerException[0]);
}
Console.WriteLine("Status final: {0}", tarefa.Status);
```



Aqui você inicia uma operação assíncrona **C** que simplesmente chama `Task.Delay` para simular o trabalho real **B**, mas fornece um token de cancelamento. Desta vez, você realmente tem vários threads envolvidos: quando atinge a expressão `await`, o controle retorna para o método de chamada, ponto em que você solicita que o token de cancelamento seja cancelado em 1 segundo **D**. Você então espera (sincronicamente) que a tarefa termine **E**, esperando totalmente que ela termine terminar com uma exceção. Finalmente, você mostra o status da tarefa **F**.

A saída de 15.8 é assim:

```
Esperando 30 segundos...
Status inicial: WaitingForActivation
Capturado System.Threading.Tasks.TaskCanceledException: uma tarefa foi cancelada.
Situação final: Cancelado
```

Você pode pensar nisso como se o cancelamento fosse transitivo por padrão: se a operação **A** está aguardando a operação **B** e a operação **B** é cancelada, então você considera a operação **A** como sendo cancelado também.

Claro, você não precisa deixar assim. Você poderia ter capturado a `OperationCanceledException` no método `DelayFor30Seconds` e continuado a fazer outra coisa, ou retornou imediatamente, ou até mesmo lançou uma exceção diferente. De novo, o recurso assíncrono não remove o controle; está apenas fornecendo um comportamento padrão útil.

**CUIDADO ONDE VOCÊ FUNCIONA ISSO!** A Listagem 15.8 funciona bem em um aplicativo de console ou quando chamada a partir de um thread de pool de threads, mas se você executá-la em um thread UI do Windows Forms (ou qualquer outro contexto de sincronização de thread único), isso causará um impasse. Você pode ver por quê? Pense em qual thread o método `DelayFor-30Seconds` tentará retornar quando a tarefa atrasada for concluída e então pense em qual thread a chamada `task.Wait()` está sendo executada. Isto é um exemplo relativamente simples, mas o mesmo tipo de erro causou problemas para vários desenvolvedores quando eles começaram com código assíncrono. Fundamentalmente, o problema está em usar a chamada do método `Wait()`, ou o método Propriedade de resultado , ambas serão bloqueadas até que a tarefa relevante seja concluída. Não estou dizendo que você não deveria usá-los, mas você deveria pensar com muito cuidado sempre que você os *usar* . Normalmente, você deveria usar `await` para esperar assincronamente pelos resultados das tarefas.

Isso cobre praticamente o comportamento dos métodos assíncronos. É provável que a maioria do uso do recurso assíncrono no C# 5 será por meio de métodos assíncronos, mas eles fazem tem um irmão próximo...

## 15.4 Funções anônimas assíncronas

Não gastarei muito tempo em funções anônimas assíncronas. Como você provavelmente esperado, elas são uma combinação de dois recursos: funções anônimas (expressões lambda e métodos anônimos) e funções assíncronas (código que pode incluir aguardar expressões). Basicamente, eles permitem criar delegados<sup>8</sup> que representam operações assíncronas. Tudo o que você aprendeu até agora sobre métodos assíncronos aplica-se também a funções anônimas assíncronas.

Você cria uma função anônima assíncrona como qualquer outra função anônima método ou expressão lambda, apenas com o modificador assíncrono no início. Aqui está um exemplo:

```
Func<Task> lambda = async () => aguardar Task.Delay(1000);
Func<Task<int>> anonMethod = delegado assíncrono()
{
    Console.WriteLine("Iniciado");
    aguarde Task.Delay(1000);
    Console.WriteLine("Concluído");
    retornar 10;
};
```

O delegado que você cria deve ter uma assinatura com um tipo de retorno `void`, `Task` ou `Task<T>`, assim como acontece com um método assíncrono. Você pode capturar variáveis, como acontece com outras funções anônimas e adicione parâmetros. Além disso, a operação assíncrona não inicia até que o delegado seja invocado e múltiplas invocações criam vários operações. A invocação de delegado realmente *inicia* a operação; assim como antes, não é o `wait` que inicia uma operação, e você não *precisa* usar o `wait` com o resultado de uma função anônima assíncrona.

---

<sup>8</sup> Caso você esteja se perguntando, não é possível usar funções anônimas assíncronas para criar árvores de expressão.

A listagem a seguir mostra um exemplo um pouco mais completo (embora ainda inútil).

#### Listagem 15.9 Criando e chamando uma função assíncrona usando uma expressão lambda

```
Func<int, Task<int>> function = async x => {
    Console.WriteLine("Iniciando... x={0}", x); aguarde Task.Delay(x *
    1000); Console.WriteLine("Concluído...
    x={0}", x); *2;
    retornar x
};

Tarefa<int> primeiro = function(5); Tarefa<int>
segundo = função(3); Console.WriteLine("Primeiro
resultado: {0}", primeiro.Result); Console.WriteLine("Segundo resultado: {0}",
segundo.Result);
```

Escolhi deliberadamente os valores aqui para que a segunda operação seja concluída mais rapidamente que a primeira. Mas como você espera a primeira terminar antes de imprimir os resultados (usando a propriedade `Result`, que bloqueia até que a tarefa seja concluída - novamente, tome cuidado ao executar isso!), a saída será semelhante a esta:

```
Iniciando... x=5 Iniciando...
x=3 Concluído... x=3

Concluído...x=5
Primeiro resultado: 10
Segundo resultado: 6
```

Novamente, isso é exatamente o mesmo que se você colocasse o código assíncrono em um método assíncrono.

Acho difícil ficar muito entusiasmado com funções anônimas assíncronas, mas elas têm sua utilidade. Embora não seja possível incluí-los em expressões de consulta LINQ, ainda há casos em que você pode desejar realizar transformações de dados de forma assíncrona. Você só precisa pensar em todo o processo de uma maneira um pouco diferente.

Voltaremos a essa ideia quando discutirmos a composição, mas primeiro quero mostrar uma área onde elas são realmente muito úteis. Prometi anteriormente que mostraria outra maneira de executar a validação antecipada de argumentos no início de um método assíncrono. Você deve se lembrar que queríamos verificar a nulidade de um valor de parâmetro antes de iniciar a operação principal. A listagem a seguir é um método único que alcança o mesmo resultado que a implementação dividida na listagem 15.6.

#### Listagem 15.10 Validação de argumento usando uma função anônima assíncrona

```
tarefa estática<int> ComputeLengthAsync(string texto) {
    if (texto == nulo) {
        lançar novo ArgumentNullException("texto");
    }
    Func<Task<int>> func=async()=>
```

**B** Valida de forma totalmente síncrona

**C** Cria uma função assíncrona

```

{
    aguarde Task.Delay(500); retornar
    texto.Comprimento;
};

retornar função();           ← D Chama a função assíncrona
}

```

← Simula trabalho assíncrono real

Você notará que este não é um método assíncrono. Se fosse, a exceção seria encerrada em uma tarefa em vez de ser lançada imediatamente. Você ainda deseja retornar uma tarefa, portanto, após a validação B, basta encerrar o trabalho em uma função anônima assíncrona C, chamar o delegado D e retornar o resultado.

Embora isso ainda seja um pouco feio, é mais limpo do que dividir o método em dois.

Porém, há uma penalidade de desempenho a ser observada: esse pacote extra não vem de graça. Na maioria dos casos, tudo bem, mas se você estiver escrevendo uma biblioteca que pode ser usada em trabalhos de desempenho crítico, verifique o custo em seu cenário real antes de decidir qual abordagem usar.

SUPERIORIDADE VB? Na versão 11, o Visual Basic finalmente ganhou o suporte ao bloco iterador que o C# tem desde a versão 2. O atraso permitiu que a equipe refletisse sobre as deficiências do C# - a implementação do Visual Basic permite funções de iterador anônimas, permitindo o mesmo tipo de in-method dividido entre execução ansiosa e adiada. O recurso (ainda) não foi adicionado ao C#...

Agora você já viu praticamente tudo o que há em termos do recurso assíncrono no C# 5. No restante do capítulo, examinaremos alguns detalhes de implementação e, em seguida, veremos como aproveitar ao máximo o recurso. Tudo isso presumirá que você está razoavelmente confortável com tudo o que foi feito antes - se você ainda não experimentou nenhum código de amostra (ou, idealmente, seu próprio código experimental), agora seria um ótimo momento para fazê-lo.

Mesmo se você achar que entende a teoria, vale a pena brincar com o async e esperar para realmente ter uma ideia de como é programar a assincronia em um estilo um tanto síncrono.

### 15.5 Detalhes de implementação: transformação do compilador Lembro-me vividamente da

noite de 28 de outubro de 2010. Anders Hejlsberg estava apresentando async/await no PDC e, pouco antes de sua palestra começar, uma avalanche de material para download foi disponibilizada — incluindo um rascunho das mudanças na especificação C#, um CTP (Community Technology Preview) do compilador C# 5 e os slides que Anders estava apresentando. A certa altura, eu estava assistindo à palestra ao vivo e folheando os slides enquanto o CTP era instalado. Quando Anders terminou, eu estava escrevendo código assíncrono e testando coisas.

Nas semanas seguintes, comecei a desmontar partes — observando exatamente qual código o compilador estava gerando, tentando escrever minha própria implementação simplista da biblioteca que acompanha o CTP e, em geral, examinando-a de todos os ângulos. À medida que novas versões foram surgindo, descobri o que havia mudado e se tornou cada vez mais

confortável com o que estava acontecendo nos bastidores. Quanto mais eu via, mais apreciava a quantidade de código padrão que o compilador fica feliz em escrever em nosso nome.

É como olhar uma linda flor ao microscópio: a beleza ainda está aí para ser admirada, mas há muito mais do que pode ser visto à primeira vista.

Nem todo mundo é como eu, é claro. Se você quiser apenas confiar no comportamento que já descrevi e simplesmente confiar que o compilador fará a coisa certa, tudo *bem*. Alternativamente, você não perderá nada se pular esta seção por enquanto e voltar a ela mais tarde – nada do resto do capítulo depende disso.

É improvável que você precise depurar seu código até o nível que veremos aqui... mas acredito que esta seção lhe dará mais informações sobre como todo o recurso funciona. O padrão `awaitable` certamente faz muito mais sentido quando você olha o código gerado e verá alguns dos tipos que a estrutura fornece para ajudar o compilador. Alguns dos detalhes mais assustadores só estão presentes devido à otimização; o design e a implementação são cuidadosamente ajustados para evitar alocações desnecessárias de heap e trocas de contexto, por exemplo.

Como uma aproximação aproximada, fingiremos que o compilador C# realiza uma transformação de “código C# usando `async/await`” para “código C# sem usar `async/await`”. Na realidade, os componentes internos do compilador não estão disponíveis para nós e é mais do que provável que essa transformação ocorra em um nível inferior ao do C#. Certamente o IL gerado nem sempre pode ser expresso em C# não assíncrono, pois o C# tem restrições mais rígidas em relação ao controle de fluxo do que o IL. Mas é mais simples pensarmos nisso como C#, em termos de como o quebra-cabeça do código se encaixa.

O código gerado é parecido com uma cebola, com camadas de complexidade. Começaremos do lado de fora, avançando em direção à parte complicada – as expressões de espera e a dança dos aguardadores e das continuações.

### 15.5.1 Visão geral do código gerado

Ainda comigo? Vamos começar. Não vou entrar em *toda* a profundidade que poderia aqui - isso poderia preencher centenas de páginas - mas darei a você informações suficientes para entender a estrutura geral, e então você poderá ler as várias postagens que escrevi no blog nos últimos anos para obter detalhes mais complexos ou simplesmente escrever algum código assíncrono e descompilá-lo. Além disso, abordarei apenas métodos assíncronos — que incluirão todo o maquinário interessante e você não precisará lidar com a camada extra de indireção que as funções anônimas assíncronas apresentam.

**AVISO, CORAJOSO VIAJANTE - AQUI ESTÃO DETALHES DE IMPLEMENTAÇÃO!** Esta seção documenta alguns aspectos da implementação encontrada no compilador Microsoft C# 5, lançado com o .NET 4.5. Alguns detalhes mudaram substancialmente entre as versões CTP e na versão beta, e podem mudar novamente no futuro. Mas penso que é improvável que as *ideias* fundamentais mudem muito. Se você entender o suficiente desta seção para se sentir confortável de que não há mágica envolvida, apenas código realmente inteligente gerado pelo compilador, você deverá ser capaz de realizar quaisquer alterações futuras nos detalhes com facilidade.

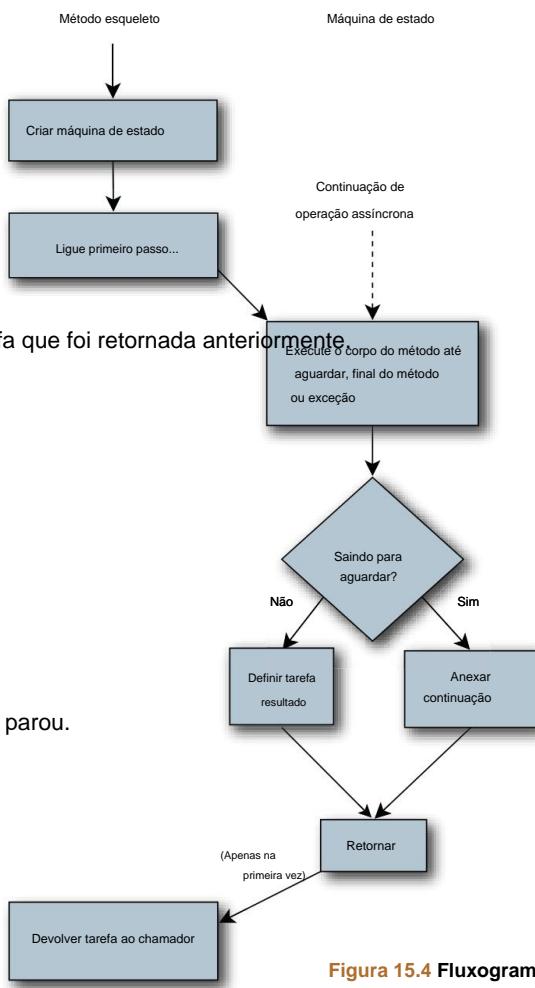
Como mencionei algumas vezes, a implementação (tanto nesta aproximação quanto no código gerado pelo compilador real) é basicamente na forma de uma *máquina de estados*. O compilador irá gerar uma estrutura aninhada privada para representar o método assíncrono e também deve incluir um método com a mesma assinatura daquele que você declarou. Eu chamo isso de método do *esqueleto* – não há muito nele, mas todo o resto depende dele.

O método esqueleto precisa criar a máquina de estado, fazê-la executar uma única etapa (onde uma *etapa* é qualquer código executado antes da primeira expressão de espera genuinamente aguardada) e, em seguida, retornar uma tarefa para representar o progresso da máquina de estado. (Não se esqueça de que até você atingir a primeira expressão de espera que realmente precisa esperar, a execução será síncrona.) Depois disso, o trabalho do método estará concluído - a máquina de estado cuidará de todo o resto, e as continuações anexadas a outras as operações assíncronas simplesmente dizem à máquina de estado para executar outra etapa. A máquina de estado sinaliza quando chega ao fim, fornecendo o resultado apropriado à tarefa que foi retornada anteriormente.

A Figura 15.4 mostra um diagrama de fluxo disso, da melhor forma que posso representá-lo.

É claro que o “executar” A etapa “corpo do método” só começa no início do método na primeira vez que ele é chamado, a partir do método esqueleto. Depois que, cada vez que você chega a esse bloco, é por meio de uma continuação, momento em que a execução continua efetivamente de onde parou.

Agora temos duas coisas para examinar: o método do esqueleto e a máquina estatal. Para a maior parte No restante desta seção, usarei um método assíncrono de amostra única, mostrado na listagem a seguir.



**Figura 15.4 Fluxograma do código gerado**

**Listagem 15.11 Método assíncrono simples para demonstrar transformações do compilador**

```
tarefa assíncrona estática<int> SumCharactersAsync(IEnumerable<char> texto) {
    total interno = 0;
    foreach (char ch no texto) {

        int unicode = ch; aguarde
        Task.Delay(unicode); total += unicode;

    } aguardar Task.Yield(); retorno
    total;
}
```

A Listagem 15.11 não faz nada de útil, mas na verdade estamos interessados apenas no fluxo de controle. Vale a pena observar alguns pontos antes de começarmos:

- ÿ O método possui um parâmetro (texto). ÿ Ele contém um loop para o qual você efetivamente precisa voltar quando a continuação ação é executada.
- ÿ Possui duas expressões de espera de tipos diferentes: Task.Delay retorna uma Tarefa, mas Task.Yield() retorna um YieldAwaitable.
- ÿ Possui variáveis locais óbvias (total, ch e unicode) que você precisará manter rastreamento de chamadas.
- ÿ Possui uma variável local implícita criada pela chamada de text.GetEnumerator(). ÿ Retorna um valor no final do método.

A versão original deste código tinha texto como parâmetro de string , mas o compilador C# sabe como iterar strings de maneira eficiente, usando a propriedade Length e o indexador, o que tornou o código descompilado mais complicado.

Não apresentarei o código descompilado completo , embora esteja na fonte para download. Nas próximas seções, veremos algumas das partes mais importantes. Você não verá exatamente esse código se você mesmo descompilar o código; Renomeei variáveis e tipos para que fiquem mais legíveis, mas é efetivamente o mesmo código.

Vamos começar com a parte mais simples: o método esqueleto.

### 15.5.2 Estrutura do método esqueleto

Embora o código do método esqueleto seja simples, ele oferece algumas dicas sobre as responsabilidades da máquina de estados. O método esqueleto gerado para a listagem 15.11 é assim:

```
[Depurador passo a passo]
[AsyncStateMachine(typeof(DemoStateMachine))] static Task<int>
SumCharactersAsync(IEnumerable<char> texto) {
    var máquina = new DemoStateMachine(); máquina.texto =
    texto; machine.builder =
    AsyncTaskMethodBuilder<int>.Create();
```

```

    máquina.estado = -1;
    machine.builder.Start(máquina de referência);
    retornar máquina.builder.Task;
}

```

O tipo `AsyncStateMachineAttribute` é apenas um dos novos atributos introduzidos para assíncrono. Na verdade, é para o benefício das ferramentas – é improvável que você precise consumir faça você mesmo, e você não deve começar a decorar seus próprios métodos com ele.

Você já pode ver três dos campos da máquina de estado:

- ÿ Um para o parâmetro (texto). Obviamente há tantos campos aqui como ali são parâmetros.
- ÿ Um para um `AsyncTaskMethodBuilder<int>`. Essa estrutura é efetivamente responsável por unir a máquina de estado e o método de esqueleto. Há um equivalente não genérico para métodos que retornam apenas `Task` e uma estrutura `AsyncVoidMethodBuilder` para métodos que retornam `void`.
- ÿ Um para estado, começando com um valor de -1. O valor inicial é sempre -1 e veremos o que vários valores possíveis significam mais tarde.

Dado que a máquina de estado é uma estrutura e `AsyncTaskMethodBuilder<int>` é uma struct, você ainda não realizou *nenhuma* alocação de heap intencionalmente. É perfeitamente possível pelas várias ligações que você está fazendo, é claro, mas vale a pena observando que o código tenta evitá-los tanto quanto possível. A natureza da assincronia significa que se alguma expressão de espera precisar realmente esperar, você precisará de muitos desses valores no heap, mas o código garante que eles sejam colocados em caixa apenas quando necessário ser. Tudo isso é um detalhe de implementação, assim como o heap e a pilha são detalhes de implementação, mas para que o assíncrono seja prático em tantas situações quanto possível, as equipes envolvidas na Microsoft trabalharam em estreita colaboração para reduzir as alocações para o mínimo.

A chamada `machine.builder.Start(ref machine)` é interessante. O uso de passagem por referência aqui permite evitar a criação de uma cópia da máquina de estado (e portanto, uma cópia do construtor) - isso serve tanto para desempenho quanto para correção. O compilador realmente gostaria de tratar tanto a máquina de estados quanto o construtor como classes, então `ref` é usado liberalmente em todo o código. Para usar interfaces, vários métodos levam o construtor (ou aguardador) como um parâmetro usando um parâmetro de tipo genérico que é restrito a implementar uma interface (como `IAsyncStateMachine` para o estado máquina). Isso permite que os membros da interface sejam chamados sem qualquer boxe sendo necessário. A ação do método é simples de descrever – ele torna o estado máquina dá o primeiro passo, de forma síncrona, retornando apenas quando o método tem concluído ou atingiu um ponto em que precisa aguardar uma operação assíncrona.

Uma vez concluída a primeira etapa, o método esqueleto solicita ao construtor o tarefa de retornar. A máquina de estado usa o construtor para definir resultados ou exceções quando termina.

### 15.5.3 Estrutura da máquina de estados

A estrutura geral da máquina de estado é bastante simples. Ele sempre implementa a interface IAsyncStateMachine (introduzida no .NET 4.5) usando implementação de interface explícita. Os dois métodos declarados por essa interface (MoveNext e SetStateMachine) são os únicos dois métodos que ela contém. Ele também tem vários campos – alguns privados, outros públicos.

Por exemplo, esta é a declaração recolhida da máquina de estado da listagem 15.11:

```
[CompilerGenerated]
estrutura privada DemoStateMachine: IAsyncStateMachine {
    texto público IEnumerable<char>;
    iterador público IEnumerator<char>; caridade pública;
    público interno total;
    público interno unicode;
    TaskAwaiter privado taskAwaiter; privado
    YieldAwaitable.YieldAwaiter rendimentoAwaiter; estado interno público;
    construtor público
    AsyncTaskMethodBuilder<int>; pilha de objetos privados;
    void IAsyncStateMachine.MoveNext() {...}
    [DebuggerHidden] void
    IAsyncStateMachine.SetStateMachine(máquina IAsyncStateMachine) { ... }
}
```

**B** Campos para parâmetros

**C** Campos para variáveis locais

**D** Campos para garçons

**E** Infraestrutura comum

Neste exemplo, dividi os campos em várias seções. Você já viu que o campo de texto B que representa o parâmetro original é definido pelo método esqueleto, junto com os campos construtor e estado, que são infraestruturas comuns compartilhadas por todas as máquinas de estado.

Cada variável local também possui seu próprio campo C, pois você precisa preservar os valores nas invocações do método MoveNext(). Às vezes, há variáveis locais que são usadas apenas entre duas expressões de espera específicas e realmente não precisam ser preservadas em campos, mas, na minha experiência, a implementação atual sempre as eleva a campos de qualquer maneira. Além de tudo, isso melhora a experiência de depuração, já que geralmente você não esperaria que variáveis locais perdessem seus valores, mesmo que não haja mais nada no código que as utilize.

Há um único campo para cada tipo de aguardador usado no método assíncrono, se forem tipos de valor, e um campo para todos os aguardadores que forem tipos de referência (em termos de tipo de tempo de compilação). Nesse caso, você tem duas expressões de espera que usam dois tipos diferentes de estruturas de espera, então você tem dois campos D. Se a segunda expressão de espera também usou um TaskAwaiter, ou se TaskAwaiter e YieldAwaiter eram ambas classes, você teria apenas um único campo. Apenas um garçom pode estar ativo por vez, então não importa que você só possa armazenar um valor por vez. Você tem que propagar

awaiters em expressões de espera para que, quando a operação for concluída, você possa obter o resultado.

Dos campos comuns de infraestrutura E, você já viu estado e construtor. Apenas como um lembrete, o estado é usado para acompanhar onde você chegou, para que a continuação possa chegar ao ponto certo no código. construtor é usado para várias coisas, incluindo a criação de um Task ou Task<T> para o método esqueleto retornar - uma tarefa que será preenchida com o resultado correto quando o método assíncrono terminar.

O campo stack é um pouco mais misterioso – é usado quando uma expressão await ocorre como parte de uma instrução que precisa acompanhar algum estado extra que não é representado por variáveis locais normais. Você verá um exemplo disso na seção 15.5.6 — ele não é usado na máquina de estado gerada para a listagem 15.11.

O método MoveNext() é onde toda a inteligência do compilador realmente entra em ação, mas antes de descrever isso, daremos uma olhada rápida em SetStateMachine. Tem a mesma implementação em todas as máquinas de estado e se parece com isto:

```
void IAsyncStateMachine.SetStateMachine(máquina IAsyncStateMachine) {
    construtor.SetStateMachine(máquina);
}
```

Em resumo, este método é usado para permitir que uma cópia em caixa da máquina de estados tenha uma referência a si mesma, dentro do construtor. Não entrarei em detalhes de como todo boxe é gerenciado – tudo o que você precisa entender é que a máquina de estado é encaixotada onde necessário, e os vários aspectos do mecanismo assíncrono garantem que, após o boxe, a única cópia em caixa é usada consistentemente. Isso é muito importante, pois estamos falando de um *tipo de valor mutável* (arrepiol!). Se algumas alterações fossem aplicadas a uma cópia da máquina de estados e algumas alterações fossem aplicadas a outra cópia, tudo desmoronaria muito rapidamente.

Se você quiser pensar de outra maneira - e isso será importante se você realmente começar a pensar sobre como as variáveis de instância da máquina de estado são propagadas - a máquina de estado é uma estrutura para evitar alocações desnecessárias de heap no início, mas a maioria das o código tenta *agir* como se fosse realmente uma classe. O malabarismo de referência em torno de Set-StateMachine faz tudo funcionar.

Certo... agora temos tudo no lugar, exceto o código real que estava no método assíncrono. Vamos mergulhar em MoveNext().

#### 15.5.4 Um ponto de entrada para governar todos eles

Se você descompilar um método assíncrono – e eu realmente espero que você o faça – você verá que o método MoveNext() na máquina de estado fica muito longo, muito rápido, principalmente em função de quantas expressões await você tem. Ele contém toda a lógica do método original e o delicado balé necessário para lidar com todas as transições de estado,<sup>9</sup> e algum código wrapper para lidar com o resultado ou exceção geral.

---

<sup>9</sup> Realmente parece uma dança, com passos intrincados que devem ser executados exatamente na hora e no lugar certos.

Ao escrever código assíncrono manualmente, você normalmente colocaria continuações em métodos separados: começaria em um método, depois continuaria em outro e talvez terminaria em um terceiro. Mas isso torna difícil lidar com o controle de fluxo, como loops, e é desnecessário para o compilador C#. Não é que a legibilidade do código gerado seja importante. A máquina de estado possui um único ponto de entrada, `MoveNext()`, que é usado desde o início e para as continuações de todas as expressões de espera. Cada vez que `MoveNext()` é chamado, a máquina de estado determina onde chegar no método por meio do campo de estado. Este é o ponto inicial lógico do método ou o fim de uma expressão de espera, quando você estiver pronto para avaliar o resultado. Cada máquina de estado é executada apenas uma vez. Efetivamente, há uma instrução `switch` baseada no estado, com diferentes casos correspondendo a instruções `goto` com rótulos diferentes.

O método `MoveNext()` normalmente se parece com isto:

```
void IStateMachine.MoveNext() {  
  
    // Para um método assíncrono declarado para retornar Task<int> int result;  
  
    tentar  
    {  
        bool doFinallyBodies = true; mudar (estado) {  
  
            // Código para pular para o lugar certo...  
        }  
  
        //Corpo principal do método  
  
    } catch (Exceção e) {  
  
        estado = -2;  
        construtor.SetException(e);  
        retornar;  
  
    } estado = -2;  
    construtor.SetResult(resultado);  
}
```

O estado inicial é sempre -1, e esse *também* é o estado quando o método está executando seu código (em vez de ser pausado enquanto aguarda). Quaisquer estados não negativos indicam o alvo de uma continuação. A máquina de estado termina no estado -2 quando é concluída. Em máquinas de estado criadas em configurações de depuração, você verá referência a um estado de -3 — nunca se espera que você realmente *acabe* nesse estado. Ele existe para evitar uma instrução `switch` degenerada, o que resultaria em uma experiência de depuração pior.

A variável `result` é definida durante o curso do método, no ponto em que o método assíncrono original tinha uma instrução de retorno. Isso é então usado na chamada do construtor `.SetResult()`, quando você atinge o final lógico do método. Mesmo os tipos não genéricos `AsyncTaskMethodBuilder` e `AsyncVoidMethodBuilder` possuem métodos `SetResult()`; o primeiro comunica o fato de que o método foi concluído

à tarefa retornada do método esqueleto, e o último sinaliza a conclusão do SynchronizationContext original. (As exceções são propagadas para o SynchronizationContext original da mesma maneira. É uma maneira um pouco mais suja de acompanhar o que está acontecendo, mas fornece uma solução para situações em que você realmente *precisa* ter métodos void.)

A variável `doFinallyBodies` é usada para determinar se algum bloco finalmente no código original (incluindo os implícitos das instruções `using` ou `foreach`) deve ser executado quando a execução sai do escopo de um bloco `try`. Conceitualmente, você só deseja executar um bloco finalmente ao sair do bloco `try` normalmente. Se você acabou de retornar do método antes de anexar uma continuação a um aguardador, o método será logicamente “pausado”, então você não deseja executar o bloco final. Quaisquer blocos finalmente apareceriam no corpo principal da seção de código do método, junto com o bloco `try` associado.

A maior parte do corpo do método é reconhecível em termos do método assíncrono original. É certo que você precisa se acostumar com todas as variáveis locais que agora aparecem como variáveis de instância na máquina de estado, mas isso não é muito difícil. As partes complicadas estão por toda parte em expressões de espera - como você poderia esperar.

### 15.5.5 Controle em torno de expressões de espera

Apenas como um lembrete, qualquer expressão de espera representa uma bifurcação em termos de possíveis caminhos de execução. Primeiro, o awaiter é buscado para a operação assíncrona que está sendo aguardada e, em seguida, sua propriedade `IsCompleted` é verificada. Se isso retornar verdadeiro, você poderá obter os resultados imediatamente e continuar. Caso contrário, você precisa fazer o seguinte:

- ÿ Lembre-se do garçom para mais tarde
- ÿ Atualize o estado para indicar de onde continuar ÿ Anexe uma continuação ao aguardador
- ÿ Retornar de `MoveNext()`, garantindo que quaisquer blocos finalmente *não* sejam executados

Então, quando a continuação for chamada, você precisará pular para o ponto certo, recuperar o aguardador e redefinir seu estado antes de continuar.

Por exemplo, a primeira expressão de espera na listagem 15.11 se parece com isto:

aguarde `Task.Delay(unicode)`:

O código gerado fica assim:

```
TaskAwaiter localTaskAwaiter = Task.Delay(unicode).GetAwaiter();
if (localTaskAwaiter.IsCompleted) {
    vá para DemoAwaitCompletion;

} estado = 0;
taskAwaiter = localTaskAwaiter;
construtor.AwaitUnsafeOnCompleted(ref localTaskAwaiter, ref this); doFinallyBodies = falso; retornar;
```

`DemoAwaitContinuação:`

```
localTaskAwaiter = taskAwaiter;
```

```

taskAwaiter = padrão(TaskAwaiter);
estado = -1;
DemoAwaitCompletion:
    localTaskAwaiter.GetResult();
    localTaskAwaiter = padrão(TaskAwaiter);

```

Se você estivesse aguardando uma operação que retornasse um valor, por exemplo, atribuir o resultado de await client.GetStringAsync(...) usando um HttpClient — a chamada Get-Result() perto do final seria onde você obteria o valor.

O método AwaitUnsafeOnCompleted anexa a continuação ao aguardador, e a instrução switch no início do método MoveNext() garantiria que quando MoveNext() é executado novamente, o controle passa para DemoAwaitContinuation.

**AGUARDO CONCLUÍDO VS. AWAITUNSAFEONCOMPLETED** Anteriormente, mostrei a você um conjunto nocional de interfaces, onde `IAwaiter<T>` estendeu `INotifyCompletion` com seu método `OnCompleted`. Há também um `ICriticalNotifyCompletion` interface, com um método `UnsafeOnCompleted`. A máquina de estado chama `builder.AwaitUnsafeOnCompleted` para aguardadores que implementam `ICritical-NotifyCompletion` ou `builder.AwaitOnCompleted` para aguardadores que apenas implementam `INotifyCompletion`. Veremos as diferenças entre esses duas chamadas na seção 15.6.4 quando discutirmos como o padrão aguardável interage com contextos.

Observe que o compilador limpa as variáveis locais e de instância do aguardador, portanto que pode ser coletado como lixo quando apropriado.

Uma vez que você possa identificar um bloco como esse como correspondendo a uma única expressão de espera, o código gerado realmente não será tão ruim para ser lido em formato descompilado. Pode ser mais instruções goto (e rótulos correspondentes) do que você esperaria, devido ao CLR restrições, mas entender o padrão de espera é o maior problema compreensão, na minha experiência.

Ainda há uma coisa que preciso explicar: a misteriosa variável de pilha no estado máquina...

### 15.5.6 Acompanhando uma pilha

Quando você pensa em um stack frame, você provavelmente pensa nas variáveis locais que você declarado no método. Claro, você deve estar ciente de algumas variáveis locais ocultas, como o iterador para um loop `foreach`, mas isso não é tudo o que vai para a pilha... pelo menos logicamente.<sup>10</sup> Em várias situações, existem expressões intermediárias que não podem ser usadas até que algumas outras expressões sejam avaliadas. Os exemplos mais simples destes são binários operações como adição e invocações de métodos.

Como um exemplo trivial, considere esta linha:

```
var x = y * z;
```

<sup>10</sup> Como Eric Lippert gosta de dizer, a pilha é um detalhe de implementação – algumas variáveis que você pode esperar que sejam na pilha, na verdade, acabam no heap, e algumas variáveis podem acabar existindo apenas em registros. Para o efeitos desta seção, estamos apenas falando sobre o que acontece logicamente na pilha.

No pseudocódigo baseado em pilha, é algo assim:

```
empurre você
empurre z
multiplicar
loja x
```

Agora suponha que você tenha uma expressão de espera aí:

```
var x = y      * aguarde z;
```

Você precisa avaliar y e armazená-lo em algum lugar antes de esperar z, mas você pode muito bem acabar retornando do método MoveNext() imediatamente, então você precisa de uma lógica pilha para armazenar y. Quando a continuação for executada, você poderá restaurar o valor e faça a multiplicação. Neste caso, o compilador pode atribuir o valor de y ao variável de instância de pilha. Isso envolve boxe, mas significa que você pode usar um único variável.

Esse é um exemplo simples. Imagine que você tivesse algo onde vários valores precisava ser armazenado, assim:

```
Console.WriteLine("{0}: {1}", x, aguarda tarefa);
```

Você precisa da string de formato e do valor de x em sua pilha lógica. Desta vez, o compilador cria um `Tuple<string, int>` contendo os dois valores e armazena esses referência na pilha. Assim como o garçom, você só precisa de uma pilha lógica por vez, portanto, não há problema em usar sempre a mesma variável.<sup>11</sup> Na continuação, os argumentos individuais podem ser obtidos da tupla e usados na chamada do método. O download o código-fonte contém uma descompilação completa deste exemplo, com ambas as instruções anteriores (`LogicalStack.cs` e `LogicalStackDecompiled.cs`).

A segunda instrução acaba usando um código como este:

```
string localArg0 = "{0} {1}";
int Arg1 local = x;
localAwaiter = tarefa.GetAwaiter();
if (localAwaiter.IsCompleted)
{
    vá para SecondAwaitCompletion;
}
var localTuple = new Tuple<string, int>(localArg0, localArg1);
pilha = localTuple;
estado = 1;
aguardador = localAwaiter;
construtor.AwaitUnsafeOnCompleted(ref awaiter, ref this);
doFinallyBodies = falso;
retornar;
SegundoAguardarContinuação:
localTuple = (Tupla<string, int>) pilha; localArg0 = localTuple.Item1;
localArg1 = localTuple.Item2; pilha = nulo;
```

---

<sup>11</sup> É certo que há momentos em que o compilador poderia ser mais inteligente quanto ao tipo da variável, ou evitar incluir uma se ela nunca for necessária, mas tudo isso pode ser adicionado em uma versão posterior, como uma otimização adicional.

```
localAwaiter = aguardador; aguardador  
= padrão(TaskAwaiter<int>); estado = -1;  
  
SegundoAwaitCompletion:  
    int localArg2 = localAwaiter.GetResult(); Console.WriteLine(localArg0,  
    localArg1, localArg2); // Audacios
```

As linhas em negrito aqui são aquelas que envolvem elementos da pilha lógica.

Neste ponto, provavelmente já fomos tão longe quanto precisávamos: se você chegou até aqui com sucesso, sabe mais sobre os detalhes do que está acontecendo nos bastidores do que 99% dos desenvolvedores provavelmente saberão. Tudo bem se você não seguiu tudo da primeira vez - se sua experiência for parecida com a minha ao ler o código dessas máquinas de estado, você vai querer esperar um pouco e depois voltar a isso .

### 15.5.7 Descobrindo mais

Quer ainda mais detalhes? Crie um descompilador. Eu recomendo que você use programas bem pequenos para investigar o que o compilador faz – é muito fácil se perder em um labirinto de pequenas continuações sinuosas, todas iguais, se você escrever algo não trivial. Pode ser necessário reduzir o nível de otimização que o descompilador executa para que ele mostre uma visão bastante próxima do código, em vez de uma interpretação. Afinal, um descompilador perfeito apenas reproduziria suas funções assíncronas, o que anularia todo o propósito do exercício!

O código gerado pelo compilador nem sempre pode ser descompilado em C# válido.

Sempre existe o problema de usar deliberadamente nomes indizíveis tanto para variáveis quanto para tipos, mas, mais importante, há alguns casos em que IL válida não tem equivalente direto em C#. Por exemplo, em IL é legítimo desviar para uma instrução que está dentro de um loop – afinal, IL nem sequer *tem* o conceito de loop como tal. Em C#, você não pode ir para um rótulo dentro de um loop de fora do loop, portanto, tal instrução não pode ser representada de forma totalmente correta. Mesmo o compilador C# não pode fazer tudo do seu jeito: o IL ainda tem algumas restrições nos alvos de salto, então você frequentemente descobrirá que o compilador precisa passar por uma série de saltos para chegar ao lugar certo.

Da mesma forma, tenho visto alguns descompiladores ficarem um pouco confusos quanto à ordem exata das instruções de atribuição na pilha lógica, ocasionalmente movendo a atribuição das variáveis temporárias (localArg0 e localArg1 , por exemplo) para o lado errado de a verificação `IsCompleted` . Acredito que isso se deva ao código não ser exatamente igual à saída normal do compilador C#. Não é tão ruim quando você sabe o que procurar, mas significa que ocasionalmente você provavelmente acabará caindo na IL.

## 15.6 Usando `async/await` de forma eficaz

Mostrei como as funções assíncronas se comportam e como elas são nos bastidores. Agora você é um especialista em programação assíncrona, certo? Obviamente

não.<sup>12</sup> Como muitos aspectos da programação, há muito a ser dito sobre experiência... e muito poucas pessoas tiveram muita experiência com funções assíncronas até aqui. Embora não possa lhe dar experiência, posso fornecer algumas dicas e sugestões que devem ser tornar sua vida um pouco mais fácil.

Enquanto escrevo isto, as equipes que mais sabem sobre programação assíncrona que usam C# 5 são aqueles da Microsoft, que o viveram e respiraram durante o desenvolvimento e receberam feedback de testadores beta e similares. Para esse fim, eu cuidadosamente recomendo o blog da equipe de programação paralela (<http://blogs.msdn.com/b/pfxteam/>), que tem muito mais conselhos do que tenho espaço para dar aqui.

Claro, isso não significa que não tenha algumas sugestões...

#### 15.6.1 O padrão assíncrono baseado em tarefas

Um dos benefícios do recurso de função assíncrona no C# 5 é que ele fornece uma abordagem consistente para assincronia. Mas isso poderia ser facilmente prejudicado se todos viessem criar suas próprias maneiras de usá-lo – como nomear métodos assíncronos, como as exceções devem ser levantadas e assim por diante. A Microsoft resolveu isso publicando o Padrão Assíncrono Baseado em Tarefas (TAP) — um conjunto de convenções que todos devem seguir. Isso é disponível como um documento independente (<http://mng.bz/B68W>) ou no MSDN como documento separado páginas (<http://mng.bz/4N39>).

É claro que a Microsoft também tem seguido isso: o .NET 4.5 contém um grande número de APIs assíncronas para todos os tipos de cenários. Assim como acontece com as convenções normais do .NET para nomenclatura, design de tipo e similares, se você seguir as mesmas convenções do restante da estrutura, outros desenvolvedores acharão seu código muito mais fácil de trabalhar.

O TAP é muito legível e tem apenas 38 páginas – recomendo fortemente que você leia o documento completo. No restante desta seção, abordarei o que considero ser o mais partes importantes.

Os métodos assíncronos devem terminar com o sufixo Async — GetAuthenticationTokenAsync, FetchUserPortfolioAsync e assim por diante. No .NET Framework isso tem já causou algumas colisões - WebClient já tinha métodos como DownloadStringAsync seguindo o padrão assíncrono baseado em eventos, e é por isso que o novo Os métodos baseados em TAP têm nomes um pouco feios de DownloadStringTaskAsync, UploadDataTaskAsync e similares. TaskAsync é o sufixo recomendado se você tiver suas próprias colisões de nomes também. Onde a assincronia é óbvia, o sufixo pode ser totalmente descartado - Task.Delay e Task.WhenAll são exemplos disso. Como um general regra, se todo o negócio do método for assíncrono, em vez de alcançar algum objetivo de negócios, provavelmente é seguro eliminar o sufixo.

Os métodos TAP geralmente retornam Task ou Task<T> — novamente, há exceções como como Task.Yield, onde entra o padrão aguardável, mas estes devem ser poucos e distantes entre. É importante ressaltar que a tarefa retornada de um método TAP deve ser quente. Em outras palavras, a operação que ela representa já deveria estar em andamento - o chamador não deveria

<sup>12</sup> Claro, você pode ser um especialista em programação assíncrona, mas apenas ler este capítulo não terá fiz isso por você.

precisa iniciá-lo manualmente. Para a maioria dos desenvolvedores, isso provavelmente parece óbvio, mas há existem outras plataformas onde a convenção é criar uma tarefa fria que não inicia até que você solicite explicitamente - um pouco como um bloco iterador em C#. Em particular, F# segue esta convenção e também é algo que você precisa considerar no Reactive Extensões (Rx).

Geralmente há quatro sobrecargas a serem consideradas ao criar um método assíncrono. Todos adotariam os mesmos parâmetros básicos, mas forneceriam opções diferentes em termos de relatórios de progresso e cancelamento. Suponha que você estávamos considerando desenvolver um método assíncrono que seria logicamente equivalente a um método síncrono como este:

```
Funcionário LoadEmployeeById(string id)
```

Seguindo as convenções da TAP , você pode fornecer qualquer um ou todos estes:

```
// NOTA À PRODUÇÃO: Consulte Jon sobre formatação.  
Não abrevie!  
Tarefa<Funcionário> LoadEmployeeById(string id)  
Task<Employee> LoadEmployeeById(string id, CancellationToken cancelamentoToken)  
Task<Employee> LoadEmployeeById(string id, IProgress<int> progresso)  
Task<Employee> LoadEmployeeById(string id,  
    CancellationToken cancelamentoToken, IProgress<int> progresso)
```

Aqui o `IProgress<int>` pode ser um `IProgress<T>` para qualquer tipo `T` apropriado para usar para relatórios de progresso. Por exemplo, se o seu método assíncrono encontrou uma coleção de registros e depois os processou um por um, você poderia aceitar uma `IProgress<Tuple<int, int>>`, que poderia relatar tanto o número de relatórios processados até o momento quanto o número de relatórios no total.

Eu evitaria tentar encaixar relatórios de progresso em operações onde realmente não faz sentido. O cancelamento é geralmente mais fácil de apoiar, porque muitos métodos de estrutura o suportam. Se o seu método assíncrono consiste basicamente em realizando diversas outras operações assíncronas (possivelmente com dependências), você pode achar mais fácil simplesmente aceitar um token de cancelamento e transmiti-lo posteriormente.

As operações assíncronas devem verificar erros de uso – normalmente argumentos inválidos – de forma síncrona. Isso é um pouco estranho, mas pode ser implementado usando um método dividido, conforme mostrado na seção 15.3.6, ou com um único método usando um função assíncrona anônima, conforme mostrado na seção 15.4. Embora seja tentador para validar argumentos preguiçosamente, você se amaldiçoará ao tentar resolver uma falha que é mais difícil de diagnosticar do que o necessário.

Operações baseadas em IO – onde você transfere um trabalho para um disco ou outro computador – são ótimos candidatos à assincronia, sem nenhuma desvantagem óbvia. As tarefas vinculadas à CPU são menos importantes. É fácil descarregar algum trabalho no pool de threads e até mesmo mais fácil no .NET 4.5 do que era antes, graças ao método `Task.Run` , mas fazer isso dentro do código da biblioteca estaria fazendo suposições em nome do chamador. Diferente os chamadores podem ter requisitos diferentes; se você apenas expor um síncrono método, você dá ao chamador a flexibilidade para trabalhar da maneira mais apropriada.

Eles podem iniciar uma nova tarefa, se necessário, ou chamá-la de forma síncrona, se desejarem que o thread atual esteja ocupado executando o método por algum tempo.

Tarefas que consistem em esperar por resultados de outros sistemas e depois processá-los de uma maneira potencialmente demorada são mais complicadas. Embora eu ache que é improvável que diretrizes rígidas e rápidas sejam úteis, é importante documentar o comportamento.

Se você acabar consumindo muita CPU no contexto do chamador, deixe isso bem claro.

Outra opção é evitar usar o contexto do chamador, usando o método `Task.ConfigureAwait`. Atualmente, esse método possui apenas um único parâmetro, `continueOnCapturedContext`, embora, para maior clareza, valha a pena usar um argumento nomeado para especificá-lo.

O método retorna uma implementação do padrão aguardável. Quando o argumento é verdadeiro, o awaitable se comporta exatamente como normal, portanto, se o método assíncrono for chamado em um thread de UI, por exemplo, a continuação após a expressão `await` ainda será executada no thread de UI. Isso é útil se você deseja acessar os elementos da IU. No entanto, se você não tiver nenhum requisito especial, poderá especificar `false` para o argumento; nesse caso, a continuação geralmente será executada no mesmo contexto em que a operação original foi concluída.<sup>13</sup> Para uma carga de trabalho mista que busca alguns dados e os processa e, em seguida, salva-o em um

banco de dados, você pode ter um código como este:

```
public static async Task<int> ProcessRecords() {
    Lista<Registro> registros = aguarda FetchRecordsAsync()
        .ConfigureAwait(continueOnCapturedContext: false);

    // ... tratamento de registro aqui ... await
    SaveResultsAsync(resultados)
        .ConfigureAwait(continueOnCapturedContext: false);

    // Informa ao chamador quantos registros foram processados return records.Count;
}
```

A maior parte desse método provavelmente será executada em um thread do pool de threads; isso é exatamente o que você deseja, pois não está fazendo nada que exija execução no thread original. (O jargão para isso é que a operação não tem nenhuma afinidade de thread.) Entretanto, isso não afeta o chamador; se um método de UI assíncrono aguardar o resultado da chamada de `ProcessRecords`, esse método assíncrono ainda continuará no thread de UI. É apenas o código dentro de `ProcessRecords` que declara que não se importa com seu contexto de execução.

Indiscutivelmente, você realmente não precisa chamar `ConfigureAwait` na segunda expressão de espera aqui, já que há muito pouco trabalho restante, mas em geral você deve usá-lo

---

<sup>13</sup> Geralmente, mas nem sempre. Os detalhes não são documentados explicitamente, mas há momentos em que você realmente não deseja executar no mesmo contexto. Você deve considerar que `ConfigureAwait(false)` está dizendo: "Não me importo onde a continuação é executada", em vez de anexá-la explicitamente a um contexto específico.

em cada expressão de espera , e é uma boa ideia adquirir o hábito de fazer isso de forma consistente. Se você quiser dar flexibilidade ao chamador sobre o contexto no qual o método é executado, você poderia potencialmente tornar isso um parâmetro para o método assíncrono.

Observe que `ConfigureAwait` afeta apenas a parte *de sincronização* do contexto de execução. Outros aspectos, como a representação, são propagados independentemente, como você verá na seção 15.6.4.

**TPL DATAFLOW** Embora TAP seja apenas um conjunto de convenções e alguns exemplos, a Microsoft também criou uma biblioteca separada chamada “TPL Dataflow”, que está disponível para fornecer blocos de construção de nível superior para cenários específicos, particularmente aqueles que podem ser modelados usando padrões produtor/consumidor. A maneira mais simples de começar é provavelmente por meio do pacote NuGet (`Microsoft .Tpl.Dataflow`). O uso é gratuito e há muitas orientações sobre ele. Mesmo que você não o use diretamente, vale a pena dar uma olhada, apenas para ter uma ideia de como programas paralelos *podem* ser projetados.

Mesmo sem nenhuma biblioteca extra, você ainda pode construir código assíncrono elegante seguindo os princípios normais de design, e um dos aspectos mais importantes disso é a composição.

### 15.6.2 Compondo operações assíncronas

Uma das coisas que mais adoro na assincronia do C# 5 é a forma como ela é composta de forma tão natural. Isto se manifesta de duas maneiras diferentes. Obviamente, os métodos assíncronos retornam tarefas e normalmente envolvem a chamada de outros métodos que retornam tarefas.

Podem ser operações assíncronas diretas (a parte inferior da cadeia, por assim dizer) ou apenas métodos mais assíncronos. Todo o empacotamento e desempacotamento necessários para transformar resultados em tarefas e vice-versa são feitos pelo compilador.

A outra forma de composição é a maneira como você pode criar blocos de construção neutros em termos de operação para controlar como as tarefas são tratadas. Esses blocos de construção não precisam saber nada sobre o que as tarefas estão fazendo — eles permanecem puramente no nível de abstração de `Task<T>`. Eles são um pouco parecidos com os operadores LINQ , mas trabalham em tarefas em vez de sequências. Alguns blocos de construção estão integrados à estrutura, mas você pode escrever os seus próprios.

#### COLETANDO RESULTADOS EM UMA ÚNICA CHAMADA

Como exemplo, vamos considerar a tarefa de buscar muitos URLs. Na seção 15.3.6 você fez isso um de cada vez, parando assim que obteve sucesso. Suponha que desta vez você queira iniciar as solicitações em paralelo e, em seguida, registrar o resultado para cada URL. Lembrando que os métodos assíncronos retornam tarefas já em execução, você pode iniciar uma tarefa para cada URL com bastante facilidade:

```
var tarefas = urls.Select(url => {
```

```
    usando (var cliente = new HttpClient()) {
```

```
    retornar aguardar cliente.GetStringAsync(url);
}
}).Listar();
```

Observe que `ToList()` é necessário para materializar a consulta LINQ. Isso garante que você inicie cada tarefa uma e apenas uma vez — caso contrário, cada vez que você iterasse as tarefas, você iniciaria outro conjunto de buscas. (O código seria ainda mais simples se você não se importasse em descartar o `HttpClient`, mas mesmo com esse problema, não é tão ruim.)

O TPL fornece um método `Task.WhenAll` que combina os resultados de muitas tarefas, cada uma fornecendo um único resultado, em uma única tarefa com vários resultados. A assinatura da sobrecarga que você usará é semelhante a esta:

```
tarefa estática<TResult[]> WhenAll<TResult>(IEnumerable<Task<TResult>> tarefas)
```

Essa é uma declaração assustadora, mas o propósito do método é bastante simples quando você começa a usá-lo. Você tem um `List<Task<string>>`, então pode escrever isto:

```
string[] resultados = aguardar Task.WhenAll(tarefas);
```

Isso irá esperar até que todas as tarefas sejam concluídas e reunir os resultados em uma matriz. Esta é uma das ocasiões em que se diversas tarefas lançam exceções, apenas a primeira será lançada imediatamente, mas você sempre pode iterar sobre as tarefas para descobrir quais delas falharam e por quê, ou usar o método de extensão `WithAggregatedExceptions` mostrado na listagem 15.2.

Se você se preocupa apenas com o retorno da primeira solicitação, existe outro método chamado `Task.WhenAny` que não espera pela conclusão bem-sucedida da primeira tarefa; apenas espera que a primeira tarefa atinja um estado terminal.

Nesse caso, você pode querer algo um pouco diferente. Talvez seja mais útil relate todos os resultados assim que eles chegarem.

#### COLETANDO RESULTADOS À MEDIDA QUE CHEGAM

Embora `Task.WhenAll` fosse um exemplo de bloco de construção transformacional integrado ao .NET, o próximo exemplo mostra como você pode criar seus próprios métodos de maneira semelhante. A documentação do TAP fornece alguns exemplos de código muito semelhantes, criando um método chamado `Interleaved`, e veremos uma versão ligeiramente alternativa.

A ideia da listagem 15.12 é permitir que você passe uma sequência de tarefas de entrada, e o método retornará uma sequência de tarefas de saída. Os resultados das tarefas nas duas sequências serão os mesmos, mas com uma diferença crucial: as tarefas de saída serão concluídas na ordem em que são fornecidas, para que você possa aguardá-las uma de cada vez e saber que receberá o resultado. resultados assim que estiverem disponíveis. Agora, isso pode parecer mágica — para mim parece — então vamos dar uma olhada no código e ver como ele funciona.

**Listagem 15.12 Transformando uma sequência de tarefas em uma nova coleção em ordem de conclusão**

```
    público estático IEnumerable<Task<T>> InCompletionOrder<T>
        (esta fonte IEnumerable<Task<T>>)
    {
        var entradas = source.ToList(); var caixas =
            inputs.Select(x => new TaskCompletionSource<T>())
                .Listar();
        int índiceAtual = -1; foreach (var
            tarefa nas entradas) {

            tarefa.ContinueWith(concluído => {

                var nextBox = caixas[Interlocked.Increment(ref currentIndex)]; PropagateResult(concluído, nextBox);

            }, TaskContinuationOptions.ExecuteSynchronously);
        }
    }
}
```

A Listagem 15.12 depende de um tipo muito importante no TPL — `TaskCompletionSource`<T>. Este tipo permite criar uma tarefa sem resultado ainda e fornecer um resultado (ou uma exceção) posteriormente. Isso é construído na mesma infraestrutura subjacente que `AsyncTaskMethodBuilder`<T> usa para fornecer uma tarefa para um método assíncrono retornar, permitindo que a tarefa seja preenchida com o resultado quando o corpo do método for concluído.

Para explicar os nomes um pouco curiosos das variáveis, muitas vezes penso nas tarefas como caixas de papelão, com a promessa de que em algum momento elas terão um valor dentro (ou uma falha). Um `TaskCompletionSource`<T> é como uma caixa com um buraco atrás — você pode entregá-la a alguém e, mais tarde, furtivamente inserir o valor no buraco.<sup>14</sup> É exatamente isso que o método `PropagateResult` faz — não é muito interessante, então eu omiti aqui, mas basicamente ele propaga o resultado de um `Task`<T> concluído em um `TaskCompletionSource`<T>. Se a tarefa original for concluída normalmente, o valor de retorno será copiado na origem de conclusão da tarefa. Se a tarefa original falhar, a exceção será copiada na fonte de conclusão da tarefa. Se a tarefa original foi cancelada, a origem da conclusão da tarefa será cancelada.

A parte realmente inteligente aqui (e não recebo nenhum crédito por isso - a sugestão foi enviada para mim por e-mail) é que quando esse método é executado, ele não sabe qual `TaskCompletionSource`<T> corresponderá a qual tarefa de entrada. Em vez disso, ele simplesmente anexa a mesma continuação a cada tarefa, e essa continuação diz: “Encontre o próximo `TaskCompletionSource`<T> (incrementando atomicamente um contador) e propague o resultado”. Em outras palavras, as caixas são preenchidas na ordem de saída à medida que as tarefas originais são concluídas.

<sup>14</sup> Qualquer semelhança com a física quântica é mera coincidência e não serei responsabilizado por quaisquer experimentos envolvendo `Task`<Cat>.

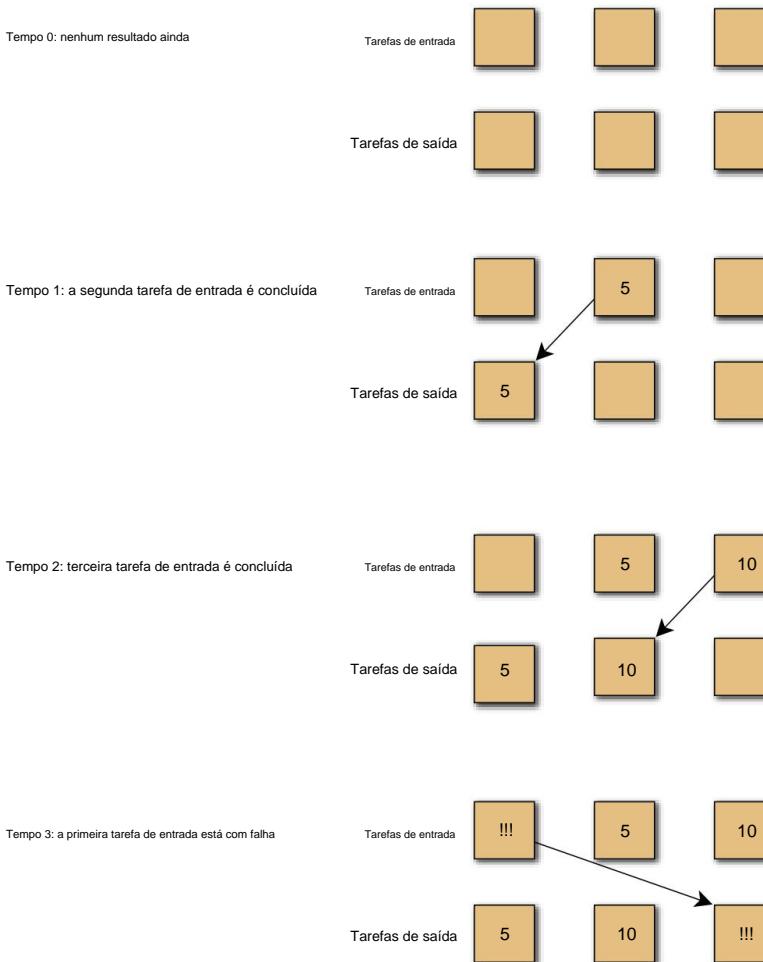


Figura 15.5 Visualização do pedido

A Figura 15.5 mostra três tarefas de entrada e as tarefas de saída correspondentes retornadas por o método. As tarefas de saída são concluídas na ordem retornada, mesmo que a entrada as tarefas são concluídas em uma ordem diferente.

Com este maravilhoso método de extensão implementado, você pode escrever o seguinte código, que pega uma coleção de URLs, lança solicitações para cada um deles em paralelo, escreve o comprimento de cada página à medida que ela é concluída e retorna o comprimento total.

#### Listagem 15.13 Exibindo comprimentos de página conforme os dados são retornados

```
tarefa assíncrona<int> ShowPageLengthsAsync(params string[] urls)
{
    var tarefas = urls.Select(url assíncrono =>
    {
        // Código para executar a solicitação para a URL e retornar o comprimento
    });
}
```

```
usando (var cliente = new HttpClient())
{
    retornar aguardar cliente.GetStringAsync(url);
}
}).Listar();

total interno = 0;
foreach (var tarefa em tarefas.InCompletionOrder())
{
    string página = aguarda tarefa;
    Console.WriteLine("Obteve o comprimento da página {0}", page.Length);
    total += página.Comprimento;
}
retorno total;
```

Existem dois pequenos problemas com a listagem 15.13:

↳ Assim que uma tarefa falha, toda a operação assíncrona falha sem indicação dos resultados restantes. Isso pode estar bem, ou você pode querer ter certeza que você registre todas as falhas. (Ao contrário do .NET 4, deixar exceções de tarefas passarem despercebidas não interromperá o processo por padrão, mas você deve pelo menos pensar sobre o que você deseja que aconteça com outras tarefas.) ↳

Você perde o controle de qual página acompanha qual URL.

Ambos são razoavelmente facilmente corrigidos com um pouco mais de código e podem até sugerir mais blocos de construção reutilizáveis. O objetivo de mostrar esses exemplos não era examinar os requisitos individuais – era abrir a mente para as possibilidades proporcionadas pela composição.

Interleaved não é o único exemplo no white paper da TAP – tem muitas ideias, com código de exemplo para ajudá-lo.

### 15.6.3 Teste de unidade de código assíncrono

Estou um pouco nervoso por começar a escrever esta seção. No momento, eu não acredito que a comunidade tem experiência suficiente para chegar a uma definição definitiva respostas sobre como testar código assíncrono. Tenho certeza que haverá alguns erros ao longo do caminho, e sem dúvida que serão exploradas diversas abordagens concorrentes. O ponto importante é que, assim como o código síncrono, se você projetar para testabilidade a partir de no início, você pode testar a unidade do código assíncrono de maneira eficaz.

#### INJETANDO ASSÍNCRONIA COM SEGURANÇA

Nesta seção apresentarei uma abordagem para situações em que você é capaz de controlar o operações assíncronas das quais seu próprio código assíncrono depende. Não tenta para resolver as dificuldades de testar código que usa `HttpClient` e tipos igualmente difíceis de falsificar, mas isso não é novidade - se você tiver dependências difíceis de usar em testes, você sempre enfrentará problemas.

Suponha que você queira testar o código de “ordenação mágica” da seção anterior. Você deseja ser capaz de criar tarefas que serão concluídas em uma ordem especificada e (pelo menos em

alguns testes) certifique-se de que você pode realizar afirmações entre as conclusões das tarefas. Além disso, você gostaria de fazer tudo isso sem o envolvimento de outros threads – você deseja o máximo de controle e previsibilidade possível. Em essência, você deseja controlar o tempo.

Minha solução para isso é essencialmente falsificar o tempo usando uma classe TimeMachine que fornece uma maneira de avançar o tempo programaticamente com tarefas agendadas que são concluídas de maneiras específicas em horários específicos. Combine isso com um Contexto de Sincronização que é efetivamente uma versão bombeada manualmente do familiar bombeamento de mensagens do Windows Forms, e você terá um equipamento de teste bastante razoável. Não mostrarei todo o código da estrutura usado para hospedar isso, pois é um pouco longo e relativamente chato, mas está tudo no código de exemplo. Vou mostrar alguns testes.

Vamos começar com o caso de sucesso geral: se você programar três tarefas para serem concluídas nos tempos 1, 2 e 3 e chamar InCompletionOrder com essas tarefas em uma ordem diferente, ainda deverá obter os resultados em ordem:

```
[TestMethod] public
void TasksCompleteInOrder() {

    var tardis = new TimeMachine(); var tarefa1 =
    tardis.ScheduleSuccess(1, "t1"); var tarefa2 = tardis.ScheduleSuccess(2,
    "t2"); var tarefa3 = tardis.ScheduleSuccess(3, "t3");

    var tarefasOutOfOrder = new[] { tarefa2, tarefa3, tarefa1 };

    tardis.ExecutelnContext(advancer => {

        var inOrder = tarefasOutOfOrder.InCompletionOrder().ToList(); avanço.AdvanceTo(3);
        Assert.AreEqual("t1",
        inOrder[0].Result); Assert.AreEqual("t2", inOrder[1].Result);
        Assert.AreEqual("t3", inOrder[2].Result); });

}
```

O método ExecutelnContext substitui temporariamente o SynchronizationContext do thread atual por um ManuallyPumpedSynchronizationContext (também no código de exemplo) e, em seguida, fornece um avanço para o delegado especificado pelo argumento do método. Esse adiantador pode ser usado para avançar o tempo em quantidades específicas, com tarefas sendo concluídas (e executando continuações) nos momentos apropriados. Neste teste, você apenas avança até que todos sejam concluídos.

Aqui está um segundo teste que demonstra que você pode controlar o tempo de uma forma mais detalhada:

```
// Etapas de configuração omitidas, que são iguais ao teste anterior. tardis.ExecutelnContext(advancer => {

    var inOrder = tarefasOutOfOrder.InCompletionOrder().ToList();

    Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[0].Status); Assert.AreEqual(TaskStatus.WaitingForActivation,
    inOrder[1].Status);
```

```
Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);
avanço.Avanço();
Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[0].Status);
Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[1].Status);
Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

avanço.Avanço();
Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[1].Status);
Assert.AreEqual(TaskStatus.WaitingForActivation, inOrder[2].Status);

avanço.Avanço();
Assert.AreEqual(TaskStatus.RanToCompletion, inOrder[2].Status);
});

});
```

Aqui você pode ver as tarefas de saída sendo concluídas na ordem correta.

Você pode estar se perguntando por que os tempos aqui são apenas números inteiros – você pode ter esperava que DateTime e TimeSpan se envolvessem. Isso é deliberado – a única linha do tempo que você realmente tem é a artificial estabelecida pela máquina do tempo, e os únicos pontos interessantes no tempo são aqueles onde as tarefas são concluídas.

Claro, o método que você está testando é um pouco incomum aqui de duas maneiras:

- ÿ Na verdade, não é implementado com async. ÿ São dadas tarefas diretamente como argumentos.

Se você estivesse testando um método assíncrono mais focado nos negócios, provavelmente agendaria todos os resultados para suas dependências, adiantaria o tempo para concluí-los todos e então verifique o resultado da tarefa retornada. Você teria que ser capaz de fornecer falsificações ao seu código de produção da maneira normal – a única diferença que a assincronia faz aqui é que em vez de usar stubs ou mocks para retornar os resultados diretos das chamadas, você peça-lhes que devolvam as tarefas produzidas pelo TimeMachine. Todos os benefícios normais de a inversão de controle ainda se aplica – você só precisa de uma maneira de criar tarefas apropriadas.

Essa ideia única claramente não será uma panacéia, mas espero que pelo menos tenha persuadido você da possibilidade de testar a unidade de código assíncrono sem chamadas arbitrárias para Thread.Sleep e o risco constante de instabilidade do teste.

#### EXECUÇÃO DE TESTES ASSÍNCRONOS

Os testes na seção anterior são totalmente síncronos. Você não usa assíncrono ou aguardam nos próprios testes. Quando você está usando a classe TimeMachine para todos os seus testes, isso é bastante razoável, mas em outros casos você pode querer escrever testes métodos decorados com async.

Você pode fazer isso facilmente:

```
[Teste] // NUnit TestAttribute
público assíncrono void BadTestMethod()
{
    //Código usando wait
}
```

Isso será compilado em qualquer estrutura de teste normal... mas pode não fazer o que você espera. Em particular, você pode acabar com todos os seus testes sendo iniciados em paralelo e, possivelmente, "terminando" antes que eles consigam afirmar alguma coisa.

Acontece que o NUnit suporta testes assíncronos a partir da versão 2.6.2, e o método anterior funcionaria devido a alguma inteligência na implementação, mas se você tentasse executá-lo em versões anteriores, o teste seria iniciado e concluído, como no que diz respeito ao executor de teste, assim que atingir a primeira espera "lenta". Quaisquer falhas posteriores no método acabariam sendo relatadas ao SynchronizationContext do teste, que pode não estar esperando por isso.

Para estruturas de teste que suportam testes assíncronos, uma abordagem muito melhor é faça esses testes retornarem Task, assim:

```
[Teste]  
tarefa assíncrona pública GoodTestMethod() {  
  
    //Código usando wait  
}
```

Agora é muito mais fácil para a estrutura de teste saber quando seus testes foram concluídos e verificar se há falhas. Tem o benefício adicional de testar estruturas que não

que suportam testes assíncronos podem nem tentar executá-los, em vez disso reportam um aviso, o que é muito melhor do que executar os testes incorretamente. Enquanto escrevo isto, as versões mais recentes do NUnit, do xUnit e do Visual Studio Unit Testing Framework (também conhecido informalmente como MS Test) suportam testes assíncronos – outras estruturas também podem fazer o mesmo. Verifique a estrutura e a versão específicas que deseja usar antes de começar a escrever tais testes.

Você também deve ter cuidado com a possibilidade de impasses. Ao contrário dos testes de máquina do tempo na seção anterior, você provavelmente não deseja que todas as continuações sejam executadas em um único thread, a menos que esse thread também esteja bombeando como um thread de UI faria.

Às vezes, você controla todas as tarefas envolvidas e pode raciocinar para usar um contexto de thread único... outras vezes, você precisa ser mais cuidadoso e pode querer que vários threads sejam capazes de disparar continuações, desde que seu código de teste não seja executado em paralelo consigo mesmo. Eu ficaria nervoso com isso para testes de unidade, mas se você estiver usando o mesmo tipo de estrutura para testes funcionais, testes de integração ou até mesmo testes de produção, normalmente desejará que seus testes sejam executados em tarefas reais, em vez de no falsificações fornecidas pela máquina do tempo.

Estou confiante de que com o tempo a comunidade desenvolverá ótimas ferramentas para nos ajudar a testar cada vez mais nosso código. Estou convencido de que uma proporção significativa do código futuro será naturalmente assíncrono e tenho certeza absoluta de que não quero escrever esse código sem testes. Estamos quase terminando a assincronia agora, mas prometi anteriormente que voltaria àquela interessante chamada de método `AwaitUnsafeOnCompleted` no código gerado.

### 15.6.4 O padrão aguardável redux

Na seção 15.3.3 mostrei algumas interfaces imaginárias que dão a ideia básica correta sobre o padrão aguardável. Mesmo quando expliquei que isso não era bem realidade, eu falsifiquei um pouco. A menos que você esteja implementando o próprio padrão awaitable ou observando atentamente o código descompilado, você realmente não precisa saber sobre a falsificação, mas se chegou até aqui, provavelmente desejará saber tudo.

A interface genuína que mencionei anteriormente foi `INotifyCompletion`, que se parece com isto:

```
interface pública INotifyCompletion {  
  
    void OnCompleted(Continuação da ação);  
}
```

No entanto, há outra interface que estende isso - ainda no namespace `System.Runtime.CompilerServices`:

```
interface pública ICriticalNotifyCompletion: INotifyCompletion {  
  
    void UnsafeOnCompleted(Continuação da ação);  
}
```

Todo o raciocínio por trás dessas duas interfaces tem o contexto em sua essência. Já mencionei `SynchronizationContext` diversas vezes neste capítulo, e você pode muito bem já ter se separado com isso antes; é um contexto de sincronização que permite que as chamadas sejam empacotadas em um thread apropriado, seja um pool de threads específico ou um único thread de UI , ou o que for necessário. Porém, não é o único contexto envolvido. Existem muitos deles – `SecurityContext`, `LogicalCallContext` e `HostExecution-Context`, por exemplo. O avô de todos eles, entretanto, é `ExecutionContext`. Ele atua como um contêiner para todos os outros contextos e é nisso que nos concentraremos nesta seção.

É muito importante que o `ExecutionContext` flua pelos pontos de espera; você não quer voltar ao seu método assíncrono quando uma tarefa for concluída, apenas para descobrir que esqueceu qual usuário está representando, por exemplo. Para que o contexto flua, ele precisa ser capturado quando você anexa a continuação e, em seguida, restaurado quando a continuação for executada. Isto é conseguido através dos métodos `ExecutionContext.Capture` e `ExecutionContext.Run` , respectivamente.

Existem dois trechos de código que podem executar esse par de captura/restauração: o aguardador e a classe `AsyncTaskMethodBuilder<T>` (junto com seus irmãos). Você pode esperar que possa simplesmente decidir de uma forma ou de outra e deixar por isso mesmo. Mas várias outras compensações entram em jogo. É fácil esquecer de fluir o contexto de execução no aguardador, então faz sentido implementá-lo *uma vez* no código do construtor de método. Por outro lado, seu waiter estará diretamente acessível a qualquer código que o utilize, então você não gostaria de expor uma possível falha de segurança confiando em todos os chamadores usando o código gerado pelo compilador...sugerindo que deveria estar no código do waiter . Mas da mesma forma, você não gostaria de capturar e restaurar o contexto duas vezes, de forma redundante. Como podemos resolver esta dicotomia?

Já vimos a resposta: use duas interfaces diferentes com significados sutilmente diferentes. Se você implementar o padrão awaitable, seu método OnCompleted (que é obrigatório) deverá fluir no contexto de execução. Se você optar por implementar ICriticalNotifyCompletion, seu método UnsafeOnCompleted não deverá fluir no contexto de execução... e deverá ser decorado com o atributo [SecurityCritical] para evitar que código não confiável o chame. Os construtores de métodos são confiáveis, é claro, e fluem o contexto, então está tudo bem – chamadores parcialmente confiáveis ainda podem usar seu aguardador com eficiência, mas possíveis invasores não serão capazes de evitar o fluxo de contexto.

Eu deliberadamente mantive esta seção bastante breve; Acho todo o tópico dos contextos um tanto confuso e há ainda mais complexidades que não mencionei. Se você estiver implementando seu próprio waiter sem delegar a um existente (e provavelmente não precisará), você definitivamente deveria ler a postagem do blog “ExecutionContext vs SynchronizationContext” de Stephen Toub (<http://mng.bz/Ye65>) para mais detalhes.

#### 15.6.5 Operações assíncronas no WinRT

O Windows 8 introduziu a Windows Store no ecossistema de aplicativos — e junto com ela, o WinRT. Entro em mais detalhes sobre o WinRT no apêndice C, mas ele foi projetado para ser um ambiente moderno, orientado a objetos e não gerenciado. Em muitos aspectos, é o novo Win32. Alguns dos tipos familiares do .NET não estão disponíveis no WinRT, e mesmo aqueles que estão disponíveis foram, em sua maioria, desprovidos de bloqueio de chamadas relacionadas ao IO.

Os tipos que ainda residem no CLR geralmente expõem operações assíncronas por meio de Task<T> como você já viu, mas esse tipo não existe no próprio WinRT.

Em vez disso, há várias interfaces, todas estendendo uma interface IAsyncInfo principal:

- ÿ IAsyncAction ÿ
- IAsyncResultWithProgress<TProgress> ÿ
- IAsyncOperation<TResult> ÿ
- IAsyncOperationWithProgress<TResult, TProgress>

Você pode pensar na diferença entre os tipos Action e os tipos Operation como sendo semelhante à diferença entre Task e Task<T>, ou entre Action e Func: uma Action não tem valor de retorno, enquanto uma Operation tem. As versões WithProgress criam relatórios de progresso em um único tipo, em vez de exigir sobrecargas de método com IProgress<T> conforme o TAP.

Os detalhes dessas interfaces estão além do escopo deste livro, mas há muitos recursos disponíveis para explicá-las. Sugiro que você comece com a postagem do blog do Windows 8 de Stephen Toub “Aprofundando-se no WinRT e aguardando” (<http://mng.bz/F1TF>).

Em termos de manipulação dessas interfaces do C# 5, existem alguns pontos importantes:

- ÿ Os métodos de extensão GetAwaiter permitem aguardar ações e operações diretamente.
- ÿ Os métodos de extensão AsTask permitem visualizar uma ação ou operação como uma tarefa, com suporte para tokens de cancelamento e relatórios de progresso via IProgress<T>.
- ÿ Os métodos de extensão AsAsyncOperation e AsAsyncAction vão na direção oposta, executando uma tarefa e retornando um wrapper compatível com WinRT.

Tudo isso é fornecido pela classe System.WindowsRuntimeSystemExtensions , no assembly System.Runtime.WindowsRuntime.dll .

Mais uma vez você viu o valor do padrão aguardável. O compilador C# realmente não se importa em chamar um método de extensão para aguardar a operação assíncrona. É apenas outro tipo aguardável. Na maioria das vezes, você provavelmente conseguirá deixar uma operação assíncrona em seu tipo nativo e aguardá-la normalmente. É bom ter a flexibilidade de tratar uma operação assíncrona do WinRT como um Task<T> mais familiar para cenários mais complexos.

Outra opção para executar código no modelo WinRT de assincronia é usar o método Run na classe System.Runtime.InteropServices.WindowsRuntime.AsyncInfo . Usar isso é mais limpo do que chamar Task.Run(...).AsAsyncOperation se você precisar entregar um IAsyncOperation (ou IAsyncAction) para algum outro código.

A assincronia realmente não é opcional ao escrever aplicativos WinRT. Muitas vezes, a plataforma não oferece a opção de escrever código síncrono para IO. É claro que você *pode* fazer todo o trabalho sozinho, mas usar os recursos do C# 5 torna o WinRT significativamente mais simples de usar. Tenho certeza de que não é coincidência que a linguagem tenha ganhado assincronia aproximadamente na mesma época em que o WinRT foi lançado. A Microsoft não está apenas mergulhando na água aqui; é assim que você *escreverá* aplicativos da Windows Store em C#.

## 15.7 Resumo

Espero que as seções mais complicadas e aprofundadas deste capítulo não tenham obscurecido a elegância dos recursos assíncronos do C# 5. A capacidade de escrever código assíncrono eficiente em um modelo de execução mais familiar é um grande avanço e Acredito que será transformador – uma vez que seja bem compreendido. Na minha experiência ao fazer apresentações sobre assíncrono, muitos desenvolvedores ficam facilmente confusos com o recurso na primeira vez que o veem e usam. Isso é totalmente comprensível, mas, por favor, não deixe que isso o desencoraje. Esperamos que este capítulo ajude a responder pelo menos algumas de suas perguntas à medida que você avança, mas há uma grande quantidade de documentação por aí e muitas pessoas prontas para ajudar no Stack Overflow, é claro.

Falando de outros recursos, devo enfatizar que tentei principalmente cobrir aqui os aspectos *linguísticos* da assincronia, de acordo com o resto do livro. Há muito mais no desenvolvimento assíncrono do que apenas conhecer os recursos da linguagem,

porém, e recomendo que você leia tudo o que puder sobre o TPL. Mesmo que você não possa usar Ainda em C# 5, se você estiver usando o .NET 4, poderá começar a usar `Task<T>` como um modelo limpo para operações assíncronas. Sempre que você estiver tentado a usar um método Thread bruto , pense se o TPL pode fornecer uma abstração mais alta para permitir que você alcance o mesmo objetivo de forma mais simples.

Resumindo: funções assíncronas em C# 5 *rock*. Isso não é tudo o que há para olhar no entanto. Há alguns pequenos recursos que eu realmente deveria abordar antes de encerrar esta edição...

# 16

Recursos bônus do C# 5  
e pensamentos finais

Este capítulo cobre

- ÿ Mudanças nas variáveis capturadas
- ÿ Atributos de informações do chamador
- ÿ Considerações finais

O C# 2 tinha vários recursos pequenos, mas díspares, junto com os principais. C#3 tinha vários recursos menores construídos no LINQ. Mesmo o C# 4 tinha recursos relativamente pequenos sobre os quais vale a pena entrar em detalhes.

O C# 5 quase não possui recursos além da assincronia. Tem apenas dois pequenos extras, ambos pequeno. A equipe de design C# sempre avalia o custo de um recurso (em termos de design, implementação, testes, documentação e educação do desenvolvedor) contra seus benefícios. Tenho certeza de que há muitas solicitações de recursos pendentes que a equipe gostaria para satisfazer, então presumivelmente os custos desses recursos pequenos eram pequenos o suficiente para permitir que eles façam o corte.

A primeira mudança não é tanto um recurso, mas uma correção de um erro anterior no o design da linguagem...

16.1 Mudanças nas variáveis capturadas em loops foreach Na seção 5.5.5, dei um aviso sobre o código que usava uma função anônima (normalmente uma expressão lambda) dentro de um loop foreach, capturando a variável do loop.

A listagem a seguir mostra um exemplo simples de tal código, que parece gerar x, depois y e depois z.

#### Listagem 16.1 Usando variáveis de iteração capturadas

```
string[] valores = { "x", "y", "z" }; var ações = new  
Lista<Ação>();  
  
foreach (valor da string em valores) {  
  
    ações.Add(() => Console.WriteLine(valor));  
}  
  
foreach (ação de ação em ações) {  
  
    Ação();  
}
```

Em C# 3 e C# 4, isso na verdade imprimiria z três vezes — a variável de loop (valor) seria capturada pela expressão lambda, e teoricamente havia apenas uma “instância” de variável que mudava de valor em cada iteração do loop. Todos os três delegados se refeririam à mesma variável e, no momento em que fossem executados no final, o valor dessa variável seria z. Isto não foi um erro de implementação no compilador; foi como a linguagem foi especificada para se comportar.

No C# 5, a linguagem funciona como você provavelmente esperava: cada iteração do loop introduz efetivamente uma variável separada. Cada um dos delegados se referirá a uma variável diferente, com o valor daquela iteração do loop.

Não há muito mais a dizer sobre esse recurso - ele está apenas corrigindo uma área da linguagem que causou problemas para muitos desenvolvedores. (Você provavelmente ficaria surpreso com quantas perguntas sobre Stack Overflow isso causou.)

Quero deixar um aviso: se você estiver na posição bastante incomum de escrever código que precisa ser compilado com várias versões diferentes do compilador C#, você precisa estar ciente de que o comportamento irá variar. O código da listagem 16.1 não produz nenhum aviso em nenhuma versão do C# — o comportamento apenas muda silenciosamente para o C# 5. Tenha cuidado e certifique-se de ter testes de unidade para recorrer!

Vamos para o recurso final...

#### 16.2 Atributos de informações do chamador

Alguns recursos são muito gerais – expressões lambda, variáveis locais digitadas implicitamente, genéricas e similares. Outros são mais específicos - o LINQ realmente se destina a consultar dados de uma forma ou de outra, embora tenha como objetivo generalizar sobre muitas fontes de dados diferentes. O recurso final do C# 5 é extremamente direcionado: há dois casos de uso significativos (um óbvio e outro um pouco menos), e eu realmente não espero que ele seja muito usado fora dessas situações.

### 16.2.1 Comportamento básico

O .NET 4.5 introduz três novos atributos: `CallerFilePathAttribute`, `CallerLineNumberAttribute` e `CallerMemberNameAttribute`, todos no namespace `System.Runtime.CompilerServices`. Assim como acontece com outros atributos, ao aplicar qualquer um deles, você pode omitir o sufixo `Attribute` e, como essa é a maneira mais comum de usar atributos, abreviarei os nomes apropriadamente no restante do livro.

Todos os três atributos só podem ser aplicados a parâmetros e só são úteis quando aplicados a parâmetros *opcionais*. A ideia é simples: se o site de chamada não fornecer o argumento, o compilador usará o arquivo atual, o número da linha ou o nome do membro para preencher o argumento, em vez de usar o valor padrão normal. Se o chamador *fornecer* um argumento, o compilador o deixará em paz.

A listagem a seguir mostra um exemplo de ambos os casos.

#### Listagem 16.2 Usando atributos de informações do chamador corretamente e abusando deles

```
static void ShowInfo([CallerFilePath] string arquivo = null,
                     [CallerLineNumber] linha interna = 0,
                     [CallerMemberName] membro da string = null)
{
    Console.WriteLine("{0}:{1} - {2}", arquivo, linha, membro);
}
...
ShowInfo();
ShowInfo("LiesAndDamnedLies.java", -10);
```

O compilador preenche tudo

O compilador preenche apenas o nome

A saída da listagem 16.2 seria algo assim:

```
c:\Users\Jon\Code\Chapter16\CallerInfoDemo.cs:21 - Principal LiesAndDamnedLies.java:-10 -
Principal
```

Claro, você normalmente não forneceria um valor falso para nenhum desses argumentos, mas é útil poder passar o valor explicitamente, principalmente se você quiser registrar o chamador do método atual, usando os mesmos atributos.

O nome do membro funciona para todos os membros, normalmente da forma óvia, com os seguintes nomes especiais razoavelmente previsíveis:

- ÿ Construtor estático: `.cctor`
- ÿ Construtor: `.ctor`
- ÿ Finalizador: `Finalizar`

O nome usado como parte de uma chamada de método durante um inicializador de campo é o nome do campo.

Há duas situações em que as informações do membro chamador *não são* preenchidas. A primeira é a inicialização do atributo; A Listagem 16.3 fornece um exemplo de um atributo ao qual você poderia esperar que recebesse o nome do membro ao qual foi aplicado, mas infelizmente o compilador não preenche nada automaticamente neste caso.

**Listagem 16.3 Tentando usar atributos de informações do chamador em uma declaração de atributos**

```
[AttributeUsage(AttributeTargets.All)] classe pública
MemberDescriptionAttribute: Atributo {

    public MemberDescriptionAttribute([CallerMemberName] string membro = nulo) {

        Membro = membro;

    }

    string pública Membro {obter; definir; }
}
```

Isso definitivamente poderia ser útil. Já vi situações em que os desenvolvedores encontraram atributos por meio de reflexão, mas tiveram que preencher sua própria estrutura de dados para manter um ping de mapeamento entre o nome do membro e o atributo, o que poderia ser feito automaticamente pelo compilador.

A omissão de digitação dinâmica é mais facilmente perdoável. A listagem a seguir demonstra o tipo de uso que infelizmente não funciona.

**Listagem 16.4 Tentando usar atributos de informações do chamador com chamada dinâmica**

```
class TypeUsedDynamically {

    interno void ShowCaller([CallerMemberName] string caller = "Desconhecido") {

        Console.WriteLine("Chamado por: {0}", chamador);
    }
}

dinâmico x = novo TypeUsedDynamically(); x.ShowCaller();
```

A Listagem 16.3 apenas imprime Called by: Unknown como se o atributo não estivesse presente.

Embora isso possa parecer decepcionante, considere a alternativa: para funcionar, o compilador precisaria incorporar o nome do membro, o nome do arquivo e o número da linha em cada chamada dinâmica que poderia eventualmente exigir a informação. No geral, acho que os custos superariam os benefícios para a maioria dos desenvolvedores.

**16.2.2 Registro**

O caso mais óbvio em que as informações do chamador são úteis é ao gravar em um arquivo de log.

Anteriormente, ao registrar em log, você normalmente construiria um rastreamento de pilha (usando `System.Diagnostics.StackTrace`, por exemplo) para descobrir de onde vinham as informações de log. Isso normalmente fica oculto nas estruturas de registro, mas ainda está lá — e é feio. É potencialmente um problema em termos de desempenho e é frágil diante do alinhamento do compilador JIT.

É fácil ver como uma estrutura de registro poderia fazer uso do novo recurso para permitir que informações somente do chamador fossem registradas de maneira muito barata, preservando até mesmo os números das linhas.

e nomes de membros diante de uma compilação que teve informações de depuração removidas e mesmo após ofuscação. Isso não ajuda nos casos em que você deseja registrar um rastreamento completo da pilha, é claro, mas também não diminui sua capacidade de fazer isso.

Enquanto escrevo, não tenho conhecimento de nenhuma estrutura de registro que tenha aproveitado isso; seria necessário um build direcionado especificamente ao .NET 4.5, para começar, ou uma estrutura com os atributos declarados explicitamente, como você verá na seção 16.2.4. Mas deve ser fácil escrever suas próprias classes wrapper que utilizem qualquer estrutura de registro de sua preferência e forneçam informações do chamador. Com o tempo, tenho certeza de que os frameworks irão se atualizar e fornecer essa funcionalidade imediatamente.

### 16.2.3 Implementando INotifyPropertyChanged

O uso menos óbvio de apenas um desses atributos, [CallerMemberName], pode ser *muito* óbvio para você se você implementar INotifyPropertyChanged com frequência.

A interface é muito simples – é um evento único do tipo PropertyChangedEventHandler. Este é um tipo delegado com a seguinte assinatura:

delegado público void PropertyChangedEventHandler (remetente do objeto,

PropertyChangedEventArgs e)

PropertyChangedEventArgs, por sua vez, possui um único construtor:

```
public PropertyChangedEventArgs(string nomeadapropriedade)
```

Uma implementação típica de INotifyPropertyChanged antes do C# 5 pode ser parecida com a seguinte.

#### Listagem 16.5 Implementando INotifyPropertyChanged da maneira antiga

```
classe OldPropertyNotifier: INotifyPropertyChanged {

    evento público PropertyChangedEventHandler PropertyChanged;

    private int primeiroValor; público int
    PrimeiroValor
    {
        get { return primeiroValor; } definir

        {
            if (valor! = primeiroValor) {

                primeiroValor = valor;
                NotifyPropertyChanged("PrimeiroValor");
            }
        }
    }

    // Outras propriedades com o mesmo padrão

    private void NotifyPropertyChanged(string nomeadapropriedade) {

        Manipulador PropertyChangedEventHandler = PropertyChanged; if (manipulador! = nulo)
```

```
        {
            manipulador(este, novo PropertyChangedEventArgs(propertyName));
        }
    }
}
```

O objetivo do método auxiliar é evitar a necessidade de colocar a verificação de nulidade em cada propriedade. Você poderia facilmente torná-lo um método de extensão para evitar repeti-lo em cada implementação, é claro.

Isso não é apenas prolixo (que não mudou) – é frágil. O problema é que o nome da propriedade (FirstValue) é especificado como uma string literal e, se você refatorar o nome da propriedade para outra coisa, poderá facilmente esquecer de alterar a string literal. Se você tiver sorte, suas ferramentas e testes ajudarão você a identificar o erro, mas ele ainda é muito feio.

Com o C# 5, a maior parte do código permanece a mesma, mas você pode fazer o compilador preencher o nome da propriedade usando `CallerMemberName` no método auxiliar, como segue.

#### Listagem 16.6 Implementando INotifyPropertyChanged usando informações do chamador

```
// Dentro do setter if (value !=  
firstValue)  
{  
    primeiroValor = valor;  
    NotifyPropertyChanged();  
}  
  
...  
  
void NotifyPropertyChanged([CallerMemberName] string propertyName = null)  
{  
    //Exatamente o mesmo código de antes  
}
```

Mostrei apenas as seções do código que foram alteradas — é simples assim. Agora, quando você alterar o nome da propriedade, o compilador usará o novo nome. Não é uma melhoria surpreendente, mas é ainda melhor.

#### 16.2.4 Usando atributos de informações do chamador sem .NET 4.5

Assim como os métodos de extensão, os atributos de informações do chamador apenas permitem que você peça ao compilador para mexer levemente em seu código durante o processo de compilação. Eles não usam nenhuma informação que você não possa fornecer - você só precisa ter cuidado ao fazer isso. Assim como os métodos de extensão, é possível usá-los ao direcionar uma versão anterior do .NET àquela que *realmente* contém os atributos — você só precisa declarar os atributos você mesmo. Isto é tão simples quanto copiar a declaração do MSDN. Os atributos em si não possuem parâmetros, então você só precisa fornecer um corpo vazio para a declaração da classe, que ainda deve estar no namespace `System.Runtime.CompilerServices`.

O compilador C# tratará os atributos fornecidos pelo usuário exatamente da mesma maneira que trataria os atributos reais no .NET 4.5. A desvantagem dessa abordagem é que você

você terá problemas se você criar o mesmo código no .NET 4.5. Você precisará remover seus atributos artesanais nesse ponto, para evitar confundir o compilador.

Se você estiver usando .NET 4, Silverlight 4 ou 5 ou Windows Phone 7.5, outra opção é usar o pacote Microsoft.Bcl NuGet. Isso fornece esses atributos junto com vários outros tipos úteis que você poderia desejar.

E é isso: C# 5 pronto.

### 16.3 Considerações finais As duas

primeiras edições do C# in Depth foram encerradas com um capítulo dedicado ao futuro como eu o percebi no momento em que escrevi este artigo. Se você possui uma (ou ambas!) dessas edições, talvez queira olhar para trás e dar uma risadinha consigo mesmo. Não creio que tenha dito nada escandalosamente errado, mas claramente não tinha ideia do quanto as coisas poderiam mudar em apenas alguns anos.

Também gostaria de salientar que não tinha ideia do que viria no C# 4 ou no C# 5 até que eles fossem anunciados pela Microsoft. Tanto a digitação dinâmica quanto as funções assíncronas foram grandes surpresas para mim. Tive a sorte de apresentar minhas ideias para C# 5 em uma conferência, com a presença de alguns membros da equipe C#, e estou extremamente satisfeito por elas terem seguido seu próprio caminho. Caso eu ainda não tenha sido claro, async/await rocks é um recurso, e está muito além de qualquer coisa que eu pudesse ter inventado.

O que está reservado para a indústria? Mais mobilidade, mais entrada por toque, serviços em nuvem mais distribuídos, possivelmente realidade aumentada – todas essas são apostas razoavelmente seguras agora. Mas se estas forem as forças mais perturbadoras da indústria até ao final de 2014, ficarei muito desapontado. As melhores coisas na computação parecem surgir do nada – depois de muitos anos de árduo esforço das pessoas envolvidas, é claro – e surpreendem a todos.

O mesmo tipo de coisa pode ser dito sobre C#. Ainda tenho minha lista de desejos de recursos secundários, e talvez o C# 6 seja uma versão organizada, com muitos recursos secundários em vez dos enormes que vimos no passado. Talvez a linguagem seja expandida de forma extensível, permitindo que outros desenvolvedores criem eles próprios esses recursos menores. Ou talvez o novo recurso matador seja algo que eu nem sabia que precisava – mais uma vez.

As equipes de C# e .NET certamente não ficaram ociosas. Mesmo deixando de lado o C# 5 e todo o trabalho necessário para integrar o .NET à interface do Windows 8, conhecemos um projeto no qual eles estão trabalhando arduamente: Roslyn. Nomeado como um trocadilho com a orientação do escritório de Eric Lippert quando ele trabalhou no projeto, Roslyn é outro nome para a ideia de “compilador como serviço” que vem sendo comentada há tanto tempo. Roslyn fornecerá uma API que os desenvolvedores podem usar para analisar código C# (ou VB), modificá-lo programaticamente, compilá-lo em IL e assim por diante. Suspeito que relativamente poucos desenvolvedores terão necessidade disso, mas aqueles que o fizerem ficarão imensamente felizes com isso e criará coisas maravilhosas para o resto de nós. Imagine ser capaz de escrever suas próprias ferramentas de refatoração, análise de convenção de código mais sofisticada, geração de código e muito mais – tudo com uma API projetada para ser poderosa e com desempenho suficiente para ser o mecanismo para versões futuras do Visual.

Estúdio. Talvez mais importante para a maioria de nós, Roslyn oferece à equipe C# um campo de jogo no qual é relativamente fácil implementar novos recursos. Talvez eles se tornem ainda mais aventureiros e ambiciosos no futuro!

Posso afirmar uma coisa com bastante certeza: continuarei gostando de escrever, falar e usar C# por algum tempo, independentemente de a linguagem evoluir ou não. Acho difícil acreditar que a programação se tornará *menos* interessante na próxima década.

Como nas edições anteriores, recomendo que você faça coisas incríveis. Escreva um código fabulosamente claro com o qual seus colegas adorarão trabalhar. Desenvolva a próxima grande novidade no mundo do código aberto. Ajude outros desenvolvedores no Stack Overflow. converse com grupos de usuários, conferências, amigos e qualquer pessoa que queira ouvir sobre qualquer que seja sua paixão. Desejo-lhe muita sorte em qualquer uma dessas coisas que você empreenda, e espero que este livro tenha fornecido alguma pequena ajuda para alcançar suas ambições.

## apêndice A

### Operadores de consulta padrão LINQ

---

Existem muitos operadores de consulta padrão no LINQ, apenas alguns deles são suportados diretamente em expressões de consulta C# — os outros precisam ser chamados manualmente como métodos normais. Alguns dos operadores de consulta padrão são demonstrados no texto principal do livro, mas estão todos listados neste apêndice.

A maioria dos exemplos usa as duas sequências de amostra a seguir:

```
string[] palavras = {"zero", "um", "dois", "três", "quatro"}; int[] números = {0, 1, 2, 3, 4};
```

Para completar, incluí os operadores que já vimos, embora na maioria dos casos o capítulo 11 contenha mais detalhes sobre eles do que os fornecidos aqui.

O comportamento especificado aqui é o do LINQ to Objects; outros provedores podem funcionar de maneira diferente. Para cada operador, especifiquei se ele usa execução diferida ou imediata. Se um operador usar execução adiada, também indiquei se ele transmite ou armazena em buffer seus dados.

Há algum tempo, reimplementei o LINQ to Objects do zero em um projeto chamado Edulinq, blogando detalhes sobre cada operador e considerando possibilidades de otimização, avaliação preguiçosa e assim por diante. Para obter mais detalhes do que você provavelmente gostaria de saber sobre o LINQ to Objects, visite a página inicial do projeto Edulinq em <http://edulinq.googlecode.com>.

#### A.1 Agregação

Todos os operadores de agregação (ver tabela A.1) resultam num único valor em vez de uma sequência. Média e Soma operam em uma sequência de números (qualquer um dos tipos numéricos integrados) ou em uma sequência de elementos com um delegado para converter de cada elemento em um dos tipos numéricos integrados. Min e Max têm sobrecargas para tipos numéricos, mas também podem operar em qualquer sequência usando o comparador padrão para o tipo de elemento ou usando um delegado de conversão. Count e Long Count são equivalentes entre si, apenas com tipos de retorno diferentes. Ambos têm duas sobrecargas — uma que apenas conta o comprimento da sequência e outra que recebe um predicado, e apenas os elementos que correspondem ao predicado são

Tabela A.1 Exemplos de operadores de agregação

Expressão	Resultado
números.Soma()	10
números.Contagem()	5
números.Média()	2
números.LongCount(x => x % 2 == 0)	3 (como um longo; existem três números pares)
palavras.Min(palavra => palavra.Comprimento)	3 ("um" e "dois")
palavras.Max(palavra => palavra.Comprimento)	5 ("três")
números.Aggregate("semente", (atual, item) => atual + item, resultado=> resultado.ToUpper())	"SEED01234"

O operador de agregação mais generalizado (mostrado na linha inferior da tabela A.1) é apenas chamado de Agregado. Todos os outros operadores de agregação poderiam ser expressos como chamadas para Aggregate, embora seja relativamente doloroso fazer isso. A ideia básica é que há sempre um “resultado até agora”, começando com uma semente inicial. Um delegado de agregação é aplicado para cada elemento da sequência de entrada; o delegado leva o resultado até agora e o elemento de entrada e produz o próximo resultado. Como etapa opcional final, uma conversão é aplicada do resultado da agregação ao valor de retorno do método. Esta conversão pode resultar em um tipo diferente, se necessário. Não é tão complicado quanto soa, mas ainda assim é improvável que você o use com frequência.

Todos os operadores de agregação utilizam execução imediata. A sobrecarga para Count que não usa um predicado é otimizada para implementações de ICollection e ICollection<T>; nessa situação, usará a propriedade Count da coleção sem ler nenhum dado.<sup>1</sup>

## A.2 Concatenação

Existe um único operador de concatenação: Concat (ver tabela A.2). Como você pode esperar, isso opera em duas sequências e retorna uma única sequência que consiste em todos os elementos da primeira sequência seguida por todos os elementos da segunda. As duas entradas as sequências devem ser do mesmo tipo, a execução é adiada e todos os dados são transmitidos.

Tabela A.2 Exemplo de Concat

Expressão	Resultado
números.Concat(novo[] {2, 3, 4, 5, 6})	0, 1, 2, 3, 4, 2, 3, 4, 5, 6

<sup>1</sup> Não existe esse atalho para LongCount. Pessoalmente, nunca vi esse método usado no LINQ to Objects.

## A.3 Conversão

Os operadores de conversão cobrem uma ampla gama de utilizações, mas todos vêm em pares.

Os exemplos na tabela A.3 usam duas sequências adicionais para demonstrar Cast e

Do tipo:

```
object[] allStrings = {"Estes", "são", "todos", "strings"};
object[] notAllStrings = {"Número", "at", "the", "end", 5};
```

**Tabela A.3 Exemplos de conversão**

Expressão	Resultado
allStrings.Cast<string>()	"Estes", "são", "todos", "strings" (como IEnumerable<string>)
allStrings.OfType<string>()	"Estes", "são", "todos", "strings" (como IEnumerable<string>)
notAllStrings.Cast<string>()	A exceção é lançada durante a iteração, no ponto de falha na conversão
notAllStrings.OfType<string>()	"Número", "em", "o", "fim" (como IEnumerable<string>)
números.ToArray()	0, 1, 2, 3, 4 (como int[])
números.ToList()	0, 1, 2, 3, 4 (como Lista<int>)
palavras.ToDictionary(w => w.Substring(0, 2))	Conteúdo do dicionário: "ze": "zero" "Num" "tw": "dois" "th": "três" "fo": "quatro"
//A chave é o primeiro caractere da palavra palavras.ToLookup(palavra => palavra[0])	Conteúdo da pesquisa: 'z': "zero" 'o': "um" 't': "dois", "três" 'f': "quatro"
palavras.ToDictionary(palavra => palavra[0])	Exceção: só pode ter uma entrada por chave, portanto falha em 't'

ToDictionary e ToList são bastante autoexplicativos: eles lêem toda a sequência em memória, retornando-a como um array ou como List<T>. Ambos usam execução imediata.

Cast e OfType convertem uma sequência não digitada em uma digitada, lançando um exceção (para Cast) ou ignorando (para OfType) elementos da sequência de entrada que não são implicitamente conversíveis para o tipo de elemento de sequência de saída usando um unboxing ou conversão de referência. Isso também pode ser usado para converter sequências digitadas em mais sequências digitadas especificamente, como a conversão de IEnumerable<object> em IEnumerable<string>. Eles usam execução adiada e transmitem seus dados de entrada.

ToDictionary e ToLookup recebem delegados para obter a chave de qualquer elemento específico. ToDictionary retorna um dicionário mapeando a chave para o tipo de elemento, enquanto ToLookup retorna um ILookup<T> digitado apropriadamente. Uma pesquisa é como um dicionário onde o valor associado a uma chave não é um elemento, mas uma sequência de elementos. As pesquisas geralmente são usadas quando chaves duplicadas são esperadas como parte da operação normal, enquanto uma chave duplicada fará com que ToDictionary lance uma exceção. Sobrecargas mais complicadas de ambos os métodos permitem que um IEqualityComparer<T> personalizado seja usado para comparar chaves e um delegado de conversão seja aplicado a cada elemento antes de ser colocado no dicionário ou na pesquisa. Ambos os métodos usam execução imediata.

Existem dois operadores adicionais para os quais não forneci exemplos: AsEnumerable e AsQueryable. Eles não afetam os resultados de uma forma imediatamente óbvia, portanto não podem ser demonstrados aqui. Em vez disso, afetam a maneira como a consulta é executada. Queryable.AsQueryable é um método de extensão em IEnumerable que retorna um IQueryable (ambos os tipos são genéricos ou não genéricos, dependendo da sobrecarga escolhida). Se o IEnumerable que você chama já for um IQueryable, ele retornará a mesma referência; caso contrário, cria um wrapper em torno da sequência original. O wrapper permite que você use todos os métodos normais de extensão Queryable, passando árvores de expressão, mas quando a consulta é executada a árvore de expressão é compilada em IL normal e executada diretamente, usando o método LambdaExpression.Compile mostrado na seção 9.3 .2.

Enumerable.AsEnumerable é um método de extensão em IEnumerable<T> e possui um implementação trivial, simplesmente retornando a referência que foi chamada. Nenhum wrapper está envolvido – apenas retorna a mesma referência. Isso força os métodos de extensão Enumerable a serem usados em operadores LINQ subsequentes. Considere as seguintes expressões de consulta:

```
// Filre os usuários no banco de dados com LIKE do usuário em
context.Users
onde user.Name.StartsWith("Tim") seleciona usuário;
```

```
// Filre os usuários na memória do usuário em
context.Users.AsEnumerable() where user.Name.StartsWith("Tim")
select user;
```

A segunda expressão de consulta força o tipo de tempo de compilação da origem a ser IEnumerable<User> em vez de IQueryable<User>, para que todo o processamento seja feito na memória e não no banco de dados. O compilador usará os métodos de extensão Enumerable (usando parâmetros delegados) em vez dos métodos de extensão Queryable (usando parâmetros da árvore de expressão). Normalmente você deseja fazer o máximo de processamento possível em SQL, mas quando há transformações que exigem código local, às vezes é necessário forçar o LINQ a usar os métodos de extensãoEnumerable apropriados .

É claro que isso não é específico de bancos de dados; o tema de forçar o final de uma consulta para

use Enumerable também é aplicável a outros provedores, se eles forem baseados em IQueryable ou algo semelhante.

## A.4 Operadores de elementos

Esta é outra seleção de operadores de consulta agrupados em pares (ver tabela A.4).

Desta vez, todos os pares funcionam da mesma maneira. Há uma versão simples que escolhe um único elemento se puder ou lança uma exceção se o elemento especificado não existir, e uma versão com OrDefault no final do nome. Todos esses operadores usam imediato execução.

Tabela A.4 Exemplos de seleção de elemento único

Expressão	Resultado
palavras.ElementAt(2)	"dois"
palavras.ElementAtOrDefault(10)	nulo
palavras.Primeiro()	"zero"
palavras.Primeiro(w => w.Comprimento == 3)	"um"
palavras.Primeiro(w => w.Comprimento == 10)	Exceção: nenhum elemento correspondente
palavras.FirstOrDefault (w => w.Comprimento == 10)	nulo
palavras.Ultimo()	"quatro"
palavras.Single()	Exceção: mais de um elemento
palavras.SingleOrDefault()	Exceção: mais de um elemento
palavras.Single(palavra => palavra.Comprimento == 5)	"três"
palavras.Single(palavra => palavra.Comprimento == 10)	Exceção: nenhum elemento correspondente
palavras.SingleOrDefault (w => w.Comprimento == 10)	nulo

Os nomes dos operadores são facilmente compreendidos: Primeiro e Último retornam o primeiro e o último elementos da sequência, respectivamente, lançando uma InvalidOperationException se a sequência está vazia. Single retorna o único elemento em uma sequência, lançando uma exceção se a sequência estiver vazia ou tiver mais de um elemento. ElementAt retorna um elemento específico por índice – o quinto elemento, por exemplo. Uma ArgumentOutOfRangeException será lançada se o índice for negativo ou muito grande para o número real de elementos da coleção. Além disso, há uma sobrecarga para todos os operadores diferente de ElementAt para filtrar a sequência primeiro - por exemplo, First pode retornar o primeiro elemento que corresponde a uma determinada condição.

As versõesOrDefault desses métodos suprimem as exceções que acabei de descrito (retornando o valor padrão para o tipo de elemento), exceto em um

case: SingleOrDefault retornará um valor padrão se a sequência estiver vazia, mas se houver mais de um elemento, ainda lançará uma exceção, assim como Single. Isto é projetado para situações onde se tudo estiver correto, a sequência terá zero ou um elemento. Se você quiser lidar com sequências que podem ter mais elementos, use FirstOrDefault .

Todas as sobrecargas que não possuem um parâmetro predicado são otimizadas para instâncias de `IList<T>`, pois podem acessar o elemento correto sem iteração.

Não há otimização quando um predicado está envolvido – não faria sentido para a maioria das chamadas, embora pudesse fazer uma grande diferença ao encontrar o *último* elemento correspondente em uma lista, retrocedendo a partir do final. No momento em que este artigo foi escrito, esse caso não estava otimizado, mas poderia mudar em uma versão futura.

## A.5 Igualdade

Existe apenas um operador de igualdade padrão: `SequenceEqual` (ver tabela A.5). Isso apenas compara duas sequências quanto à igualdade elemento por elemento, incluindo ordem. Por exemplo, a sequência 0, 1, 2, 3, 4 não é igual a 4, 3, 2, 1, 0. Uma sobrecarga permite que um `IEqualityComparer<T>` específico seja usado ao comparar elementos. O valor de retorno é booleano e é calculado com execução imediata.

Tabela A.5 Exemplos de igualdade de sequências

Expressão	Resultado
<code>palavras.SequenceEqual (novo[]["zero","um","dois","três","quatro"])</code>	Verdadeiro
<code>palavras.SequenceEqual (novo[]["ZERO","UM","DOIS","TRÊS","QUATRO"])</code>	Falso
<code>palavras.SequenceEqual (novo[]["ZERO","UM","DOIS","TRÊS","QUATRO"], StringComparer.OrdinalIgnoreCase)</code>	Verdadeiro

Novamente, o LINQ to Objects perde um truque aqui em termos de otimização: se ambas as sequências tiverem uma maneira eficiente de recuperar suas contagens, faria sentido verificar se elas são iguais antes de comparar os próprios elementos. Do jeito que está, a implementação apenas percorre ambas as sequências até chegar ao fim ou encontrar uma desigualdade.

## Geração A.6

De todos os operadores de geração (ver tabela A.6), apenas um atua sobre uma sequência existente: `DefaultIfEmpty`. Isso retorna a sequência original, se não estiver vazia, ou uma sequência com um único elemento, caso contrário. O elemento normalmente é o valor padrão para o tipo de sequência, mas uma sobrecarga permite especificar qual valor usar.

Existem três outros operadores de geração que são apenas métodos estáticos em `Enumerable`:

- ÿ `Range` gera uma sequência de inteiros, com os parâmetros especificando o primeiro valor e quantos valores gerar.
- ÿ `Repeat` gera uma sequência de qualquer tipo repetindo um único valor especificado por um determinado número de vezes.
- ÿ `Vazio` gera uma sequência vazia de qualquer tipo.

Todos os operadores de geração usam execução diferida e transmitem sua saída – em outras palavras, eles não apenas preenchem previamente uma coleção e a retornam. A exceção é `Vazio`, que retorna um array vazio do tipo correto. Um array vazio é completamente imutável, portanto, o mesmo array pode ser retornado para cada chamada para o mesmo tipo de elemento.

Tabela A.6 Exemplos de geração

Expressão	Resultado
<code>números.DefaultIfEmpty()</code>	0, 1, 2, 3, 4
<code>novo int[0].DefaultIfEmpty()</code>	0 (dentro de um <code>IEnumerable&lt;int&gt;</code> )
<code>novo int[0].DefaultIfEmpty(10)</code>	10 (dentro de um <code>IEnumerable&lt;int&gt;</code> )
<code>Enumerable.Range(15, 2)</code>	15, 16
<code>Enumerable.Repeat(25, 2)</code>	25, 25
<code>Enumerable.Empty&lt;int&gt;()</code>	Um <code>IEnumerable&lt;int&gt;</code> vazio

## A.7 Agrupamento

Existem dois operadores de agrupamento, mas um deles é `ToLookup`, que você já visto na seção A.3 como um operador de conversão. Isso deixa apenas `GroupBy`, que examinamos na seção 11.6.1 na forma da cláusula `group...by` em expressões de consulta. Isto usa execução adiada, mas armazena seus resultados em buffer: quando você começa a iterar sobre a sequência de grupos resultante, toda a entrada é consumida.

O resultado de `GroupBy` é uma sequência de elementos `IGrouping<,>` digitados apropriadamente. Cada elemento possui uma chave e uma sequência de elementos que correspondem a essa chave. Em De muitas maneiras, esta é apenas uma maneira diferente de ver uma pesquisa – em vez de ter acesso aleatório aos grupos por chave, os grupos são enumerados por sua vez. A ordem em qual os grupos são retornados é a ordem em que suas respectivas chaves são descobertas. Dentro de um grupo, a ordem é a mesma da sequência original.

`GroupBy` tem um número assustador de sobrecargas, permitindo especificar não apenas como uma chave é derivada de um elemento (que é sempre obrigatório), mas também, opcionalmente, do seguindo:

- ÿ Como comparar chaves.
- ÿ Uma projeção de um elemento original para o elemento dentro de um grupo.

ÿ Uma projeção que utiliza uma chave e uma sequência de elementos correspondentes.

O resultado geral neste caso é uma sequência de elementos do tipo de resultado da projeção.

A Tabela A.7 contém exemplos da segunda e terceira opções, bem como a forma mais simples. As comparações de chaves personalizadas são um pouco mais demoradas para serem demonstradas, mas funcionam da maneira óbvia.

Tabela A.7 Exemplos GroupBy

Expressão	Resultado
palavras.GroupBy(palavra => palavra.Length)	Chave: 4; Sequência: "zero", "quatro" Chave: 3; Sequência: "um", "dois" Chave: 5; Sequência: "três"
palavras.GroupBy (palavra => word.Length, // Palavra-chave => word.ToUpper() // Elemento do grupo)	Chave: 4; Sequência: "zero", "quatro" Chave: 3; Sequência: "um", "dois" Chave: 5; Sequência: "três"
// Projetar cada par (chave, grupo) para string words.GroupBy  (palavra => palavra.Comprimento, (tecla, g) => tecla + ":" + g.Contagem())	"4: 2", "3: 2", "5: 1"

A opção especificada no último marcador raramente é usada, na minha experiência.

## A.8 Junções

Dois operadores são especificados como operadores de junção, Join e GroupJoin, ambos os quais você viu na seção 11.5 usando as cláusulas join e join...in de expressão de consulta, respectivamente. Cada método utiliza vários parâmetros: duas sequências, um seletor de chave para cada sequência, uma projeção a ser aplicada a cada par de elementos correspondentes e, opcionalmente, uma comparação de chave.

Para Join a projeção pega um elemento de cada sequência e produz um resultado; para GroupJoin, a projeção pega um elemento da sequência esquerda e uma sequência de elementos correspondentes da sequência direita. Ambos usam execução adiada e transmitem a sequência esquerda, mas leem a sequência direita na sua totalidade quando o primeiro resultado é solicitado.

Para os exemplos de junção na tabela A.8, combinaremos uma sequência de nomes (Robin, Ruth, Bob, Emma) com uma sequência de cores (Vermelho, Azul, Bege, Verde) observando o primeiro caractere de ambos os caracteres. nome e a cor, então Robin juntará com Vermelho e Bob juntará com Azul e Bege, por exemplo.

Observe que Emma não corresponde a nenhuma das cores — o nome não aparece nos resultados do primeiro exemplo, mas aparece no segundo, com uma sequência vazia de cores.

**Tabela A.8 Exemplos de junção**

Expressão	Resultado
<pre> nomes.Join // Sequência esquerda (cores, // sequência correta  name =&gt; name[0], // Seletor de tecla esquerda  color=&gt; color[0], // seletor de tecla direita  // Projeção para pares de resultados  (nome, cor) =&gt; nome +                " - " + cor ) </pre>	"Robin - Vermelho", "Rute - Vermelha", "Bob - Azul", "Bob - Bege"
<pre> nomes.GroupJoin (cores,  nome =&gt; nome[0],  cor =&gt; cor[0],  // Projeção para pares chave/sequência  (nome, correspondências) =&gt; nome + ":" +  string.Join("/",corresponde.ToArray()) ) </pre>	"Robin: Vermelho", "Rute: Vermelho", "Bob: Azul/Bege", "Emma:"

## A.9 Particionamento

Os operadores de particionamento *pulam* uma parte inicial da sequência, retornando apenas o resto, ou *pegar* apenas a parte inicial de uma sequência, ignorando o resto. Em cada caso, você pode especificar quantos elementos estão na primeira parte da sequência ou especificar um condição — a primeira parte da sequência continua até que a condição falhe. Depois de condição falha pela primeira vez, ela não é testada novamente — não importa se mais tarde elementos na correspondência de sequência. Todos os operadores de particionamento usam execução diferida e transmitem seus dados.

O particionamento efetivamente divide a sequência em duas partes distintas, seja por posição ou por predicado. Em cada caso, se você concatenar os resultados de Take ou TakeWhile com os resultados do Skip ou SkipWhile correspondente, fornecendo o mesmo argumento para ambas as chamadas, você terminará com a sequência original: cada elemento ocorrerá exatamente uma vez, na ordem original. Isto é demonstrado pelas chamadas na tabela A.9.

**Tabela A.9 Exemplos de particionamento**

Expressão	Resultado
palavras.Pegue(2)	"Zero um"
palavras.Skip(2)	"dois três quatro"
palavras.TakeWhile(palavra => palavra.Length <= 4)	"zero", "um", "dois"
palavras.SkipWhile(palavra => palavra.Length <= 4)	"três quatro"

## A.10 Projeção

Você viu dois operadores de projeção (Select e SelectMany) no capítulo 11. Select é uma projeção um-para-um simples de um elemento de origem para um elemento de resultado. Select-Many é usado quando há múltiplas cláusulas from em uma expressão de consulta; cada elemento da sequência original é usado para gerar uma nova sequência. Ambos os operadores de projeção (ver tabela A.10) utilizam execução diferida.

Tabela A.10 Exemplos de projeção

Expressão	Resultado
palavras.Selecionar(palavra => palavra.Comprimento)	4, 3, 3, 5, 4
palavras.Selecionar ((palavra, índice) => index.ToString() + ":" + palavra)	"0: zero", "1: um", "2: dois", "3: três", "4: quatro"
palavras.SelectMany (palavra => palavra.ToCharArray())	'z', 'e', 'r', 'o', 'o', 'n', 'e', 't', 'w', 'o', 't', 'h', 'r', 'e', 'e', 'f', 'nosso'
palavras.SelectMany ((palavra, índice) => Enumerable.Repeat(palavra, índice))	"um", "dois", "dois", "três", "três", "três", "quatro", "quatro", "quatro", "quatro"

Existem sobrecargas adicionais que você não viu no capítulo 11. Ambos os métodos têm sobrecargas que permitem que o índice dentro da sequência original seja usado na projeção, e SelectMany niveliza todas as sequências geradas em uma única sequência sem incluir o elemento original ou usa uma projeção para gerar um elemento de resultado para cada par de elementos. Múltiplas cláusulas from sempre usam a sobrecarga que leva a uma projeção. (Exemplos disso são prolixos e não incluídos aqui.)

Consulte o capítulo 11 para obter mais detalhes.)

O .NET 4 introduziu um novo operador chamado Zip. Este não é oficialmente um operador de consulta padrão de acordo com o MSDN, mas vale a pena conhecê-lo de qualquer maneira. São necessárias duas sequências e aplica a projeção especificada a cada par: o primeiro elemento de cada sequência, depois o segundo elemento de cada sequência e assim por diante. A sequência resultante termina quando *qualquer* uma das sequências de origem termina. A Tabela A.11 mostra dois exemplos de Zip, usando os nomes e cores da seção A.8. Zip usa execução diferida e transmite seus dados.

Tabela A.11 Exemplos de Zip

Expressão	Resultado
nomes.Zip(cores, (x, y) => x + "-" + y)	"Robin-Vermelho", "Ruth-Azul", "Bob-Bege", "Emma-Verde"

Tabela A.11 Exemplos Zip (continuação)

Expressão	Resultado
// A segunda sequência para mais cedo nomes.Zip(cores.Take(3), (x, y) => x + " - " + y)	"Robin-Vermelho", "Ruth-Azul", "Bob-Bege"

## A.11 Quantificadores

Todos os operadores quantificadores mostrados na tabela A.12 retornam um valor booleano, usando execução imediata:

- ÿ All verifica se todos os elementos da sequência satisfazem o predicado fornecido.
- ÿ Any verifica se algum dos elementos da sequência satisfaz o predicado fornecido ou se existe algum elemento para a sobrecarga sem parâmetros.
- ÿ Contém verifica se a sequência contém um elemento específico, especificando opcionalmente uma comparação a ser usada.

Tabela A.12 Exemplos de quantificadores

Expressão	Resultado
palavras.Todas(palavra => palavra.Comprimento > 3)	falso ("um" e "dois" têm exatamente três letras)
palavras.Todas(palavra => palavra.Comprimento > 2)	verdadeiro
palavras.Qualquer()	verdadeiro (a sequência não está vazia)
palavras.Qualquer(palavra => palavra.Comprimento == 6)	falso (sem palavras de seis letras)
palavras.Qualquer(palavra => palavra.Comprimento == 5)	verdadeiro ("três" satisfaz a condição)
palavras.Contém("QUATRO")	falso
palavras.Contém("QUATRO", StringComparer.OrdinalIgnoreCase)	verdadeiro

Any é um operador particularmente útil que muitas vezes é esquecido. Se você está tentando descobrir se uma sequência contém algum item (ou qualquer item que corresponda a um predicado), é muito melhor usar source.Any(...) do que source.Count(...) > 0. Eles devem fornecer o mesmos resultados, mas Any pode parar assim que encontrar o primeiro item, enquanto Count tem que conte todos os itens, mesmo que você só precise saber se o resultado é diferente de zero.

A sobrecarga para Contém que não especifica uma comparação personalizada é otimizada se a fonte implementa `ICollection<T>` delegando para a implementação da interface. Isso significa que `Enumerable.Contains()` ainda será rápido quando chamado em um `Hash Set<T>`, por exemplo.

## A.12 Filtragem

Os dois operadores de filtragem são OfType e Where. Para detalhes e exemplos do Operador OfType , consulte a seção A.3. O operador Where retorna uma sequência contendo todos os elementos que correspondem ao predicado fornecido. Possui uma sobrecarga para permitir que o predicado leve em conta o índice do elemento. É incomum exigir o índice, e o A cláusula where nas expressões de consulta não usa essa sobrecarga. Onde sempre usa execução adiada e transmite seus dados. A Tabela A.13 demonstra ambas as sobrecargas.

Tabela A.13 Exemplos de filtragem

Expressão	Resultado
palavras.Onde(palavra => palavra.Comprimento > 3)	"zero", "três", "quatro"
palavras.Onde ((palavra, índice) => índice < palavra.Comprimento)	"zero", // índice=0, comprimento=4 "um", // // índice=1, comprimento=3 "dois", índice=2, comprimento=2 "três", // índice=3, comprimento=5 // Não é "quatro", índice=4, comprimento=4

## A.13 Operadores baseados em conjuntos

É natural poder considerar duas sequências como conjuntos de elementos. Todos os quatro operadores baseados em conjuntos têm duas sobrecargas, uma delas usando a comparação de igualdade padrão para o tipo de elemento e aquele em que a comparação é especificada em um parâmetro extra. Todos eles usam execução diferida.

O operador Distinct é o mais simples – ele atua em uma única sequência e apenas retorna uma nova sequência de todos os elementos distintos, descartando duplicatas. Os outros operadores certifique-se também de que eles retornem apenas valores distintos, mas atuem em duas sequências:

- ÿ Intersect retorna elementos que aparecem em ambas as sequências.
- ÿ Union retorna os elementos que estão em qualquer sequência.
- ÿ Except retorna elementos que estão na primeira sequência, mas não na segunda.  
(Elementos que estão na segunda sequência, mas não na primeira, *não* são retornados.)

Os exemplos destes operadores na tabela A.14 utilizam duas novas sequências: abbc ("a", "b", "b", "c") e cd ("c", "d").

Tabela A.14 Exemplos baseados em conjuntos

Expressão	Resultado
abbc.Distinto()	"a", "b", "c"
abbc.Intersect(cd)	"c"
abbc.União(cd)	"a", "b", "c", "d"
abbc.Exceto(cd)	"a", "b"
cd.Exceto(abbc)	"d"

Todos esses operadores usam execução adiada, mas a distinção entre buffer e streaming é um pouco mais complicada. Distinct e Union transmitem suas sequências de entrada, enquanto Intersect e Except leem toda a sequência de entrada direita para começar, mas depois transmitem a sequência de entrada esquerda de maneira semelhante aos operadores de junção. Todos esses operadores mantêm um conjunto de elementos que já retornaram para não retornar duplicatas. Isso significa que mesmo Distinct e Union são inadequados para sequências grandes demais para caber na memória, a menos que você saiba que haverá um conjunto limitado de elementos distintos.

#### A.14 Classificação Você já

viu todos os operadores de classificação antes: OrderBy e OrderByDescendente fornecem uma ordenação primária, enquanto ThenBy e ThenByDescendente fornecem ordenações subsequentes para elementos que não são diferenciados pelo primário. Em cada caso, uma projeção é especificada de um elemento para sua chave de classificação, e uma comparação (entre chaves) também pode ser especificada. Ao contrário de alguns outros algoritmos de classificação na estrutura (como List<T>.Sort), as ordenações do LINQ são estáveis — em outras palavras, se dois elementos forem considerados iguais em termos de sua chave de classificação, eles serão retornados em a ordem em que apareceram na sequência original.

O operador de classificação final é Reverse, que simplesmente inverte a ordem da sequência. Todos os operadores de classificação (ver tabela A.15) usam execução diferida, mas armazenam seus dados em buffer.

Tabela A.15 Exemplos de classificação

Expressão	Resultado
palavras.OrderBy(palavra => palavra)	"quatro", "um", "três", "dois", "zero"
// Ordena as palavras pelo segundo caractere words.OrderBy(word => word[1])	"zero", "três", "um", "quatro", "dois"
// Ordena as palavras por comprimento; // comprimentos iguais retornados no original // ordena	"um", "dois", "zero", "quatro", "três"
palavras.OrderBy(word => word.Length)	
palavras.OrderByDescendente (palavra => palavra.Length)	"três", "zero", "quatro", "um", "dois"
// Ordena as palavras por comprimento e depois // alfabeticamente	"um", "dois", "quatro", "zero", "três"
palavras.OrderBy(word => word.Length) .ThenBy(palavra => palavra)	
// Ordena as palavras por comprimento e, em seguida, // em ordem alfabética de trás para frente words.OrderBy(word => word.Length) .ThenByDecrescente(palavra => palavra)	"dois", "um", "zero", "quatro", "três"
palavras.Reverse()	"quatro", "três", "dois", "um", "zero"

## apêndice B Coleções

### genéricas em .NET

---

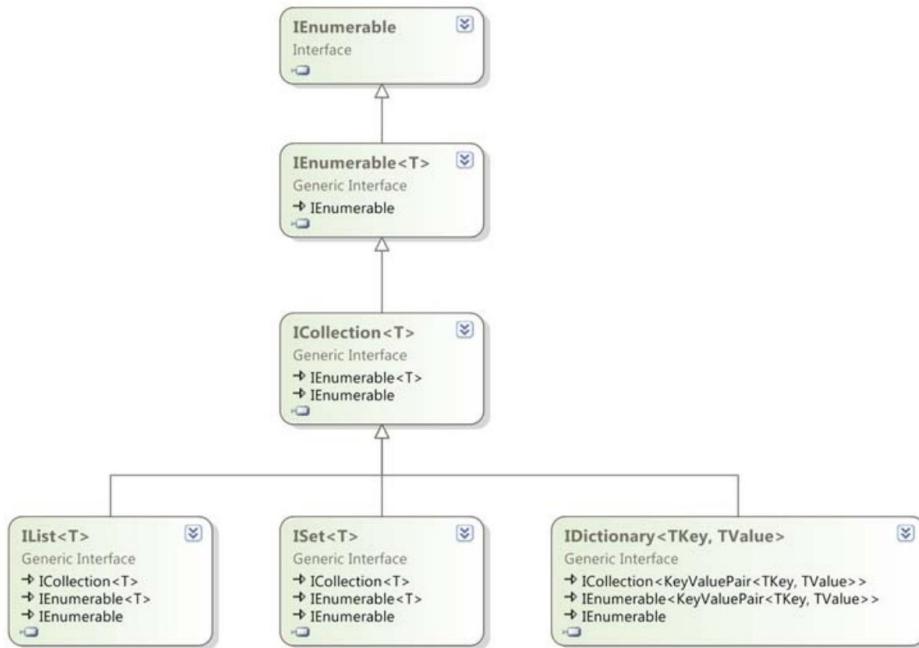
Existem muitas coleções genéricas no .NET e a lista cresceu com o tempo. Este apêndice aborda as interfaces e classes de coleção genéricas mais importantes que você precisa conhecer. Existem coleções não genéricas adicionais em `System.Collections`, `System.Collections.Specialized` e `System.ComponentModel`, mas não vou abordá-las aqui. Da mesma forma, não mencionarei as interfaces LINQ, como `ILookup< TKey, TValue >`. Este apêndice é mais uma referência do que uma orientação – pense nele como uma alternativa para navegar pelo MSDN enquanto você está codificando. Obviamente, o MSDN fornecerá mais detalhes na maioria dos casos, mas o objetivo aqui é permitir que você navegue rapidamente pelas diversas interfaces e implementações disponíveis ao escolher uma coleção específica para usar em seu código.

Não indiquei a segurança do thread de cada coleção, mas o MSDN pode fornecer mais detalhes. Nenhuma das coleções normais suporta vários gravadores simultâneos; alguns suportam um único escritor com leitores simultâneos. A seção B.6 lista as coleções simultâneas que foram adicionadas ao .NET 4. Além disso, a seção B.7 discute as interfaces de coleção somente leitura introduzidas no .NET 4.5.

#### B.1 Interfaces

Quase todas as interfaces que você precisa conhecer estão no namespace `System.Collections.Generic`. A Figura B.1 mostra como as principais interfaces anteriores ao .NET 4.5 estão relacionadas; Inclui também o `IEnumerable` não genérico como raiz da interface. Isso não inclui as interfaces somente leitura no .NET 4.5, pois o diagrama seria muito complicado para ser útil.

Como você já viu diversas vezes, a interface de coleção genérica mais fundamental é `IEnumerable< T >`, representando uma sequência de itens que podem ser iterados. `IEnumerable< T >` permite solicitar um iterador do tipo `IEnumerator< T >`. A separação entre a sequência iterável e o iterador permite que vários iteradores sejam executados independentemente na mesma sequência ao mesmo tempo. Se você quiser pensar em termos de banco de dados, uma tabela é um `IEnumerable< T >`, enquanto um cursor é um `IEnumerator< T >`. Essas são as únicas interfaces de coleção de variantes abordadas neste apêndice, tornando-se `IEnumerable< out T >` e `IEnumerator< out T >` no .NET 4; todos



**Figura B.1** Interfaces em `System.Collections.Generic`, até .NET 4

as outras interfaces envolvem valores do tipo de elemento que entram e saem dos membros, portanto devem ser invariáveis.

Em seguida vem `ICollection<T>` - isso estende `IEnumerable<T>` mas adiciona duas propriedades (`Count` e `IsReadOnly`), métodos de mutação (`Add`, `Remove` e `Clear`), `CopyTo` (que copia o conteúdo para um array) e `Contains` (que determina se a coleção contém um elemento específico). Todas as implementações de coleção genérica padrão implementam esta interface.

`IList<T>` tem tudo a ver com posicionamento: fornece um indexador, `InsertAt` e `RemoveAt` (para corresponder `Adicionar/Remover`, mas com posições) e `IndexOf` (para determinar a posição de um elemento dentro da coleção). Iterar sobre um `IList<T>` geralmente retornará o item no índice 0, depois no índice 1 e assim por diante. Isso não está completamente documentado, mas é uma suposição razoável a ser feita. Da mesma forma, normalmente espera-se que o acesso aleatório a um índice `IList<T>` seja eficiente.

`IDictionary<TKey, TValue>` representa um mapeamento de uma chave exclusiva para um valor para essa chave. Os valores não precisam ser exclusivos e podem ser nulos; as chaves não podem ser nulas. Os dicionários podem ser considerados coleções de pares chave/valor, e é por isso que o `IDictionary<TKey, TValue>` estende `ICollection<KeyValuePair<TKey, TValue>>`. Os valores podem ser recuperado com o indexador ou `TryGetValue`; ao contrário do tipo `IDictionary` não genérico, se você tenta buscar o valor de uma chave ausente, o indexador de `IDictionary<TKey, TValue>` lança uma `KeyNotFoundException`. O objetivo do `TryGetValue` é permitir que você para detectar chaves ausentes em situações onde isso é esperado na operação normal.

`ISet<T>` é uma nova interface no .NET 4, representando um conjunto distinto de valores. Ele foi aplicado retroativamente ao `HashSet<T>` do .NET 3.5, e o .NET 4 introduz uma nova implementação — `SortedSet<T>`.

Geralmente fica bastante claro qual interface (e até mesmo implementação) você deseja usar ao implementar a funcionalidade. Pode ser significativamente mais difícil decidir como expor essa coleção como parte de uma API; quanto mais específico você for no que retorna, mais os chamadores poderão contar com funcionalidades adicionais especificadas por esses tipos. Isto pode facilitar a vida do chamador, em detrimento da flexibilidade futura na sua implementação. Geralmente prefiro usar interfaces para os tipos de métodos e propriedades de retorno, em vez de garantir uma classe de implementação específica. Você também deve pensar cuidadosamente antes de expor uma coleção mutável em uma API, especialmente se essa coleção representar parte do estado do objeto ou tipo. Geralmente é preferível retornar uma cópia ou um wrapper somente leitura em torno da coleção, a menos que o objetivo do método seja permitir a mutação por meio da coleção retornada.

## B.2 Listas

Em muitos aspectos, as listas são o tipo de coleção mais simples e natural. Existem muitas implementações no framework, com diferentes habilidades e características de desempenho.

Alguns grandes rebatedores são usados em todos os lugares, e alguns mais esotéricos são usados para situações especializadas.

### B.2.1 `Lista<T>`

`List<T>` é a escolha padrão para listas na maioria dos casos. Ele implementa `IList<T>` e, portanto, `ICollection<T>`, `IEnumerable<T>` e `IEnumerable`. Além disso, ele implementa as interfaces não genéricas `ICollection` e `IList`, fazendo boxing e unboxing conforme necessário e realizando verificações de tipo em tempo de execução para garantir que novos elementos sejam sempre de um tipo compatível com `T`.

Internamente, `List<T>` armazena um array e controla o tamanho lógico da lista e o tamanho do array de apoio. Adicionar um elemento é um simples caso de definir o próximo valor no array ou (se o array já estiver cheio) copiar o conteúdo existente em um array novo e maior e então definir o valor. Isso significa que a operação tem complexidade de  $O(1)$  ou  $O(n)$  dependendo se os valores precisam ser copiados.

A estratégia de expansão não está documentada — e portanto não é garantida — mas na prática a abordagem sempre foi expandir para duplicar o tamanho recentemente exigido.

Isto resulta em uma complexidade amortizada de  $O(1)$  para anexar um item ao final da lista; às vezes será mais, mas isso se torna cada vez mais raro à medida que a lista aumenta.

Você pode gerenciar explicitamente o tamanho da matriz de apoio obtendo e definindo a propriedade `Capacidade`; o método `TrimExcess` tem o efeito de tornar a capacidade exatamente igual ao tamanho atual. Na prática, isso raramente é necessário, mas se você já sabe o tamanho eventual da lista ao criá-la, pode passar uma capacidade inicial para o construtor, evitando cópias desnecessárias.

A remoção de um elemento de `List<T>` requer que todos os elementos posteriores sejam copiados, portanto sua complexidade é  $O(n - k)$  onde  $k$  é o índice do elemento que você está removendo; aparar a cauda de uma lista é mais barato do que remover a cabeça. Por outro lado, se você estiver tentando remover um elemento por valor em vez de por índice (`Remove` em vez de `RemoveAt`), você efetivamente terminará com uma operação  $O(n)$  onde quer que o elemento esteja: cada elemento deve ser verificado quanto à igualdade ou embaralhado.

Vários métodos em `List<T>` atuam como uma espécie de precursor do LINQ. `ConvertAll` projeta uma lista em outra; `FindAll` filtra a lista original em uma nova lista contendo apenas os valores que correspondem ao predicado especificado. `Sort` executa uma classificação usando o comparador de igualdade padrão para o tipo ou um especificado como argumento. Porém, há uma grande diferença entre `Sort` e `OrderBy` do LINQ: `Sort` modifica o conteúdo da lista original, em vez de produzir uma cópia ordenada. Além disso, `Sort` é instável, enquanto `OrderBy` é estável; elementos iguais na lista original podem ser reordenados ao usar `Classificar`. Um aspecto de `List<T>` que não é suportado pelo LINQ é a pesquisa binária: se você tiver uma lista que já está classificada da maneira correta para o valor que você está procurando, o método `BinarySearch` é mais eficiente do que usar o `IndexOf` linear pesquisa.<sup>1</sup>

Um aspecto um tanto controverso de `List<T>` é o método `ForEach`. Isso faz exatamente o que parece: itera sobre a lista e executa um delegado (especificado como um argumento para o método) para cada valor. Muitos desenvolvedores solicitaram que isso fosse adicionado como um método de extensão para `IEnumerable<T>`, mas essa sugestão foi resistida até agora; Eric Lippert defende que isso é filosoficamente preocupante em seu blog (veja <http://mng.bz/Rur2>). Chamar `ForEach` usando uma expressão lambda parece um exagero para mim; por outro lado, se você já possui um delegado que deseja executar em cada elemento da lista, é melhor fazer com que o `ForEach` faça isso por você, pois ele já está lá.

## B.2.2 Matrizes

Matrizes são, em certo sentido, o nível mais baixo de coleção no .NET. Todos os arrays derivam diretamente de `System.Array` e são as únicas coleções com suporte direto no CLR. Matrizes unidimensionais implementam `IList<T>` (e as interfaces que elas estendem) e as interfaces não genéricas `IList` e `ICollection`; matrizes retangulares suportam apenas interfaces não genéricas. Arrays são sempre mutáveis em termos de seus elementos, mas sempre fixos em termos de tamanho. Todos os métodos mutantes das interfaces de coleção (como `Adicionar` e `Remover`) são implementados explicitamente e lançam `NotSupportedException`.

Matrizes de tipos de referência são sempre covariantes; há uma conversão implícita de uma referência `Stream[]` para `Object[]`, por exemplo, e uma conversão explícita ao contrário.<sup>2</sup> Isso significa que as alterações no array devem ser verificadas em tempo de execução – o próprio array sabe de que tipo ele é. é, então se você tentar armazenar um não-`Stream`

<sup>1</sup> A pesquisa binária tem complexidade  $O(\log n)$ ; uma pesquisa linear é  $O(n)$ .

<sup>2</sup> Um tanto confuso, isso também significa que há uma conversão implícita de `Stream[]` para `IList<Object>`, mesmo que o próprio `IList<T>` seja invariante.

referência em um Stream[] convertendo a referência da matriz em um Object[] primeiro, um ArrayTypeMismatchException será lançada.

Existem dois tipos diferentes de array no que diz respeito ao CLR . Um vetor é um matriz unidimensional com limite inferior de 0; qualquer outra coisa conta como uma matriz. Os vetores têm melhor desempenho e são o que você quase sempre usa em C#. Uma matriz do formulário T[,] ainda é um vetor, mas com um tipo de elemento T[]; apenas matrizes retangulares em C#, como new string[10, 20], acabam como arrays na terminologia CLR . Você não pode criar um array com um limite inferior diferente de zero diretamente em C# - você deve usar Array.CreateInstance, que permite especificar limites inferiores, comprimentos e o tipo de elemento individualmente. Se você criar uma matriz unidimensional com um limite inferior diferente de zero, você não pode então convertê-lo com sucesso para T[] - o compilador permitirá a conversão, mas falhará em tempo de execução.

O compilador C# tem suporte integrado para matrizes de diversas maneiras. Não só faz sabe como criá-los e indexá-los, mas também os apoia diretamente em loops foreach ; se você iterar usando uma expressão que é conhecida como um array em tempo de compilação, essa iteração usará a propriedade Length e o indexador do array, em vez de do que criar um objeto iterador. Isso é mais eficiente, mas a diferença de desempenho geralmente é insignificante.

Assim como List<T>, arrays suportam métodos como ConvertAll, FindAll e Binary-Search, embora no caso de arrays, esses sejam métodos estáticos da classe Array , tomando o array como o primeiro parâmetro.

Voltando ao meu primeiro ponto, arrays são estruturas de dados de baixo nível. Eles estão importantes como blocos de construção para muitas outras coleções e são eficientes em situações apropriadas, mas você deve pensar duas vezes antes de usá-los demais. Mais uma vez, Eric blogou sobre esse assunto, rotulando-os de “um tanto prejudiciais” (veja <http://mng.bz/3jd5>). Não quero exagerar neste ponto, mas vale a pena estar ciente das deficiências dos arrays ao escolher um tipo de coleção.

### B.2.3 LinkedList<T>

Quando uma lista não é uma lista? Quando é uma lista vinculada. LinkedList<T> é uma lista de várias maneiras—em particular, é uma coleção que mantém a ordem em que você adiciona itens - mas não implementa IList<T>. Isto porque não obedece ao contrato implícito de acesso eficiente por índice. É uma lista duplamente vinculada clássica da ciência da computação: ela mantém um nó principal e um nó final, e cada nó tem uma referência ao nó seguinte e anterior na lista. Cada nó é exposto como LinkedListNode<T>, que é útil se você deseja manter um ponto de inserção/remoção em algum lugar no meio de a lista. A lista mantém explicitamente um tamanho, portanto, acessar a propriedade Count é eficiente.

Listas vinculadas são inefficientes em termos de espaço em comparação com listas baseadas em array, e eles não suportam operações indexadas, mas são rápidos na inserção ou remoção de elementos em pontos arbitrários da lista, desde que você tenha uma referência ao nó no ponto relevante. Essas operações têm complexidade O(1), pois tudo o que é necessário é corrigir as referências seguintes/anteriores nos nós circundantes. Inserindo ou removendo

do início ou do final da lista é apenas um caso especial disso, onde sempre há acesso imediato ao nó que você precisa alterar. Iterar (para frente ou para trás) é também eficiente, pois basta seguir a cadeia de referências.

Embora `LinkedList<T>` implemente os métodos padrão, como `Add` (que adiciona ao final da lista), sugiro usar os métodos explícitos `AddFirst` e `AddLast` para deixar claro exatamente o que está acontecendo. Existem `RemoveFirst` e Métodos `RemoveLast` e propriedades `First` e `Last`. Todos estes retornam os nós dentro da lista, em vez dos valores desses nós; as propriedades retornam uma referência nula se a lista estiver vazia.

#### B.2.4 `Collection<T>`, `BindingList<T>`, `ObservableCollection<T>` e

##### `KeyedCollection< TKey, TItem >`

`Collection<T>` é membro do namespace `System.Collections.ObjectModel`, como são todas as listas restantes que veremos. Assim como `List<T>`, ele implementa tanto o genérico e interfaces de coleta não genéricas.

Embora você possa usar `Collection<T>` sozinho, ele é mais comumente usado como base aula. Ele sempre atua como um wrapper para outra lista: você especifica uma no construtor ou uma nova `List<T>` será criada nos bastidores. Todas as ações mutantes no coleção passa por métodos virtuais protegidos (`InsertItem`, `SetItem`, `RemoveItem`, e `ClearItems`); classes derivadas podem interceptar esses métodos, gerando eventos ou fornecendo outro comportamento personalizado. A lista empacotada é acessível para classes derivadas através do Propriedade de itens. Se esta lista for somente leitura, os métodos mutantes públicos lançam uma exceção em vez de chamar os métodos virtuais; você não precisa verificar isso novamente quando substituí-los.

`BindingList<T>` e `ObservableCollection<T>` derivam de `Collection<T>` em a fim de fornecer capacidades vinculativas. `BindingList<T>` está disponível desde .NET 2.0, mas `ObservableCollection<T>` foi introduzido com o Windows Presentation Fundação (WPF). Claro, você não precisa usá-los para vinculação de dados no usuário interfaces – você pode ter seus próprios motivos para estar interessado em alterações em uma lista. Em nesse caso, você deverá ver qual coleção fornece notificações de uma forma mais útil quando você está decidindo qual usar. Observe que você só será notificado sobre alterações que ocorrer através do invólucro; se a lista subjacente for compartilhada com outro código que possa modifique-o por conta própria, isso não gerará nenhum evento no wrapper.

`KeyedCollection< TKey, TItem >` é uma espécie de híbrido entre uma lista e um dicionário, permitindo que um item seja obtido por chave e também por índice. Ao contrário dos dicionários normais, a chave deve estar efetivamente incorporada ao item, em vez de ser independente. Em muitos casos isto é natural; por exemplo, você pode ter um tipo `Cliente` com uma propriedade `CustomerID`. `KeyedCollection<,>` é uma classe abstrata; classes derivadas implementam o método `GetKeyForItem` para fornecer uma maneira de extrair uma chave de qualquer item adicionado à coleção. Em nosso cenário de cliente, o método `GetKeyForItem` seria basta retornar o ID do cliente fornecido. Assim como um dicionário, a chave deve ser única dentro da coleção – a tentativa de adicionar outro item com a mesma chave falhará com

uma exceção. Embora chaves nulas não sejam permitidas, GetKeyForItem pode retornar nulo (se o tipo de chave é um tipo de referência), caso em que a chave será ignorada (e o item não poderá ser buscado por sua chave).

### B.2.5 `ReadOnlyCollection<T>` e `ReadOnlyObservableCollection<T>`

Nossas duas listas finais são mais wrappers, fornecendo acesso somente leitura mesmo quando o a lista subjacente é mutável. Novamente, interfaces de coleção genéricas e não genéricas são implementados. Uma mistura de implementação de interface explícita e implícita é usada para que os chamadores que usam uma expressão em tempo de compilação do tipo concreto sejam desencorajados de usar operações mutantes que falharão.

`ReadOnlyObservableCollection<T>` deriva de `ReadOnlyCollection<T>` e implementa o mesmo `INotifyCollectionChanged` e `INotifyPropertyChanged` interfaces como `ObservableCollection<T>`. Um `ReadOnlyObservableCollection<T>` a instância só pode ser construída com uma lista de apoio `ObservableCollection<T>`. Embora a coleção ainda seja somente leitura para os chamadores, eles podem observar alterações feito em outro lugar na lista de apoio.

Embora normalmente eu aconselhe usar uma interface ao decidir o tipo de retorno de métodos em uma API, pode ser útil expor deliberadamente `ReadOnlyCollection<T>` a fornecem uma indicação clara aos chamadores de que eles não poderão modificar a coleção retornada. Mas você ainda precisará documentar se a coleção subjacente poderia ser mudou em outro lugar, ou se é efetivamente constante.

## B.3 Dicionários

As opções de dicionários na estrutura são muito mais limitadas do que as de listas.

Existem apenas três implementações não simultâneas convencionais do `IDictionary< TKey, TValue >`, embora também seja implementado pelo `ExpandoObject` (como você viu em capítulo 14), `ConcurrentDictionary` (que veremos junto com outras coleções simultâneas) e `RouteValueDictionary` (usado para rotear solicitações da Web, especialmente no ASP.NET MVC).

Apenas para lembrar, o objetivo principal de um dicionário é fornecer uma linguagem eficiente pesquisa de uma chave para um valor.

### B.3.1 `Dicionário< TKey, TValue >`

A menos que você tenha requisitos especializados, `Dictionary< TKey, TValue >` é o padrão escolha do dicionário da mesma maneira que `List< T >` é a implementação de lista padrão. Ele usa uma tabela hash para implementar uma pesquisa eficiente, embora isso signifique que a eficiência do dicionário depende de quanto boa é sua função de hash. Você pode usar as funções padrão de hash e igualdade (chamadas para `Equals` e `GetHashCode` dentro dos próprios objetos-chave) ou especificar um `IEqualityComparer< TKey >` como um argumento construtor.

O caso de uso mais simples é implementar um dicionário com chaves de string, que usa o chaves sem distinção entre maiúsculas e minúsculas, conforme mostrado no código a seguir:

```
var comparar = StringComparer.OrdinalIgnoreCase; var dict = new
Dicionário<String, int>(comparador); dict["TESTE"] = 10;
Console.WriteLine(dict["teste"]);
```

### ← Saídas

**10** Embora as chaves dentro de um dicionário devam ser exclusivas, os códigos hash não o são. É perfeitamente aceitável que duas chaves desiguais tenham o mesmo hash; isso é conhecido como *colisão de hash* e, embora reduza um pouco a eficiência do dicionário, ele ainda funcionará corretamente. O dicionário *falhará* se as chaves forem mutáveis e alterarem seus códigos hash após serem inseridas no dicionário. Chaves de dicionário mutáveis são quase sempre uma má ideia, mas se você realmente *precisar* usá-las, certifique-se de não alterá-las após a inserção.

Os detalhes exatos da implementação da tabela hash não são especificados e podem mudar com o tempo, mas um aspecto importante pode causar confusão: *não há garantia de ordem dentro de Dictionary< TKey, TValue >*, mesmo que possa parecer assim. Se você adicionar itens a um dicionário e depois iterá-lo, poderá ver os itens saindo no pedido de inserção, mas *não confie nisso*. É um tanto lamentável que, como uma peculiaridade da implementação, simplesmente adicionar entradas sem nunca excluir nenhuma tenda a preservar a ordem - uma implementação que embaralhasse a ordem naturalmente provavelmente causaria menos confusão.

Assim como List<T>, Dictionary< TKey, TValue > mantém suas entradas em uma matriz e as expande quando necessário, levando à expansão O(1) amortizada. O acesso por chave também é O(1) assumindo um hash razoável; se todas as chaves tiverem o mesmo código hash, você terá acesso O(n) porque o dicionário precisa verificar a igualdade de cada chave. Na maioria dos cenários práticos, isso não é um problema.

### B.3.2 SortedList< TKey, TValue > e SortedDictionary< TKey, TValue >

Um observador casual poderia imaginar que uma classe chamada SortedList<,> seria uma lista... mas não. Ambos os tipos são, na verdade, dicionários e nenhum deles implementa IList<T>. Pode ser mais informativo para eles serem nomeados ListBacked-SortedDictionary e TreeBackedSortedDictionary, mas é tarde demais para mudar agora.

Há muitos pontos em comum entre essas duas classes: ambas usam um IComparer < TKey > em vez de um IEqualityComparer < TKey > para comparar chaves e ambas mantêm as chaves de forma classificada, com base nessa comparação. Ambos possuem desempenho O(log n) ao encontrar valores, realizando efetivamente uma busca binária. Mas suas estruturas de dados internas são muito diferentes: SortedList<,> mantém uma matriz de entradas que são mantidas classificadas, enquanto SortedDictionary<,> usa uma estrutura de árvore vermelha e preta (veja a entrada da Wikipedia em <http://mng.bz/K1S4>). Isso leva a diferenças significativas nos tempos de inserção e remoção, bem como na eficiência da memória. Se você estiver criando um dicionário a partir de dados classificados principalmente, um SortedList<,> será preenchido com eficiência; se você imaginar as etapas envolvidas para manter um List<T> classificado, poderá ver que adicionar um único item ao final da lista é barato (O(1) se você ignorar a expansão), enquanto adicionar itens aleatoriamente é caro, porque envolve copiar itens existentes (O(n) no pior caso). Adicionar itens à árvore balanceada em SortedDictionary<,> é sempre bastante

barato (complexidade O ( $\log n$ )), mas envolve um nó de árvore separado no heap para cada entrada, levando a mais sobrecarga e fragmentação de memória do que a matriz de estruturas de entrada de chave/valor em um `SortedList<, >`.

Ambas as coleções expõem suas chaves e valores como coleções separadas e em ambas casos em que a coleção retornada está ativa e mudará conforme o dicionário subjacente mudanças. Mas as coleções expostas por `SortedList<, >` implementam `IList<T>`, então você pode acessar efetivamente as entradas por índice de chave classificado, se realmente desejar.

Não quero desanimá-lo muito com toda essa conversa sobre complexidade. A não ser que tu tiver uma quantidade muito grande de dados, você provavelmente não precisará se preocupar muito com quais implementação que você usa. Se for provável que você tenha um grande número de entradas em seu dicionário, você deve analisar cuidadosamente as características de desempenho de ambas as coleções para decidir qual delas usar.

### B.3.3 `ReadOnlyDictionary< TKey, TValue >`

Quando você estiver familiarizado com `ReadOnlyCollection<T>`, que discutimos na seção B.2.5, `ReadOnlyDictionary< TKey, TValue >` não deve trazer surpresas para você. De novo, é simplesmente um wrapper em torno de uma coleção existente (um `IDictionary< TKey, TValue >` desta vez) que oculta todas as operações mutantes por trás da implementação explícita da interface e lança uma `NotSupportedException` se forem chamadas de qualquer maneira.

Tal como acontece com as listas somente leitura, isso é apenas um wrapper; se a coleção subjacente (aquele passado para o construtor) for modificado, essas modificações ficarão visíveis através do invólucro.

## B.4 Conjuntos

Antes do .NET 3.5, não havia nenhuma coleção de conjuntos públicos na estrutura. Quando os desenvolvedores precisavam de algo para representar um conjunto no .NET 2.0, eles normalmente usariam um Dicionário`<, >`, usando os itens definidos como chaves e fornecendo valores fictícios. Essa situação foi melhorado um pouco com `HashSet<T>` no .NET 3.5, e agora o .NET 4 adicionou um `SortedSet<T>` e uma interface `ISet<T>` comum. Embora logicamente seja uma interface definida poderia consistir apenas em operações Adicionar/Remover/Contém, `ISet<T>` especifica um número de outras operações para manipular o conjunto (`ExceptWith`, `IntersectWith`, `Symmetric ExceptWith` e `UnionWith`) e para testar várias condições mais complexas (`Set Equals`, `Overlaps`, `IsSubsetOf`, `IsSupersetOf`, `IsProperSubsetOf` e `IsProper SupersetOf`). Os parâmetros para todos esses métodos são expressos em termos de

`IEnumerable<T>` em vez de `ISet<T>`, o que é inicialmente surpreendente, mas significa que sets interagem com o LINQ de maneira natural.

### B.4.1 `HashSet<T>`

Um `HashSet<T>` é efetivamente um `Dictionary<, >` sem os valores. Ele tem as mesmas características de desempenho e, novamente, você pode especificar um `IEqualityComparer<T>` para personalizar como os itens são comparados. Novamente, você não deve confiar em um `HashSet<T>` mantendo a ordem em que você adiciona valores.

Um recurso adicional suportado por `HashSet<T>` é o método `RemoveWhere` , que remove qualquer entrada que corresponda a um determinado predicado. Isso permite podar um conjunto sem se preocupar com a proibição normal de modificar uma coleção enquanto você itera sobre isso.

#### B.4.2 `SortedSet<T>` (.NET 4)

Assim como a comparação anterior de `HashSet<T>` com `Dictionary<,>`, um `SortedSet<T>` é como um `SortedDictionary<,>` sem valor. Ele mantém uma árvore vermelha e preta de valores, fornecendo complexidade  $O(\log n)$  para adição, remoção e verificação de contenção. Quando você iterar sobre o conjunto, os valores serão produzidos em uma ordem de classificação.

Ele fornece o mesmo método `RemoveWhere` que `HashSet<T>` (apesar de não estar em a interface) e adicionalmente fornece propriedades (`Min` e `Max`) para retornar os valores mínimo e máximo. Um método mais intrigante é `GetViewBetween`, que retorna outro `SortedSet<T>` oferecendo uma visualização do conjunto original entre um valor inferior e limite superior, ambos inclusivos. Esta é uma visualização ao vivo mutável - alterações no vista são refletidas no conjunto original e vice-versa. O exemplo a seguir demonstra isso:

```
var baseSet = new SortedSet<int> { 1, 5, 12, 20, 25 };
var view = baseSet.GetViewBetween(10, 20);
visualizar.Adicionar(14);
Console.WriteLine(baseSet.Count);                                ← Saídas 6
foreach (valor int em vista)
{
    Console.WriteLine(valor);                                     ← Saídas 12, 14, 20
}
```

Embora `GetViewBetween` seja conveniente, não é totalmente gratuito: operações na visualização pode ser mais caro do que o esperado, a fim de manter a consistência interna. Em particular, acessar a propriedade `Count` de uma visualização é uma operação  $O(n)$  se a propriedade subjacente set mudou desde a última caminhada na árvore. Como todas as ferramentas poderosas, esta deve ser usada com cuidado.

Um recurso final do `SortedSet<T>`: ele expõe um método `Reverse()` que permite para iterar sobre ele na ordem inversa. Isso não é usado por `Enumerable.Reverse()`, que armazena em buffer o conteúdo da sequência em que é chamado. Se você sabe que deseja acessar um conjunto classificado na ordem inversa, pode ser útil manter uma expressão do tipo `SortedSet<T>` em vez de usar um tipo de interface mais geral, para que você possa acessar este implementação mais eficiente.

### B.5 Fila<T> e Pilha<T>

Filas e pilhas são essenciais em todos os cursos de ciência da computação. Eles são às vezes referidas como estruturas FIFO (primeiro a entrar, primeiro a sair) e LIFO (último a entrar, primeiro a sair), respectivamente. A ideia básica é a mesma para ambas as estruturas de dados: você adiciona itens à coleção e em algum outro momento você os remove. A diferença é a ordem em que eles são removidos: uma fila funciona como uma fila em uma loja, onde a primeira pessoa a entrar na

a fila é a primeira a ser atendida; uma pilha funciona como uma pilha de pratos onde o último prato colocado no topo é o primeiro a ser retirado. Um uso comum para filas e pilhas é manter uma lista de itens de trabalho ainda a serem processados.

Assim como no `LinkedList<T>`, embora você possa usar os métodos normais da interface de coleção para acessar filas e pilhas, recomendo usar os específicos da classe para deixar seu código mais claro.

### B.5.1 Fila<T>

`Queue<T>` é implementado com um buffer circular: essencialmente ele mantém um array, com um índice lembrando o próximo slot ao qual adicionar um item, e outro índice lembrando o próximo slot do qual retirar um item. Se o índice add alcançar o índice remove, o conteúdo será copiado para um array maior.

`Queue<T>` fornece os métodos `Enqueue` e `Dequeue` para adicionar e remover itens; um método `Peek` permite que você veja qual item será retirado da fila em seguida, sem realmente removê-lo. Tanto `Dequeue` quanto `Peek` lançam `InvalidOperationException` se forem chamados em uma fila vazia. A iteração na fila produz valores na ordem em que seriam retirados da fila.

### B.5.2 Pilha<T>

A implementação `Stack<T>` é ainda mais simples que `Queue<T>` — você pode pensar nela como sendo uma `List<T>`, mas com um método `Push` para adicionar um novo item ao final da lista, `Pop` para remover o item final e `Peek` para ver o item final sem removê-lo. Novamente, `Pop` e `Peek` lançam `InvalidOperationException` quando chamados em uma pilha vazia. A iteração na pilha produz valores na ordem em que seriam exibidos – portanto, o valor adicionado mais recentemente é gerado primeiro.

## B.6 Coleções simultâneas (.NET 4)

Como parte das extensões paralelas no .NET 4, há várias novas coleções em um novo namespace `System.Collections.Concurrent`. Eles são projetados para serem seguros diante de operações simultâneas de vários threads, com relativamente pouco bloqueio. O namespace também contém três classes que são usadas para particionar coleções para operações paralelas, mas não as veremos aqui.

### B.6.1 IProducerConsumerCollection<T> e BlockingCollection<T>

Três das novas coleções implementam a nova interface `IProducerConsumerCollection<T>`, que foi projetada para ser usada com `BlockingCollection<T>`. Ao descrever filas e pilhas, mencionei que elas são frequentemente usadas para armazenar itens de trabalho para processamento posterior; o padrão produtor/consumidor é uma forma de executar esses itens de trabalho simultaneamente. Às vezes, há um único thread produtor criando trabalho e vários threads consumidores executando os itens de trabalho. Noutros casos, os consumidores também podem ser produtores; por exemplo, um rastreador da web pode processar uma página da web e descobrir mais links para serem rastreados posteriormente.

`IProducerConsumerCollection<T>` atua como uma abstração para o armazenamento de dados de o padrão produtor/consumidor, e `BlockingCollection<T>` envolve isso em um formato fácil de usar e também fornece a capacidade de limitar quantos itens podem ser armazenados em buffer em qualquer momento. `BlockingCollection<T>` assume que nada mais será adicionado a a coleção embrulhada diretamente; todos os interessados deverão utilizar a embalagem para adicionando e removendo itens de trabalho. As sobrecargas do construtor que não levam em conta O parâmetro `IProducerConsumerCollection<T>` usa um `ConcurrentQueue<T>` para backup de armazenamento.

O `IProducerConsumerCollection<T>` fornece apenas três métodos particularmente interessantes: `ToArray`, `TryAdd` e `TryTake`. `ToArray` copia o conteúdo atual de a coleção para um novo array; este é um instantâneo da coleção no momento em que o método é chamado. `TryAdd` e `TryTake` seguem o padrão `TryXXX` normal , retornando um valor booleano para indicar sucesso ou falha, e fazem o que você espera: tente adicionar um item à coleção ou tente remover um da coleção. Permitir um modo de falha eficiente reduz a necessidade de bloqueio. Em uma fila`<T>`, por exemplo, você deseja manter um bloqueio para combinar as operações de “teste se há algum item na fila” e “retirar um item da fila, se houver” — caso contrário, `Dequeue` poderá lançar uma exceção.

`BlockingCollection<T>` comportamento de bloqueio de camadas sobre essas camadas não bloqueadoras métodos, com uma série de sobrecargas para permitir que tempos limites e tokens de cancelamento sejam Especificadas. Normalmente você não precisará usar `BlockingCollection<T>` ou `IProducer-ConsumerCollection<T>` diretamente; você chamará outras partes de Extensões Paralelas que use-os para você. Vale a pena saber que eles estão lá, caso você precise do seu próprio comportamento personalizado.

### B.6.2 ConcurrentBag`<T>`, ConcurrentQueue`<T>`, e ConcurrentStack`<T>`

A estrutura vem com três implementações de `IProducerConsumerCollection<T>`. Essencialmente, eles diferem em termos da ordem em que os itens são recuperado; a fila e a pilha agem como você esperaria de seus processos não simultâneos equivalentes, enquanto `ConcurrentBag<T>` não garante nenhuma ordem.

Todos os três implementam `IEnumerable<T>` de maneira segura para threads. O iterador retornou by `GetEnumerator()` irá iterar sobre um instantâneo da coleção; você pode modificar o coleção enquanto você está iterando, e as alterações não serão vistas no iterador. Todos três também oferecem um método `TryPeek` semelhante ao `TryTake`, mas que não remove um valor da coleção. Ao contrário do `TryTake` este método não é especificado em `IProducer-ConsumerCollection<T>`.

### B.6.3 ConcurrentDictionary`< TKey, TValue >`

`ConcurrentDictionary< TKey, TValue >` implementa o `IDictionary` padrão Interface `< TKey, TValue >` (enquanto nenhuma das coleções simultâneas implementa `IList<T>`) e é essencialmente um dicionário baseado em hash seguro para threads. Ele suporta vários

threads lendo e escrevendo simultaneamente e também permite iteração segura de threads, embora diferentemente das três coleções da seção anterior, as modificações feitas ao dicionário durante a iteração pode ou não ser refletido no iterador.

Há mais do que apenas acesso seguro a threads. Enquanto as implementações normais de dicionário basicamente oferecem add-or-update através do indexador, e add-or-throw através do Add método, `ConcurrentDictionary< TKey, TValue >` oferece uma verdadeira miscelânea de opções. Você pode atualizar o valor associado a uma chave com base em seu valor anterior, obter um valor baseado em uma chave ou adicione-o se a chave não estava presente anteriormente, condicionalmente atualize um valor somente se ele for o que você esperava que fosse antes e muitas outras possibilidades, todas elas agindo atomicamente. É tudo desconcertante para começar, mas Stephen Toub da equipe de Extensões Paralelas tem uma postagem no blog fornecendo detalhes de quando você deve usar qual método (consulte <http://mng.bz/WMdW>).

#### B.7 Interfaces somente leitura (.NET 4.5)

O .NET 4.5 introduziu três novas interfaces de coleção: `IReadOnlyCollection<T>`, `IReadOnlyList<T>` e `IReadOnlyDictionary< TKey, TValue >`. Enquanto escrevo isto, eles não estão amplamente utilizados - mas vale a pena conhecê-los, principalmente para saber o que não são. A Figura B.2 mostra como eles se relacionam entre si e com as interfaces `IEnumerable`.

Se você pensou que `ReadOnlyCollection<T>` estava exagerando a verdade com seu nome, essas interfaces são ainda mais sorrateiras. Eles não apenas permitem que mutações sejam feitas

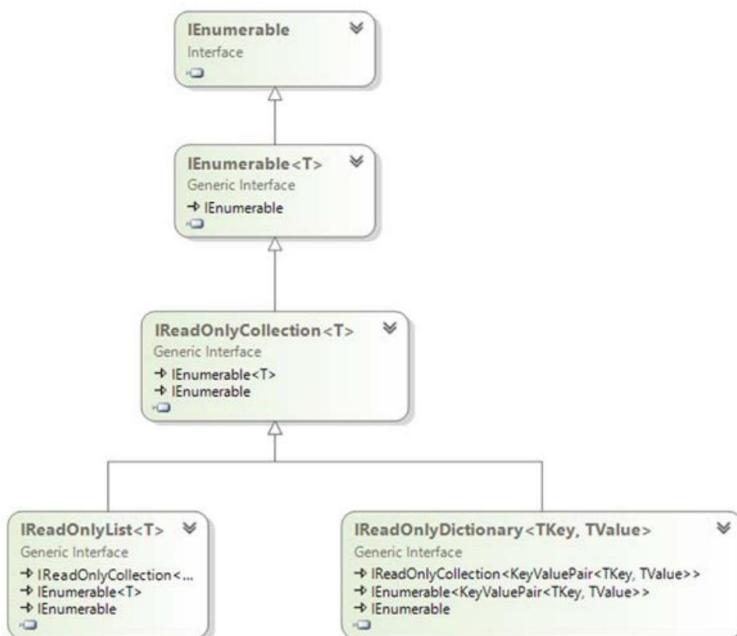


Figura B.2 Interfaces somente leitura no .NET 4.5

por outro código; eles até permitem mutações através do mesmo objeto, se este for uma coleção mutável. Por exemplo, `List<T>` implementa `IReadOnlyList<T>` mesmo que claramente não seja uma coleção somente leitura.

Isso não quer dizer que as interfaces não sejam úteis, é claro. Em particular, tanto `IReadOnlyCollection<T>` quanto `IReadOnlyList<T>` são covariantes em `T`, assim como `IEnumerable<T>` mas expoem mais operações. Infelizmente `IReadOnlyDictionary< TKey, TValue >` é invariante em ambos os parâmetros de tipo, em parte devido à implementação `IEnumerable<KeyValuePair< TKey, TValue >>` — que é invariante porque `KeyValuePair< TKey, TValue >` é uma estrutura, que é, portanto, invariante por si só. Além disso, o covariância de `IReadOnlyList<T>` significa que ele não pode expor nenhum método que aceite um `T`, como `Contains` e `IndexOf`. O grande benefício é que isso expõe um indexador a buscar itens por índice.

Não consigo me imaginar usando muito essas interfaces agora, mas no futuro acho eles serão muito importantes. No final de 2012, a Microsoft lançou sua primeira prévia de um Pacote NuGet de coleções imutáveis, chamado `Microsoft.Bcl.Immutable`. Uma BCL postagem no blog da equipe (<http://mng.bz/Xlqd>) fornece mais detalhes, mas fundamentalmente fornece o que diz na lata: coleções totalmente imutáveis, junto com congeláveis (mutáveis até eles estão congelados) coleções. Claro, se o tipo de elemento for mutável (como `StringBuilder`), isso só leva você até certo ponto, mas ainda estou animado com isso, apesar de tudo o que é normal. razões pelas quais a imutabilidade é útil.

## B.8 Resumo

O .NET Framework contém um rico conjunto de coleções (embora não seja particularmente rica coleção de conjuntos). Estes têm vindo a crescer gradualmente, juntamente com o resto do framework, embora as coleções mais comumente usadas sejam provavelmente `List<T>` e `Dictionary< TKey, TValue >` por algum tempo.

Certamente existem estruturas de dados que poderiam ser adicionadas no futuro, mas o benefício sempre deve ser ponderado em relação ao custo de adicionar algo à estrutura central. Talvez veremos APIs explicitamente baseadas em árvores no futuro, em vez de serem um detalhe de implementação das coleções existentes. Talvez veremos montes de Fibonacci, caches de referência fraca e coisas do gênero — mas, como você viu, já há muita coisa para os desenvolvedores absorverem e há o risco de sobrecarga de informações.

Se você precisa de uma estrutura de dados específica para o seu projeto, vale a pena procurar online para uma implementação de código aberto; As Power Collections da Wintellect têm um história particularmente forte como alternativa às coleções integradas (<http://powercollections.codeplex.com>). Mas na maioria dos casos, a estrutura provavelmente será adequada às suas necessidades. Esperamos que este apêndice tenha expandido ligeiramente seus horizontes em termos do que está disponível imediatamente.

## apêndice C

## Resumos de versões

Os números de versão no .NET às vezes podem ser confusos. A estrutura, o tempo de execução, o Visual Studio e o C# são todos numerados separadamente. Este apêndice é uma rápida guia sobre como eles se encaixam e os principais recursos de cada versão. Em cada caso, Descrevi os recursos das versões 2.0 e superiores; listando todas as características de .NET 1.0 e 1.1 seriam bastante inúteis.

## C.1 Principais versões da estrutura de desktop

Quando os desenvolvedores se referem aos lançamentos do .NET, geralmente se referem aos principais lançamentos da estrutura da área de trabalho. Na maioria dos casos, uma versão da estrutura foi acompanhada por uma versão do Visual Studio (ou Visual Studio .NET, como foi nomeado em 2002 e lançamentos de 2003). A exceção a isso foi o .NET 3.0, que era essencialmente apenas um conjunto de bibliotecas (embora essas bibliotecas fossem bastante significativas). Um conjunto de extensões do Visual Studio 2005 foi disponibilizado para os novos recursos, mas o Visual Studio 2008 continha mais apoio. A Tabela C.1 mostra qual versão de qual aspecto do framework foi lançado quando.

Quando o .NET 3.5 foi lançado, o .NET 2.0 SP1 e o .NET 3.0 SP1 também foram lançados; estes continham o 2.0 SP1 CLR e BCL. Da mesma forma, o lançamento do .NET 3.5 SP1 coincidiu com o .NET 2.0 SP2 e o .NET 3.0 SP2.

Tabela C.1 Versões da estrutura de desktop e seus componentes

Data	Estrutura	Estúdio visual	C#	CLR
Fevereiro de 2002	1,0	2002	1,0	1,0
Abril de 2003	1.1	2003	1.2	1.1
Novembro de 2005	2,0	2005	2,0	2,0
Novembro de 2006	3,0	(Extensões até 2005)	n / D	2,0
Novembro de 2007	3.5	2008	3,0	2.0SP1
Abril de 2010	4	2010	4,0	4.0 (não havia versão 3.0)
Agosto de 2012	4,5	2012	5,0	4,0 ou 4,5a

a. Isso depende do seu ponto de vista. Você descobrirá mais tarde.

O Visual Studio 2008 foi a primeira versão a oferecer suporte a *multitargeting*, permitindo escolher para qual versão da estrutura você deseja construir. Em muitos casos, você pode usar novos recursos do C# enquanto almeja uma versão anterior — esse é basicamente o caso se o recurso for implementado apenas pela mágica do compilador, sem qualquer suporte do CLR ou das bibliotecas. Mais informações sobre como fazer isso estão disponíveis no site do livro

(consulte <http://mng.bz/YpRB>) — em alguns casos, há soluções alternativas se um recurso não funcionar imediatamente. Vale a pena notar que se você direcionar o .NET 2.0 (não é possível direcionar 1.0 ou 1.1) do Visual Studio 2008 ou 2010, você estará realmente direcionando o service pack relevante (2.0 SP1 ou 2.0 SP2); isso significa que é possível criar código que use novos recursos de um service pack (uma introdução notável foi System.DateTimeOffset no 2.0 SP1) e descobrir que ele falha se você tentar executá-lo em uma máquina que realmente tenha a versão original do .NET 2.0. Pessoalmente, eu tentaria atualizar as máquinas para pelo menos executar o service pack mais recente e, de preferência, uma versão completa mais recente da estrutura.

## C.2 Recursos da linguagem C# Se você leu

o livro inteiro, deverá ser capaz de escrever esta seção sozinho. (É tentador deixar um monte de linhas em branco para você preencher, mas não sou tão preguiçoso assim.)

Um fato trivial: o número da versão 1.2 na tabela C.1 não é um erro de digitação; olhando as especificações, a Microsoft realmente pulou a versão 1.1 para lançar um compilador C# 1.2 com o .NET 1.1. As mudanças na versão 1.2 foram em sua maioria pequenas, mas houve uma mudança significativa no longo prazo: é somente a partir do C# 1.2 em diante que o código traduzido para um loop foreach testa se o iterador implementa IDisposable e o descarta adequadamente. Como você viu, essa mudança é crucial para blocos iteradores que possuem recursos para limpeza.

De qualquer forma, para completar, aqui estão os recursos de linguagem, junto com as referências do capítulo para obter mais detalhes.

### C.2.1 C# 2.0

Os principais recursos do C# 2 eram genéricos (veja o capítulo 3), tipos anuláveis (capítulo 4), métodos anônimos e outras melhorias relacionadas a delegados (capítulo 5) e blocos iteradores (capítulo 6). Além disso, vários recursos menores foram introduzidos: tipos parciais, classes estáticas, propriedades com diferentes modificadores de acesso para getters e setters, aliases de namespace, diretivas pragma e buffers de tamanho fixo. Veja o capítulo 7 para mais detalhes.

### C.2.2 C#3.0

C# 3 desenvolvido principalmente para LINQ, embora muitos recursos sejam úteis em outros lugares. Propriedades automáticas, digitação implícita de arrays e variáveis locais, inicializadores de objetos e coleções e tipos anônimos são todos abordados no capítulo 8. Expressões lambda e árvores de expressão (capítulo 9) ampliaram o progresso relacionado ao delegado feito na versão 2.0, e métodos de extensão (capítulo 10) forneceram o último ingrediente para expressões de consulta (capítulo 11). Os métodos parciais foram adicionados apenas no C# 3, mas são abordados com a inclusão de tipos parciais no capítulo 7.

### C.2.3 C# 4.0

O C# 4.0 possui alguns recursos voltados à interoperabilidade, mas não tem a mesma obstinação do C# 3.0. Novamente, há uma divisão razoavelmente clara entre os pequenos recursos mostrados no capítulo 13 (argumentos nomeados, parâmetros opcionais, melhor COM interoperabilidade, variação genérica) e o enorme recurso de digitação dinâmica (capítulo 14).

### C.2.4 C# 5.0

O C# 5.0 tem tudo a ver com a assincronia que vimos no capítulo 15, com dois outros recursos muito pequenos (mudanças na captura de variáveis foreach e atributos de informações do chamador) entrando sorrateiramente. capítulo 16. Embora a assincronia introduza apenas um único novo tipo de expressão (aguarde, dentro de uma função assíncrona ), muda enormemente o modelo de execução. Eu ia Argumentam que, mesmo que a equipe C# estivesse pronta para entregar outros grandes recursos da nova linguagem (e pelo que sei, eles estavam), retê-los por um tempo seria uma opção sensata. É importante que a comunidade C# realmente analise o async/await com cuidado, e isso levará tempo.

## C.3 Recursos da biblioteca Framework

Seria impossível listar todos os novos recursos da estrutura de maneira sensata aqui. Em particular, cada área da estrutura (Windows Forms, ASP.NET e assim por diante) obtém recursos extras em cada versão – não apenas na biblioteca principal de classes base. eu inclui os recursos que acredito serem os destaques mais importantes. O MSDN tem uma lista muito mais abrangente em <http://mng.bz/6tiZ>.

### C.3.1.NET 2.0

Os maiores recursos nas bibliotecas 2.0 suportavam recursos do CLR e linguagens: genéricos e tipos anuláveis. Embora os tipos anuláveis não exigissem muitas alterações, várias das coleções genéricas com as quais você está acostumado estão presentes desde o .NET 2.0, e a API de reflexão teve que ser atualizada de acordo.

Muitas áreas receberam atualizações relativamente pequenas, como suporte para compactação, vários conjuntos de resultados ativos (MARS) em uma única conexão com o SQL Server e muitos Métodos de E/S auxiliares estáticos , como File.ReadAllText. Provavelmente é justo dizer isso estas não foram tão significativas quanto as mudanças nas estruturas de interface do usuário.

ASP.NET ganhou páginas mestras, habilidades de pré-compilação e vários novos controles. O Windows Forms deu um grande salto em termos de habilidades de layout com TableLayoutPanel e classes semelhantes, bem como obter melhor suporte para melhorias de desempenho, como como buffer duplo, um novo modelo de vinculação de dados e implantação ClickOnce. Back-groundWorker foi introduzido no .NET 2.0 para facilitar a atualização segura de uma UI em aplicativos multithread; não faz parte estritamente do Windows Forms, embora isso tenha sido seu principal caso de uso até o Windows Presentation Foundation chegar ao .NET 3.0. Falando nisso...

### C.3.2.NET 3.0

O .NET 3.0 foi um tanto curioso como uma versão principal, sem alterações de CLR , sem linguagem alterações e nenhuma alteração nas bibliotecas existentes. Em vez disso, consistia em quatro novas bibliotecas:

- ÿ Windows Presentation Foundation (WPF) é a interface de usuário de última geração estrutura; esta foi uma revolução e não uma evolução do Windows Forms, embora os dois possam viver lado a lado. Possui um modelo bem diferente do Windows Forms, sendo de natureza muito mais composicional. A interface de usuário do Silverlight é baseada em WPF.
- ÿ Windows Communication Foundation (WCF) é uma arquitetura para construir aplicações orientadas a serviços; é extensível em vez de ser limitado a um único protocolo e visa unificar os canais de comunicação existentes do tipo RPC, como o controle remoto.
- ÿ O Windows Workflow Foundation (WF) é um sistema para criar aplicativos de fluxo de trabalhoções.
- ÿ O Windows CardSpace é um sistema de identidade seguro.

Destas quatro áreas, o WPF e o WCF floresceram, enquanto o WF e o CardSpace aparecem não ter decolado tão bem. Isso não quer dizer que as últimas tecnologias não estejam sendo usados, ou que não se tornarão mais importantes no futuro, mas não estão nem perto tão difundido quanto escrevo isto.

### C.3.3.NET 3.5

A grande novidade do .NET 3.5 foi o LINQ, suportado pelo C# 3.0 e VB 9. Isso incluiu LINQ to Objects, LINQ to SQL, LINQ to XML e suporte a árvore de expressão subjacente a isso.

Outras áreas também ganharam recursos importantes: ficou muito mais fácil usar AJAX em ASP.NET, WCF e WPF ganharam, cada um, uma série de melhorias, uma estrutura complementar (System.AddIn) foi introduzida, vários novos algoritmos de criptografia foram incluído e muito mais. Como um desenvolvedor interessado em APIs simultâneas e relacionadas ao tempo , sinto-me obrigado a chamar sua atenção para a introdução do Reader-WriterLockSlim e dos tão necessários tipos TimeZoneInfo e DateTimeOffset . Se você está usando o .NET 3.5 ou superior, mas ainda depende do DateTime em todos os lugares, você deve estar ciente de que existem melhores opções disponíveis.<sup>1</sup>

Os recursos de biblioteca mais notáveis do .NET 3.5 SP1 foram o Entity Framework e relacionados tecnologias ADO.NET , mas outras tecnologias também tiveram pequenas melhorias. Também mais importante ainda, o .NET 3.5 SP1 introduziu o Client Profile - uma versão menor do .NET Framework para desktop que não inclui muitas bibliotecas destinadas ao lado do servidor desenvolvimento. Isso permite um espaço de implantação menor para aplicativos somente cliente.

---

<sup>1</sup> Meu sentimento pessoal é que isso ainda não é suporte suficiente para o complexo e intrigante mundo das datas e vezes, por isso iniciei o projeto Noda Time (veja <https://code.google.com/p/noda-time/>), mas às pelo menos com TimeZoneInfo há finalmente uma maneira limpa de representar um fuso horário diferente do local.

### C.3.4.NET 4

Muito trabalho foi feito nas bibliotecas do .NET 4 por um longo tempo, em vários aspectos. O DLR é uma grande adição, e você também viu (muito brevemente) Extensões Paralelas em outros capítulos.

Como de costume, as tecnologias de interface do usuário apresentam uma série de melhorias, embora o foco das alterações do rich client seja o WPF em vez do Windows Forms.

Muitos ajustes foram feitos nas APIs principais existentes para torná-las muito mais fáceis de usar, como String.Join aceitando um IEnumerable<T> em vez de insistir em uma matriz de strings. Estas não são melhorias surpreendentes, mas se tornarem a vida de cada desenvolvedor um pouco mais simples, isso pode ter um grande impacto cumulativo. Você já viu como algumas das interfaces e delegados genéricos existentes se tornaram covariantes ou contravariantes (IEnumerable<T> se tornando IEnumerable<out T> e Action<T> se tornando Action<in T>, por exemplo), mas existem novos tipos para explorar também.

Há um novo namespace para cálculos numéricos, System.Numeric. No momento em que este artigo foi escrito, ele continha apenas os tipos BigInteger e Complex , mas não ficaria surpreso se BigDecimal se juntasse a eles no futuro. Existem outros novos tipos dentro do namespace System , como Lazy<T> para valores inicializados lentamente e uma família Tuple de classes genéricas que fornecem o mesmo tipo de funcionalidade que a classe Pair<T1, T2> do capítulo 3, mas para cima até oito parâmetros de tipo. Tuple também suporta *comparações estruturais*, conforme representado pelas novas interfaces IStructuralEquatable e IStructuralComparable no namespace System.Collections . Embora as classes completas de extensões reativas que você viu no capítulo 12 não estejam no .NET 4, as interfaces principais IObserver<T> e IObservable<T> estão no namespace System . Mencionei esses itens específicos porque, embora novas áreas como o Managed Extensibility Framework (MEF) recebam muita atenção, é fácil ignorar tipos simples como esses. É bom ver que o tempo está sendo gasto em toda a estrutura, não apenas em coisas novas e legais.

### C.3.5.NET 4.5

Novamente, o maior motivador das mudanças do .NET 4.5 é quase certamente a assincronia. Existem versões assíncronas de quase todas as APIs para as quais você poderia querer: se demorar um pouco, você poderá fazê-lo de forma assíncrona. A Biblioteca Paralela de Tarefas do .NET 4 foi expandida (e otimizada) para ajudar nisso também.

Existem muitas outras mudanças no .NET 4.5 e seria tolice tentar descrever todas elas. Até mesmo a página do MSDN listando os destaques (<http://mng.bz/6tiZ>) é mais longa do que eu gostaria de incluir aqui. Mas a maioria dessas mudanças dependerá do projeto que você está construindo, enquanto a assincronia em toda a plataforma provavelmente afetará a todos, ao longo do tempo.

## C.4 Recursos de tempo de execução (CLR)

As alterações do CLR costumam ser menos visíveis para muitos desenvolvedores do que os novos recursos de biblioteca e linguagem. Obviamente, existem alguns recursos particularmente brilhantes, como os genéricos, que chamarão a atenção de todos, mas outros são menos óbvios. O CLR também mudou com menos frequência do que a linguagem ou as bibliotecas da estrutura, pelo menos em termos de lançamentos principais.

### C.4.1 CLR 2.0

Além dos genéricos, o CLR exigiu uma mudança extra para dar suporte aos novos recursos de linguagem do C# 2: o comportamento de boxing e unboxing de tipos de valores anuláveis que exploramos no capítulo 4.

O CLR 2.0 teve outras mudanças importantes. Os mais significativos foram o suporte para processadores de 64 bits (x64 e IA64) e a capacidade de hospedar o CLR no SQL Server 2005. A integração do SQL Server exigiu que novas APIs de hospedagem fossem projetadas, para que o host pudesse ter muito mais controle sobre o CLR, incluindo como ele aloca memória e threads. Isso permite que um host diligente garanta que o código em execução no CLR não comprometa outros aspectos de um processo crítico, como um banco de dados.

O .NET 3.5 incluiu o CLR 2.0 SP1 e o .NET 3.5 SP1 incluiu o CLR 2.0 SP2; estes tiveram alterações relativamente pequenas, como ajustes no acesso que o código em um DynamicMethod tem a membros privados de outro tipo. A equipe CLR também está sempre buscando maneiras de melhorar o desempenho, com melhorias na coleta de lixo, no JIT, nos tempos de inicialização e assim por diante.

### C.4.2 CLR 4.0

Embora o CLR não tenha precisado mudar para acomodar o DLR, a equipe ainda tem trabalhado arduamente. Estes são alguns dos destaques:

- ÿ Melhorias no desempenho do empacotamento de interoperabilidade e na consistência com stubs de IL em todos os lugares (consulte esta postagem do blog do .NET Framework para obter detalhes: <http://mng.bz/56H6>)
- ÿ Um coletor de lixo em segundo plano para substituir o coletor simultâneo no CLR 2.0 ÿ Um modelo de segurança aprimorado baseado no conceito de transparência, que é o sucessor do Code Access Security (CAS)
- ÿ Equivalência de tipos, usada para suportar o recurso PIA incorporado do C# 4 ÿ Execução lado a lado de diferentes CLRs dentro do mesmo processo

O CLR no .NET 4.5 inclui diversas melhorias, principalmente em torno da coleta de lixo. Você pode pensar nisso como um lançamento menor, efetivamente. Além dos puros benefícios de desempenho, o CLR de 64 bits também suporta a opção de configuração <gcAllowVeryLargeObjects>, que permite a criação de arrays enormes, mesmo quando os elementos são estruturas grandes... supondo que você tenha memória, é claro. Em termos do número da versão, o quadro é um pouco complicado. Na documentação, você pode ver esta versão do CLR chamada CLR 4.5. No entanto, ele ainda se anuncia como 4.0 se você consultar a propriedade Environment.Version. Por exemplo, no momento em que escrevo o CLR , estou executando a versão 4.0.30319.18033 dos relatórios. Os números de compilação e revisão podem mudar com o tempo devido aos service packs.

Mais detalhes de todos os novos recursos estão disponíveis no Blog do .NET Framework (<http://blogs.msdn.com/b/dotnet>).

## C.5 Estruturas relacionadas

É raro que qualquer coisa na computação funcione bem com um modelo único, e o .NET é nenhuma exceção. Mesmo a estrutura de desktop não é realmente uma versão única: há o perfil do cliente, os JITs de 32 e 64 bits e os CLRs de servidor e estação de trabalho ajustados para tarefas diferentes. Além disso, existem estruturas separadas que possuem seus próprios históricos de versões, adaptados a diferentes ambientes.

### C.5.1 Estrutura Compacta

O Compact Framework foi originalmente voltado para dispositivos móveis rodando Windows Móvel. Desde então, foi redirecionado para Xbox 360, Windows Phone 7 e Sym-bian S60.

O cronograma de lançamento principal do Compact Framework tem tradicionalmente refletido o do a estrutura de desktop, embora não haja nenhuma versão correspondente ao .NET 3.0. Só para manter as coisas interessantes, a versão mais atualizada (usada por alguns Windows Mobile dispositivos e WP7) é a versão 3.7.

Nas primeiras versões do Quadro Compacto faltavam algumas funcionalidades bastante essenciais, que foram em grande parte preenchidas pelos esforços da comunidade; versões posteriores conectaram muitos das lacunas mais significativas, embora obviamente ainda seja um subconjunto da estrutura do desktop. A camada GUI depende da plataforma exata; por exemplo, no Xbox 360 você usaria XNA, o Windows Mobile oferece suporte ao Windows Forms e o WP7 oferece suporte a ambos XNA e Silverlight. O código executado no Compact Framework é compilado em JIT e coletado como lixo, embora o coletores do Compact Framework não seja geracional como aqueles na estrutura de desktop.

### C.5.2 Luz Prateada

Silverlight (<http://silverlight.net/>) tem como objetivo executar aplicativos dentro navegadores, ou (a partir do Silverlight 3) em um ambiente sandbox, geralmente originalmente instalado a partir de um navegador. Como tal, é um concorrente natural do Flash; tem o óbvio vantagem de permitir que desenvolvedores C# escrevam aplicativos em uma linguagem familiar contra uma biblioteca familiar. O Silverlight instala um CLR simplificado (chamado CoreCLR — consulte <http://mng.bz/G32M>) e biblioteca de classes - por exemplo, as coleções não genéricas não são suportados, nem o Windows Forms. A camada de apresentação do Silverlight é baseado em WPF, mas eles não são idênticos. Tem um apoio particularmente forte aos meios de comunicação social, com recursos como zoom profundo e streaming de vídeo adaptável.

O Silverlight 1 foi lançado em setembro de 2007, embora estivesse restrito a uma mistura de XAML para construir a UI e JavaScript para lógica. Não foi até Silverlight 2 foi lançado em outubro de 2008 que a experiência completa de entrega de aplicativos Silverlight criados com C# se tornou uma realidade. Alguns dos recursos do CoreCLR (lado a lado Hospedagem CLR em um único processo e modelo de segurança de transparência declarativa) agora são recursos do CLR de desktop para a versão 4.0. Também incluiu uma versão inicial do o tempo de execução de linguagem dinâmica.

O progresso continuou inabalável, com o Silverlight 3 sendo lançado em julho de 2009 com mais controles, mais codecs de vídeo, bem como aplicativos offline e fora do navegador. A equipe do Silverlight repetiu o ciclo de lançamento de nove meses, lançando o Silverlight 4 em na mesma semana que o .NET 4 com outra longa lista de novos recursos. O Windows Phone 7 suportava o Silverlight 3 e alguns recursos do Silverlight 4 e, então, quando o Windows SDK do telefone 7.1 foi lançado (para oferecer suporte ao telefone com uma versão de marca de consumidor de 7,5, só para adicionar confusão), que suportava mais Silverlight 4 novamente. Ambas as versões do Windows Phone 7.x usaram uma evolução do Compact Framework CLR.

O Windows Phone 8 oferece suporte à API Silverlight para compatibilidade com versões anteriores, mas também suporta a nova API do Windows Phone Runtime , que está mais próxima da API WinRT usada para aplicativos da Windows Store. Além disso, o Windows Phone 8 usa uma versão do CoreCLR em vez do Compact Framework.

O próprio Silverlight está agora morto em termos de desenvolvimento adicional. Embora eu tenha certeza de que muitos os desenvolvedores ainda o estiverem usando, não haverá novas versões lançadas. No entanto, o WinRT deve parecer muito familiar para os desenvolvedores do Silverlight. A Microsoft tentou fazer a transição dos aplicativos Silverlight para aplicativos da Windows Store é bastante suave.

### C.5.3 Microestrutura

O Micro Framework (veja <http://mng.bz/D9qy>) é uma pequena implementação do .NET, projetado para funcionar em dispositivos muito restritos. Ele não oferece suporte a genéricos , é interpretado em vez de compilado em JIT e vem com um conjunto limitado de classes, mas incluem uma camada de apresentação, construída em torno do WPF. Para economizar espaço, você só precisa para implantar as partes da estrutura que você realmente precisa – no mínimo, pode ocupar apenas 390 KB. Obviamente, esta é uma área de nicho, mas a capacidade de escrever código gerenciado para dispositivos embarcados tem grande apelo. Não será adequado para todas as situações – não é um sistema em tempo real, por exemplo – mas onde for aplicável, é provável que funcione. melhorar drasticamente a produtividade do desenvolvedor.

O histórico de lançamentos não seguiu o da estrutura de desktop: foi o primeiro visto no relógio SPOT em 2004, mas a versão 1.0 foi lançada em 2006. Desde então, iterado várias vezes em rápida sucessão. Versão 4.0 do Micro Framework enviado em 19 de novembro de 2009 - e em um movimento que ainda me encanta e surpreende, a maior parte desta versão foi lançada em código aberto sob a licença Apache 2.0. Algumas bibliotecas, como a pilha TCP/IP e implementações de criptografia, ainda são fechado por vários motivos; essas bibliotecas complementares podem ser baixadas em binário formulário para arquiteturas específicas.

### C.5.4 Tempo de Execução do Windows (WinRT)

WinRT não é outra versão do .NET – é uma plataforma Windows totalmente nova, introduzida no Windows 8. Seu objetivo é fornecer um ambiente em área restrita nas arquiteturas de processador x86 e ARM e oferece suporte a vários idiomas - principalmente C# e VB via .NET, C++/CX (um novo tipo de C++ direcionado especificamente ao WinRT) e JavaScript. É um

API não gerenciada , mas foi projetada para se integrar intimamente ao .NET, para que os desenvolvedores de C# e VB.NET possam realmente usar as mesmas APIs que os desenvolvedores de C++/CX e JavaScript. Não há necessidade de construir uma API wrapper em torno dele, como foi o caso do Win32 com Windows Forms. A API foi projetada tendo em mente a assincronia desde o início; usar assincronia é a maneira *natural* de desenvolver aplicativos direcionados ao WinRT.

Como o Windows 8 é um sistema operacional jovem, ainda temos que ver até que ponto isso vai dar certo no longo prazo, e os desenvolvedores que desejam criar aplicativos para rodar no Windows 8 ainda podem ter como alvo o desktop tradicional, mas está claro que a Microsoft acredita que WinRT é um caminho importante para o desenvolvimento do lado do cliente. Em particular, a API do Windows Phone e a API da Windows Store provavelmente convergirão cada vez mais no futuro.

## C.6 Resumo

Com tantas versões de tantos componentes diferentes, é fácil ficar confuso — e ainda mais fácil confundir outra pessoa. Como conselho final (e quero dizer final – é difícil inserir algo profundo e significativo em um índice), recomendo que você tente ser o mais claro possível sobre esse tópico ao se comunicar com outras pessoas. Se você estiver usando algo diferente da estrutura de desktop, diga isso. Se você for citar um número de versão, especifique exatamente o que você quer dizer: “3.0” pode significar usar C# 2.0 e .NET 3.0 ou pode significar usar C# 3.0 e .NET 3.5. Além de qualquer outra coisa, depois de ler este livro, você não terá *absolutamente nenhuma desculpa* para alegar que está usando “C# 3.5” ou “C# 4.5”, a menos que esteja deliberadamente tentando me enrolar.

---

Símbolos

: (dois pontos em restrições de tipo genérico)  
 71 :: qualificador de alias de  
 namespace  
 195 != operador comparando  
 referências 72 usando comparações  
 de referência 79 usando com parâmetros de tipo  
 irrestrito 79 ? (operador condicional)  
 115–116 ?? (operador de coalescência nulo) 123–126, 130  
 operador 478 /  
 compilador switch 417  
 #pragma diretivas. Veja diretivas pragma +  
 operador, combinando instâncias delegadas  
 usando  
 35 += operador  
 35 - operador, removendo instâncias delegadas usando  
 36 -- operador  
 36 = operador  
 220 ==  
 operador comparando  
 referências 72 usando comparações  
 de referência 79 usando com parâmetros de tipo  
 irrestritos 79 => (lambda ) operador 235

---

A

Action<T> delegado 142, 234, 399, 401  
 Propriedade ActiveSheet 418  
 Adicione o método 221–222, 347, 541  
 Método AddFirst 545  
 Método AddLast 545  
 AggregateException 483–484, 488  
 operadores de agregação (LINQ) 527–528  
 Albahari, Joe 26  
 Todos os operadores 537  
 Método dos ancestrais 350

Método AncestorsAndSelf 350  
 Método de anotações 350  
 funções anônimas 234  
 assíncronas 490–492 tipos  
 de retorno de 253–254  
 funções iteradoras anônimas 492  
 métodos anônimos 52, 69  
 agindo nos parâmetros 142–144  
 ignorando parâmetros delegados 146–148  
 retornando valores 145–146  
 com fechamentos de  
 variáveis capturadas  
 148–149 examinando o comportamento  
 de 149–151 vida útil estendida de  
 152–153 instanciações de variáveis locais  
 153–155 propósito  
 de 151 misturas compartilhadas e distintas de  
 155–156 tipos anônimos  
 307 em Visual Basic  
 228 membros da visão  
 geral 227–228 225–  
 227 inicializadores de projeção 228–  
 229 usos para 229–230  
 Qualquer método  
 537 ArgumentNullException 271  
 argumentos  
 passando por valor com COM 389–390  
 vs. parâmetros 372  
 Classe Array 543–544  
 Classe ArrayList 5, 42, 86  
 arrays  
 como tipos de referência  
 45 digitados implicitamente  
 224–225 ArrayTypeMismatchException 42,  
 544 como operador  
 403 e digitação dinâmica  
 Desempenho 436 de  
 123 usando com tipos anuláveis 123

Método AsAsyncAction 517  
 Método AsAsyncOperation 517 palavra-chave ascendente 304  
 Método AsOrdered 356  
 Método AsParallel 355–357  
 Método AsTask 517  
 Método AsUnordered 356  
 modificador assíncrono 23, 465, 472, 490  
 Delegado AsyncCallback 463 código assíncrono 22–23, 463–517 operações assíncronas funções anônimas 490–492 expressões aguardam fluxo de execução para 478–481 expansão de expressões complexas 477–478 visão geral 482–490  
 transformação do compilador e pilha 501–503  
 descompilação do código 503 visão geral 493–495, 500–501 ponto de entrada único para máquina de estado 498–500 estrutura do método esqueleto 495–496 estrutura da máquina de estado 497–498  
 tratamento de exceções cancelamento 488–490 desembrulhando exceções 482–485 envolvendo exceções ao lançar 485–488 modelo de execução 468–470 visão geral das funções 465 usando 465– 468 no WinRT 516–517  
 Interface INotifyCompletion 515–516 métodos aguardar expressões em 474–477 declarando 472 modelagem 471 tipos de retorno de 473–474, 481–482 padrões para coletar resultados à medida que chegam 508–511 coletar resultados em chamada única 507–508 padrão assíncrono baseado em tarefa 504–507 teste de unidade avançando tempo programaticamente 511–513 executando testes 513–514  
 Classe AsyncInfo 517  
 AsyncStateMachineAttribute 496 Estrutura AsyncTaskMethodBuilder 496, 499 Estrutura AsyncVoidMethodBuilder 496, 499 sufixo de atributo 201, 521 Método de atributos 350 propriedades implementadas automaticamente 7, 208–211 automatizando o Microsoft Word em C# 3 387–388 em C# 4 388–389

Operador médio 528 aguarda expressões 23, 465, 467–468, 470, 472 fluxo de execução para 478–481 expandindo expressões complexas 477–478 em métodos assíncronos 474–477 visão geral 482–490 restrições em 474  
 Método AwaitOnCompleted 501  
 Método AwaitUnsafeOnCompleted 501  
**B**  
 Classe BackgroundWorker 463, 470, 556 palavra-chave base 143 BCL (Biblioteca de classes base) 54 Método BeginInvoke 470 Métodos BeginXXX 78 padrão comportamental 159 Estrutura BigInteger 558 Classe BinaryExpression 242 Classe Binder 432 fichários definidos 411 em DLR 432 BindingList<T> classe 545–546 Bloch , Joshua 82 BlockingCollection<T> classe 550–551 corpo, de expressões lambda 257 bool tipo 120, 122 boxing definido 49  
 Nullable<T> 112– 113 instruções break 167– 168 buffers de tamanho fixo, em operações de código inseguras 199–201 usando 288 vs. streaming 273 campo construtor 498 padrão construtor 223 Classe de botão 193 tipo de byte e valores nulos 106  
**C**  
 C#, especificação de linguagem 27–28 C# 1 delegados combinando e removendo 35–36 novos recursos 51–53 delegados simples 30–35 eventos 36–37 tipo de produto em sistema de 5– 6 tipos boxing e unboxing 49–50 digitação explícita versus digitação implícita 40

- C# 1, limitações do sistema de tipos (continuação) 41–44
  - tipagem estática vs. digitação dinâmica 39–40
  - tipagem segura vs. tipagem não segura 40–41 tipos de valores e tipos de referência 45
- C# 2
  - métodos anônimos 142–158 variância delegada 137–141 visão geral do recurso 555 buffers de tamanho fixo em código não seguro 199–201 Atributo InternalsVisibleTo 201–204 blocos iteradores 163–165 conversões de delegação de grupo de métodos 136–137 declaração de aliases de namespace 194–195 aliases externos 196–197 alias de namespace global 195–196 visão geral 193–194
  - tipos de valor anuláveis ? modificador 115–116 como operador, o 123 atribuindo e comparando com nulo 116–118 conversões 119 operadores levantados 119–121 operador de coalescência nulo 123–126 lógica anulável 121–123 usando como operador 123
  - tipos parciais criando 184–186 definidos 183–184 usos para 186–187
  - diretivas pragma soma de verificação pragmas 198–199 definidos 197 pragmas de aviso 197–198 acessibilidade de propriedade getter/setter separada 192–193 classes estáticas 190–192 coleções fortemente tipadas em 6–7
- C#3
  - tipos anônimos membros da visão geral 227–228 225–227
  - métodos parciais 188–190 inicializadores de projeção 228–229 usos para 229–230
  - propriedades implementadas automaticamente 7, 208–211
  - automatizando o Microsoft Word em inicializadores de coleção 387–388 criando coleções com 220–221 preenchendo coleções dentro de inicializadores de objeto 222–223
  - visão geral dos recursos 555
- matrizes digitadas implicitamente 224–225 variáveis locais digitadas implicitamente melhores práticas para 215–216
- visão geral 211 prós e contras de 214–215 restrições em 213–214 usando a palavra-chave var 211–213 inicializadores de objeto padrão de construtor 223 coleções constantes 223 tipo de exemplo para 216–217 visão geral 224 definindo propriedades 217–219 definindo propriedades em objetos incorporados 219–220 configuração de testes de unidade 223 resolução de sobrecarga 258–260 inferência de tipo 252–258
- C# 4
  - automatizando o Microsoft Word em 388–389
  - Interoperabilidade COM chamando indexadores nomeados 390–391 vinculando PIAs 391–394 passando argumentos por valor 389–390 Automação de palavras em C# 3 387–388 Automação de palavras em C# 4 388–389
  - digitação dinâmica 409–459 visão geral do recurso 556
  - eventos semelhantes a campos 406–407 bloqueio 405–406 argumentos nomeados em 8–9, 378–382 parâmetros opcionais 372–378
  - Veja também variação genérica
  - C# 5 556
  - métodos assíncronos 490–492 aguardam expressões em 474–477 exceções 483–490 modelagem 471 visão geral 469–470 tipos de retorno de 473–474, 481–482
  - Veja também operações assíncronas atributos de informações do chamador 520–525 captura de variável de iteração fixa
  - 520
  - C++ e genéricos 94 modelos vs. genéricos 101–102 caches 433–434
  - Método de chamada (árvores de expressão) 248.458 locais de chamada 431–432
  - retornos de chamada 464.469 atributos de informações do chamador
  - Interface INotifyPropertyChanged 523–524 log com visão geral 522–523 521–522 usando sem .NET 4.5
  - 524–525

CallerFilePathAttribute 521  
 CallerLineNumberAtributo 521  
 CallerMemberNameAttribute 521, 523 cancelando operações assíncronas 488–490 tokens de cancelamento 356  
 Classe CancellationToken 483, 488  
 CancellationTokenSource classe 483, 488  
 Propriedade de capacidade 542  
 Método de captura 515  
 variáveis de iteração capturadas 520  
 variável externa capturada 148  
 variáveis capturadas e instâncias de múltiplas variáveis 153–154 alterações em C# 5 520 em diferentes escopos 155  
 Operador de conversão (LINQ) 298–300, 529 método 97 conversão, tipo 41 instruções catch e aguardar expressões em 474 e produzir instruções 164  
**CCR (Simultaneidade e Coordenação** Tempo de execução) 178–180  
 Propriedade de células 418 método de encadeamento chama 273–275  
 Canal 9 357  
 pragmas de soma de verificação 198–199 classes como tipos de referência 45 parâmetros de tipo para 64  
 Método ClearItems 545  
 Cleary, Stephen 470  
 CLI (Infraestrutura de Linguagem Comum) 24  
 Clique no evento 139 chamando sem testes de nulidade 147 em C# 1 135  
 Método Clone() 43 tipos fechados e campos estáticos 84 definidos 65  
**CLR (Common Language Runtime)** 24 recursos 2.0 559 recursos 4.0 559 e palavra-chave dinâmica 414 classe CodeChecksumPragma 199 API CodeDOM 411 inicializadores de coleção observáveis frios 360 criando coleções com 220–221 preenchendo coleções dentro de inicializadores de objeto 222–223 Collection<T> classe 545–546 Coleções da classe CollectionBase 43 Classe de matriz 543–544

simultâneo (.NET 4)  
 BlockingCollection<T> classe 550–551 ConcurrentBag<T> classe 551 ConcurrentDictionary< TKey, TValue > classe 551–552 ConcurrentQueue<T> classe 551 ConcurrentStack<T> classe 551 IProducerConsumerCollection<T> interface 550–551 coleções constantes 223  
**BindingList<T>** genérica classe 545–546 Collection<T> classe 545–546 Dictionary< TKey, TValue > classe 546–547 Namespace genérico 540–542 HashSet<T> classe 548–549 KeyedCollection< TKey, TItem > classe 545–546 Classe LinkedList<T> 544–545 Classe List<T> 542–543 Classe ObservableCollection<T> 545–546 Classe Queue<T> 550 interfaces somente leitura (.NET 4.5) 552–553 Classe ReadOnlyCollection<T> 546 ReadOnlyDictionary< TKey, TValue > classe 548 ReadOnlyObservableCollection<T> classe 546 SortedDictionary< TKey, TValue > classe 547–548 SortedList< TKey, TValue > classe 547–548  
 SortedSet<T> classe (.NET 4) 549 Stack<T> classe 550 consultando 12–14 dois pontos (:) em restrições de tipo genérico 71 Interoperabilidade COM e digitação dinâmica 20–22, 417–419 chamando indexadores nomeados 390–391 vinculando PIAs 391–394 passando argumentos por valor 389–390 Automação de palavras em C# 3 387–388 Automação de palavras em C# 4 388–389 Combine o método 35 combinando restrições 75–76 delegados (C# 1) 35–36 Common Language Infrastructure Anotado Padrão 95 Common Language Infrastructure. Consulte Tempo de Execução de Linguagem Comum da CLI. Consulte Pré-visualização da tecnologia da comunidade CLR. Consulte CTP Compact Framework 560 Comparar método e código de repetição 129 Classe anulável 114 Classe comparadora 80 Método CompareTo 79, 240

- Delegado de comparação 145
- comparações
  - comparações diretas 79–81
  - interfaces genéricas 80
  - suporte personalizado, estendendo LINQ para Objetos 365
  - com operador de coalescência nulo 129–131
- Método de compilação (árvores de expressão) 244
- compilador
  - e operações assíncronas e pilha 501–503 código de
  - descompilação 503 visão geral 493–495, 500–501 ponto de entrada único para máquina de estado 498–500 estrutura do método esqueleto 495–496 estrutura da máquina de estado 497–498 e erros de tempo de compilação de
    - digitação dinâmica 440–441
    - conversões entre tipos CLR e dinâmica 435–436 criação de sites
    - de chamada 436–438 expressões
    - avaliadas dinamicamente não do tipo dinâmica 436 visão geral 434–435
    - preservando o comportamento do compilador na execução tempo 438–440
    - usando como operador 436
  - e métodos de extensão 268 e LINQ 293–296 erros em tempo de compilação 440–441 Estrutura complexa 100, 558 Operador Concat 528 operadores de concatenação (LINQ) 528 Simultaneidade e coordenação Tempo de execução. Consulte CCR ConcurrentBag<T> classe 551 ConcurrentDictionary< TKey, TValue > classe 551–552
  - ConcurrentQueue<T> classe 551
  - ConcurrentStack<T> classe 551
  - operador condicional ( ? ) 115–116, 130 Método ConfigureAwait 506–507 coleções constantes 223 restrições, tipo. Consulte restrições de tipo, tipos construídos definidos 64 recuperando com o operador typeof 90 restrições de tipo de construtor 72–73, 76 construtores
    - e discrepância de padrões 73 para tipos anônimos 227 em LINQ to XML 347–349 estático 84–85
  - Contém operador (LINQ) 537, 541 continuações 468–469 Continue com o método 468
  - contravariância 52
    - para parâmetros delegados 138–139 nas interfaces 398–399 visão geral 395 covariância e contravariância simultâneas 399–401
  - operadores de conversão (LINQ) 529–531 conversões, restrições de tipo 73–75
    - envolvendo tipos anuláveis 119
    - Método de conversão 458
    - Método ConvertAll 67–68, 97, 229
    - Delegado conversor 400
    - Método CopyTo 264, 267, 269, 541
    - CoreCLR 560
    - Método de contagem 313, 528
    - Propriedade de contagem 426–427 covariância
      - 52 definida 42
      - nas interfaces 397–398 de matrizes 94 de tipos de retorno delegado 139–140 visão geral 395 covariância e contravariância simultâneas 399–401
    - Método CreateQuery 337, 340, 343 junções cruzadas em LINQ 314–317 Classe CSharpArgumentInfo 438 Classe CSharpCodeProvider 411 CTP (Prévia de Tecnologia Comunitária) 353, 492 Propriedade atual 87, 160, 163, 171, 288 provedores de consulta personalizada 342–344
  - D
  - Dapper 417
  - ausência de
    - dados de parâmetros opcionais e valores padrão 16 representando preço desconhecido 14–15 pipeline 159 bancos de dados e valores nulos 107 criando esquema para 330 Classe Leitor de Dados 273 Datas da classe 273 do DataSet, iterando entre 172 e 173 Estrutura DateTime
      - e valores nulos 106–107
      - retornando novos valores em vez de modificar 110
    - Estrutura DateTimeOffset 557 condição de deadlock 405 membro padrão 390 valores padrão 16, 77–79

Método DefaultIfEmpty 335, 533 execução adiada 272, 288–289 expressões de consulta degeneradas 301–302 Delegar classe 31, 35, 74 delegar palavra-chave 143, 145–146 delegados e coleta de lixo 32 como tipos de referência 45 capturar variáveis em fechamentos de métodos anônimos 148–149 examinar o comportamento de 149– 151 vida útil estendida de 152–153 instâncias de variáveis locais 153– 155 misturas de variáveis compartilhadas e distintas 155–156 propósito de 151 combinar e remover 35– 36 compilar árvores de expressão em 243–244 contravariância para parâmetros delegados 138–139 covariância de tipos de retorno delegados 139–140 definidos 31 variação genérica em 399 imutabilidade de 35 sintaxe aprimorada 134–136 problemas de incompatibilidade 141 ações delegadas inline com métodos anônimos agindo no parâmetro 142–144 ignorando parâmetros delegados 146–148 retornando valores de métodos anônimos 145–146 instanciando usando expressões lambda 52 expressões lambda como Func <...> tipos de delegados 234–235 visão geral 234 conversões de grupo de métodos 136–137 novos recursos 51–53 delegados simples criando instância 32–33 declarando o tipo de delegado 31 exemplo 33– 35 encontrando método para a ação da instância 31–32 invocando a instância 33 Método de desenfileiramento 550 Método DescendantNodes 350 Método DescendantNodesAndSelf 350 Método dos descendentes 350–351 Método DescendantsAndSelf 350 palavra-chave descendente 304 Dicionário<TKey, TValue> classe 546–547 e genéricos 62–63 implementando interface genérica 66 comparações diretas 79–81 Propriedade DisplayMember 423 Descarte o método 170, 174, 193

Operador distinto 538 DLR (tempo de execução de linguagem dinâmica) 20, 55, 250, 413 fichários 432 caches 433–434 locais de chamada 431– 432 projeto de código aberto 430 visão geral 429–431 receptores 432 regras 433–434 documentação 365 notação de ponto versus expressões de consulta operações que exigem notação de ponto 324–325 vantagens da expressão de consulta 325–326 simplicidade de ponto notação 325 dotPeek 118 DSL (linguagem específica de domínio) 281 digitação de pato 412, 426–427 Dyer, Wes 280 invocação dinâmica 522 palavra-chave dinâmica visão geral 413–414 usando 414–416 vs. palavra-chave var 416 interoperabilidade de linguagem dinâmica incorporada em C# Visão geral 420 422–423 armazenando e recuperando informações de ScriptScope 420–422 Tempo de execução de linguagem dinâmica. Consulte digitação dinâmica DLR 4 e reflexão 93 Interoperabilidade COM 417–419 interoperação com uma linguagem dinâmica 21–22 simplificação da interoperabilidade COM 20–21 erros do compilador e em tempo de compilação 440–441 conversões entre tipos CLR e dinâmicos 435– 436 criação de locais de chamada 436–438 expressões avaliadas dinamicamente não do tipo dinâmico 436 visão geral 434–435 preservando o comportamento do compilador em tempo de execução 438–440 usando como operador 436 definido 39 Fichários DLR 432 caches 433– 434 locais de chamada 431–432 visão geral 429–431 receptores 432 regras 433–434

- digitação dinâmica (*continuação*)  
 palavra-chave dinâmica  
 visão geral 413–414  
 usando 414–416  
 interoperabilidade de linguagem dinâmica  
 incorporação em C# visão geral 420 422–423  
 armazenando e recuperando informações de ScriptScope 420–422  
 Suporte ao membro da classe DynamicObject de 450–452  
 substituindo o método GetDynamicMemberNames 454–455 substituindo os métodos TryXXX 452–454 visão geral 448–450
- Classe ExpandoObject  
 criando árvore DOM usando visão geral 445–448 444  
 configurando e recuperando propriedades 444–445  
 Interface IDynamicMetaObjectProvider 455–459 em C# 4  
 54–55 visão geral  
 411–412 restrições em  
 delegar restrições de conversão 442–443 métodos de extensão não resolvidos dinamicamente  
 441–442 métodos estáticos  
 443 restrições de parâmetro de tipo 443–444 usos para  
 digitação duck 426–427  
 interoperabilidade de linguagem dinâmica 419  
 inferência de tipo em tempo de execução 424–425  
 operadores genéricos 425–426  
 despacho múltiplo 427–429 visão geral 412–413 vs.  
 digitação estática (C# 1) 39–40  
 Atributo Dinâmico 413, 443  
 Classe DynamicMetaObject 454  
 Classe DynamicMethod 430  
 Suporte ao membro da classe DynamicObject de 450–452  
 substituindo o método GetDynamicMemberNames 454–455 substituindo os métodos TryXXX 452–454 visão geral 448–450
- E**
- avaliação ansiosa 273  
 Projeto Edulinq 527  
*Java efetivo, 2ª edição* 82 eficiência de chamadas de método 274 operadores de elemento (LINQ) 531–532 Método ElementAfterSelf 350 Operador ElementAt (LINQ) 531 Operador ElementAtOrDefault 531 Método dos elementos 350–351 Método ElementsBeforeSelf 350 Propriedade ElementType 337 Incorporar propriedade de tipos de interoperabilidade 417 incorporando linguagem em C# 420 Operador vazio (LINQ) 533 Métodos EndXXX 78 Método de enfileiramento 550 classes de entidade 330–331 Estrutura de Entidade 329 Classe Enum 72 e restrições de tipo de conversão 74 como tipos de valor 45 Operadores de igualdade de classe enumerável 271–273 e tipos anuláveis 119 LINQ532 \_ A classe EqualityComparer 80 é igual à palavra-chave 308 Método igual e código de repetição 129 para tipos anônimos 228 Estrutura <T> anulável 113 Classe anulável 114 substituindo 81 erros, no código do livro 27 ordem de avaliação dos argumentos nomeados 381–382 manipuladores de eventos definidos 36 log em 240–241 Classe EventArgs 138 Delegado EventHandler 135 Exceto o operador (LINQ) 538 exceções por ações na lista de invocação 36 para operações assíncronas manipulação de cancelamento 488–490 desembrulhamento de exceções 482–485 exceções de empacotamento ao lançar 485–488 Método de execução 337, 343, 420 Método ExecutInContext 512 modelo de execução de operações assíncronas 468–470 Classe ExecutionContext 515 Classe ExpandoObject criando árvore DOM usando 445–448 visão geral 444 configuração e recuperação de propriedades 444–445 conversão explícita 110 implementação explícita de interface 43 junções explícitas em LINQ to SQL 334–336 digitação explícita versus digitação implícita (C# 1) 40 variáveis de intervalo digitadas explicitamente 298–300

Classe de expressão 242–243, 430  
 Propriedade de expressão 337  
 árvores de expressão  
   e LINQ 248–249  
   compilando em delegados 243–244  
   convertendo expressões lambda em 244–248 criando  
   242–243 otimizando  
   DLR 250 visão geral 241–  
   242 referências à prova  
   de refatoração para membros 250 usando com reflexão  
   251 Enumeração ExpressionType  
 242 entendendo LINQ para objetos 364–367  
 Extensible Application Markup Language.

#### *Consulte*

Linguagem de marcação extensível XAML. *Consulte*  
 métodos de extensão  
   XML e interfaces fluentes 280–282  
   chamada com argumentos dinâmicos 441  
   definidos 11  
   descoberta pelo compilador 268–269 em C#  
   3 54 prós e  
   contras da sintaxe 282–283 para  
   265–271  
 Atributo ExtensionAttribute 268, 275  
 Extensões classe 351  
 aliases externos 196–197

## F

F# 4, 411  
 eventos semelhantes  
   a campos  
   definidos 37 em C# 4  
 406–407 campos 84–85,  
 217, 227 arquivos, iterando nas linhas em 173–176  
 Filtragem de classe 264 do  
 FileStream  
   Operadores LINQ 538  
   consultas em observáveis 360–361  
 instruções finalmente 174, 405, 481, 500  
   aguardam expressões em 474  
   usando instruções de rendimento 168–170  
 Método EncontrarTudo 238  
 Primeiro operador (LINQ) 531  
 Operador FirstOrDefault (LINQ) 531 variável de  
 tipo fixo 254  
 Atributo FixedBuffer 199 buffers  
 de tamanho fixo em código inseguro 199–201  
 nívelamento  
   de operadores de consulta nivelados 351–352  
   consultas em observáveis 362  
 interfaces fluentes e métodos de extensão 280–282 para declaração  
   e variáveis capturadas 157

e instânciação de variável 154 digitação  
 implícita com 214 instrução  
 foreach 3, 160, 170, 174, 520 e variáveis capturadas  
   157 e instânciação de variável 154  
   variáveis capturadas em 520 digitação  
   implícita com 214 em C# 5 154  
  
 iterando genéricos 87–90  
 parâmetros formais 372  
 Fowler, Martin 280  
 bibliotecas de estrutura, plataforma .NET 24–25  
 assemblies amigos 201–202 da  
   cláusula (LINQ) 335 ordem de  
   296  
   usando múltiplos para junções cruzadas 314–317  
 Func <...> delegado 175, 234–235, 399 ponteiro  
 de função 30 funções  
 anônimo  
   assíncrono 490–492 visão geral assíncrona  
   465 usando 465–  
   468

## G

---

Arquivos .cs  
 186 coleta de lixo e  
   variáveis capturadas 152 e  
   delegados 32  
 operadores de geração (LINQ) 532–533 coleções  
 genéricas BindingList<T>  
   545–546 Collection<T> 545–546  
   coleções simultâneas (.NET 4)  
   550–552 Dicionário < TKey, TValue > 546–547  
   Namespace genérico 540–542 HashSet<T>  
   548–549 KeyedCollection< TKey,  
   TItem > 545–546  
   LinkedList< T > 544–545 List< T > 542–543  
   ObservableCollection< T > 545–  
   546 Queue< T > 550  
   interfaces somente leitura (.NET 4.5) 552–553  
   ReadOnlyCollection< T >  
   546 ReadOnlyDictionary< TKey, TValue > 548  
   ReadOnlyObservableCollection< T >  
   546 SortedDictionary< TKey, TValue > 547–548  
   SortedList< TKey, TValue > 547 –548 SortedSet< T >  
   (.NET 4) 549 Pilha< T > 550

interfaces de comparação genéricas 80 tipo  
 de delegado genérico 68 métodos  
 genéricos 70 e  
   contravariância 99

- métodos genéricos (*continuação*)
  - definidos 64
    - implementação em classes não genéricas 69 inferência de tipo ao chamar 252–253
  - Namespace genérico 540–542
  - operadores genéricos e tipagem dinâmica 425–426
  - interface de tipo genérico 66
  - tipos genéricos, em IL 227
  - variância genérica
    - contravariância 44, 138, 395
    - covariância 44, 138, 395
    - invariância 395–396
    - visão geral 394
    - restrições sobre
      - alterações importantes 403 delegados multicast 403–404 devem ser explícitos 403 parâmetros de saída 402–403 conversões de referência 402 parâmetros de tipo em classes 402 covariância e contravariância simultâneas 399–401
    - usando em delegados 399
    - usando em interfaces
      - contravariância 398–399
      - covariância 397–398
      - modificador in 396–397
      - modificador externo 396–397 visão geral 396–399
    - variação genérica Java 404–405
  - genéricos
    - evitando bugs 61
    - definidos 64
    - dicionário genérico 62–63
    - métodos genéricos 67–70, 76–77
    - tipos genéricos e parâmetros de tipo 64–67
      - implementando
        - expressões de valor padrão 77–79
        - comparações diretas 79–81
        - representando par de valores 81–83
      - importância de 60–61
      - iteração 87–90
    - Compilador JIT, tratamento de 85–87
    - limitações de
      - comparação com modelos C++ 101–102
      - comparação com genéricos Java 103–104
      - falta de propriedades genéricas, indexadores e outros tipos de membros 101
      - falta de variação genérica 94–99
      - falta de restrições de operador ou restrição numérica 99–100
    - convenções de nomenclatura para parâmetros de tipo 67 sobrecarga 66 pronúncia 66
  - lendo declarações genéricas 67–70
  - reflexão e
- refletindo métodos genéricos 93–94
- System.Type, métodos e propriedades de 92
- typeof, usando com tipos genéricos 90–91
- campos estáticos e construtores estáticos 84–85 restrições de
  - tipo combinando restrições 75–76
  - restrições de tipo de construtor 72–73
  - restrições de tipo de conversão 73–75
  - restrições de tipo de referência 71–72
  - restrições de tipo de valor 72
  - vs. modelos C++ 101–102
- Consulte também* variação
- genérica 101 Método GetAwaiter 475 Função GetConsoleScreenBufferEx 200 Método GetDynamicMemberNames 448, 454–455
- Método GetEnumerator 87, 160–161, 163, 175, 337, 343
- Método GetGenericArguments 92
- Método GetGenericTypeDefinition 92
- Método GetHashCode 81, 110, 228
- Método GetInstanceRestriction 458
- Método GetInvocationList 36
- Método GetKeyForItem 545
- Método GetMetaObject 432, 449
- Método GetMethods 93
- Método GetResult 476, 501
- Propriedades getter/setter 192–193
- Método GetType 92
- Método GetTypeRestriction 459
- Método GetUnderlyingType 114
- Método GetValueOrDefault 110
- Método GetVariable 421
- Método GetViewBetween 549 alia de namespace global s 195–196 vá para as declarações 499 Groovy 3, 281 junções de grupo (LINQ) 311–314 grupo...por cláusula (LINQ) 318–321 Operador GroupBy (LINQ) 279, 319, 534
- agrupamento
  - no grupo LINQ...por cláusula 318–321 continuações de consulta 321–323 Operadores LINQ 533–534 consultas em observáveis 361–362 Operador GroupJoin (LINQ) 535 Biblioteca GTK# 25

## H

- 
- códigos hash, calculando 82
  - colisão hash 547

HashSet<T> classe 542, 548–549  
 Classe hashtable 42, 127  
 Propriedade HasValue 109  
 memória heap 48  
 Hejlsberg, Anders 492  
 funções de ordem superior 238  
 Linguagem de hospedagem  
**HostExecutionContext** classe 515 em C# 420  
**Classe HttpClient** 466

---

**Interface IAsyncAction** 516  
**Interface IAsyncActionWithProgress** 516  
**Interface IAsyncInfo** 516  
**Interface IAsyncOperation** 516  
**Interface IAsyncOperationWithProgress** 516  
**Interface IAsyncResult** 463  
**Interface IAsyncStateMachine** 497  
**Interface IAwaiter** 501  
**Interface IBase** 269  
**Interface IClonável** 43  
**Interface ICollection** 364, 428  
**Interface IComparável** 9, 78, 80  
**Interface IComparer** 9, 80, 97, 145, 365, 396, 398, 401

**Interface ICriticalNotifyCompletion** 501  
**Interface derivada de ID** 269  
**Interface do dicionário** 541  
**Interface descartável** 225  
**Interface IDynamicMetaObjectProvider** 432, 444, 455–459  
**Interfaces IEnumerable, IEnumerable<T>** 66, 87, 174, 271, 337, 396, 401, 428 e inicializadores  
 de coleção 221 e contravariância 97  
 e covariância 96

**IEnumerator, IEnumerator<T>** interfaces 159, 164, 540

**Interface IEqualityComparer** 80, 365  
**Interface IEquatable** 79–80  
**Interface IGroupedObservável** 361  
**IL** (linguagem intermediária) 24, 227 ferramenta  
*ildasm* 118, 143, 436  
**IList, IList<T>** interfaces 263, 364, 404, 541  
**ILSpy** 118  
 execução imediata 289  
 imutabilidade e  
   argumentos nomeados 383–384 e  
   parâmetros opcionais 383–384 definidos  
   110  
   tipos imutáveis 383 de  
   delegados 35  
 conversão implícita 110  
 junções implícitas 336

digitação implícita versus digitação explícita  
 40 matrizes digitadas implicitamente  
 224–225 variáveis locais digitadas  
   implicitamente práticas  
   recomendadas  
   para 215–  
   216 definidas 17  
   em C# 3 53 visão geral 211 prós  
   e contras de 214–215  
 restrições em 213–214 usando a palavra-chave var 211–213  
 no modificador 396–397, 403  
**Método InCompletionOrder** 512 indexadores  
 101 , 378, 390 Método  
**IndexOf** 541 indireção 35  
**Método**  
**InDocumentOrder** 351 operador *infoof*  
 250 herança e métodos  
 sobrecarregados 385 junções internas em LINQ 307–  
 311 em LINQ  
   to Objects 310 vs.  
  
 palavra-chave interna 308  
**Interface INotifyCompletion** 476, 501, 515–516  
**Interface INotifyPropertyChanged** 523–524 consultas  
 em processo 17–18  
**Método InsertAt** 541  
**Método InsertItem** 545  
 instanciação de variáveis 153 tipos  
 de interface como  
   tipos de referência 45  
   especificando múltiplos para restrições de tipo  
   de conversão 74  
**interfaces**  
   variância genérica em  
     contravariância 398–399  
     covariância 397–398  
     modificador de entrada  
     396–397 modificador de  
     saída 396–397 visão geral 396–399  
**interfaces somente leitura (.NET 4.5)** 552–553  
**Método intercalado** 508  
**Classe intertravada** 210  
**Linguagem intermediária. Consulte**  
 a palavra-chave interna  
**IL** 185 atributo *InternalsVisibleTo*  
   e assemblies assinados 203–204  
   assemblies amigos para 201–202  
   usos para  
**interoperabilidade** 202–203  
**COM**  
   e digitação dinâmica 417–419  
   chamando indexadores nomeados 390–  
   391 vinculando PIAs 391–  
   394 passando argumentos por valor 389–390

- 
- interoperabilidade, COM (*continuação*)  
 Automação de palavras em C# 3 387–388  
 Automação de palavras em C# 4 388–389
- incorporação de  
 linguagens dinâmicas no  
 C# 420 visão geral  
 422–423 armazenando e recuperando informações de  
 ScriptScope 420–422
- Interseccione o operador 538  
 na cláusula (LINQ) 311–314, 322
- InvalidOperationException 50, 421
- InvalidOperationException 109, 119, 550 invariância  
 395–396 invariante 94
- listas de  
 invocação definidas  
 35 exceções  
 lançadas por ações em 36
- Invocar método 470
- Interface I Observable, I Observable<T> 358–359
- IOException 486
- Interface I OrderedEnumerable<T> 304
- Interface I ProducerConsumerCollection<T> 550–551
- Interface I Progress 505
- Interface I Queryable<T> 271, 337, 339–342
- Interface I QueryProvider<T> 338, 340–341
- Interface I ReadOnlyCollection<T> 552
- Interface I ReadOnlyDictionary<T> 552
- Interface I ReadOnlyList<T> 552
- Python de Ferro em Ação* 419
- FerroPython 433
- IronRuby 433 é  
 operador 123, 403
- Propriedade Is Completada 476, 500
- Interface I Set 542
- Propriedade Is GenericMethod 93
- Propriedade Is GenericType 91–92
- Propriedade Is GenericTypeDefinition 92
- Método Is Interned 249
- Método Is Null 270
- Método Is Null Or Empty 270
- Interface de armazenamento 395
- Interface I StructuralComparable 558
- Interface I StructuralEquatable 558
- Interface I Task 179
- iteradores  
 manuseio correto em extensões LINQ 365  
 criando custom 160–163  
 usando com CCR 178–180  
 usando instruções de rendimento  
 terminando com a instrução yield break 167–168 fluxo  
 de código para 165–167 visão  
 geral 163–165  
 peculiaridades para 170–  
 171 usando blocos finalmente com 168–170
- J.
- 
- Variação genérica Java em 404–405  
 genéricos 103–104
- Java Generics FAQs 404
- Compilador JIT 85–87
- operador de junção (LINQ) 307–311
- junções em  
 LINQ junções cruzadas usando múltiplas cláusulas  
 from 314–  
 317 junções de grupo com join...into cláusulas 311 –314  
 junções internas usando a cláusula de junção  
 307–311 em LINQ to  
 SQL junções explícitas 334–  
 336 junções implícitas  
 336 operadores LINQ 534
- Json.NET 417
- JustDecompile 118
- K
- 
- perfil do kernel 25
- Modificador de chave (VB) 228
- Propriedade chave 318
- KeyedCollection< TKey, TItem > classe 545–546
- KeyNotFoundException 541
- Evento de pressionamento de tecla 135
- Classe KeyPressEventArgs 138
- KeyPressEventHandler 140
- Knuth, Donald 117
- eu
- 
- cálculo lambda 233
- expressões lambda e  
 tipos dinâmicos 442 e LINQ  
 52 como  
 delegados  
 Func<...> tipos delegados 234–235 visão  
 geral 234
- verificação do corpo de 257
- conversão para árvores de expressão 244–248
- criação 235–236 listas
- de parâmetros digitados implicitamente 236
- instanciação de tipos delegados usando 52
- atalho de parâmetro único 237–238 usando  
 expressão única como corpo 236 Método  
 Lambda 248 Classe
- LambdaExpression 243, 245 Consulta  
 integrada à linguagem. Consulte LINQ Último  
 operador (LINQ) 531 avaliação  
 preguiçosa 273, 527

- cláusula `let` (LINQ) 326 em
  - LINQ to SQL Visão geral de 333–334 305–306 operadores
- levantados definidos
  - 119 exemplos
    - de 120
- `LinkedList<T>` classe 544–545
- `LinkedListNode<T>` classe 544
- vinculando PIAs 391–394
- LINQ (Consulta Integrada à Linguagem)
  - árvores de expressão 248–249 conceitos execução
    - adiada 288–289 sequências 286–288
  - operadores padrão 290–291 expressões de consulta
- degeneradas 301–302 extensão 364
- Agrupamento de classe
  - 351 de extensões (LINQ to XML)
    - `group...by` cláusula 318–321
    - consultas contínuas 321–323
  - detalhes específicos de implementação para operadores 290
- junções junções cruzadas usando múltiplas cláusulas
  - `from` 314–
- 317 junções de grupo com `join...into` cláusulas 311–314
  - junções internas usando a cláusula `join` 307–311
- deixe a cláusula 305–306
- agregação
  - de operadores 527–528
  - concatenação 528
  - conversão 529–531
  - elemento 531–532
  - igualdade 532
  - filtragem 538
  - geração 532–533
  - agrupamento 533–534
- junção 534
  - particionamento 535
  - projeção 536–537
  - quantificadores 537
  - baseado em conjunto 538–539
- classificação 539 cláusula `orderby` 302–304
- LINQ paralelo
  - Método `AsParallel` 355–356
    - mantendo o pedido em 356–357
    - Classe `ParallelEnumerable` 354–356
    - Classe `ParallelQuery` 354–356
  - expressões de consulta 286–326 e traduções do compilador 293–296
  - Operador de conversão 298–300 em comparação com a notação de ponto 324–326 variáveis de intervalo digitadas explicitamente 298–300
  - Operador `OfType` 298–300
- variáveis de intervalo 296–298
- cláusula de seleção 293
- identificadores transparentes 306–307 onde cláusula 300–301
- LINQ to Objects 248
  - estendendo
    - a verificação de argumentos
    - 364 eliminando iteradores 365
    - documentação 365 iterando
    - uma vez sempre que possível 365
    - otimização 364–365 visão geral 364 método
    - de extensão de elemento aleatório 365–367 suportando comparações personalizadas 365 testes de unidade 364
  - junções internas para 310
  - Veja também LINQ Paralelo
- LINQ para Rx
  - Interface `IObservable<T>` 358–359
  - Interface `IObserver<T>` 358–359 visão geral 357–358 consultando
  - observáveis filtragem 360–361 nívelamento 362 agrupamento 361–362 limitações de 362–363
  - O método de intervalo 360 usa para 363
- LINQ to SQL 248 criando
  - esquema de banco de dados 330
  - criando classes de entidade 330–331
  - consultas
    - junções explícitas 334–336
    - junções implícitas 336
    - visão geral 332–333
    - usando traduções da cláusula
  - let 333–334 realizadas por
    - Interface `IQueryable<T>` 337–341
    - Interface `IQueryProvider<T>` 338–341
    - Métodos de extensão consultáveis 341–342
- Construtores LINQ
  - to XML em classes principais 347–349 em operadores de
  - consulta nivelados 345–346 Visão geral de 351–352 352–353
  - consultas em nós únicos 349–351
- LINQPad 26, 271
- Lippert, Eric 32, 501
- List, `List<T>` classe 67, 70, 86, 238–240, 542–543
- instruções de bloqueio
  - aguardar expressões em 474 em C# 4 405–406
- Log do método 241
- usando
  - expressões lambda 240–241 com atributos de informações do chamador 522–523

Classe LogicalHttpContext 515  
Operador LongCount (LINQ) 528

**M**

valor mágico 107–108  
Método principal 143  
Método MakeGenericMethod 93  
Método MakeGenericType 92  
Enorme 417  
Aula de matemática 99, 190  
Operador máximo de  
528 membros, de tipos anônimos 227–228  
Classe MemoryStream 86, 264  
metaprogramação 102  
conversões de  
    grupo de métodos  
    13 definidas 136  
Classe MethodInfo 93, 250  
MethodInvoker delegado 137, 140, 149 métodos assíncronos  
    aguardam  
        expressões em 474–477 declarando  
        472 modelando  
        471 tipos de  
        retorno de 473–474, 481–482  
    método de encadeamento chama 273–  
    275 descoberta para ação da instância delegada (C# 1)  
        31–32  
    grupo de métodos genérico  
        67–70, 93–94, conversão para tipo delegado  
            136–137  
    argumentos de tipo para 64  
    Consulte também métodos  
anônimos Micro Framework  
561 Microsoft Office. Consulte interoperabilidade COM  
Microsoft Robotics Studio 178 Microsoft  
Word  
    automatizando em C# 3 387–388  
    automatizando em C# 4 388–389  
Pacote Microsoft.Bcl.Async 464 Pacote  
Microsoft.Bcl.Immutable 553 Montagem  
Microsoft.CSharp 430 Montagem  
Microsoft.CSharp.CodeDomProvider 430  
    Montagem  
Microsoft.CSharp.Compiler 430 Microsoft.Office  
Namespace .Interop.Excel 417  
Moleiro 95  
Operador mínimo 528  
operador de sinal de menos (-)  
    e tipos anuláveis 119  
    removendo instâncias delegadas usando 36  
Campo ValorMínimo 107  
Projeto MiscUtil 348

Mono 24–25, 411, 430  
Maisprojeto LINQ 290, 364  
Evento MouseClick 136  
Classe MouseEventArgs 138  
Método MoveNext 171, 288, 337, 497–499, 501  
MulticastDelegate classe 31  
despacho múltiplo 427–429  
multitargeting 555  
mutabilidade  
    e tipos de valor 110 tipos  
    mutáveis 383

**N**

argumentos nomeados  
    e imutabilidade 383–384 ordem  
    de avaliação de 381–382 resolução  
    de sobrecarga 384–386 visão geral  
    378–379 renomeação  
    386 sintaxe para  
    379–380 usando com  
    parâmetros opcionais 382–383 indexadores  
nomeados 390–391 aliases de  
namespace declarando  
    194–195 aliases  
    externos 196 –197 alias de  
    namespace global 195–196 visão geral  
    193–194 qualificador  
    para 195 convenções  
de nomenclatura 67 NaN (não  
um número) 108 .NET 2.0  
556 .NET 3.0  
557 métodos de  
  
extensão .NET 3.5 em  
    o método de encadeamento reúne 273, 275  
    contando bugs atribuídos aos desenvolvedores  
        exemplo 279  
    somando salários exemplo 278–279 visão  
geral dos recursos  
557 .NET 4  
coleções simultâneas  
    BlockingCollection<T> classe 550–551  
    ConcurrentBag<T> classe 551  
    ConcurrentDictionary< TKey, TValue > classe  
        551–552  
    ConcurrentQueue<T> classe 551  
    ConcurrentStack<T> classe 551  
    IProducerConsumerCollection<T> interface  
        550–551 visão geral  
dos recursos 558 ISet  
    Interface <T> 542  
    SortedSet<T> classe 549  
Atributos  
    de informações do chamador .NET 4.5 524–525

- .NET 4.5 (*continuação*)  
 visão geral dos recursos  
 558 interfaces de coleção somente leitura 552–553  
 plataforma .NET  
   Linguagem C# 24  
   definida 25  
   bibliotecas de estrutura 24–25  
   tempo de  
 execução 24 .NET Reflector. *Consulte*  
 Reflector NetworkStream classe  
 264 novo operador 220  
 NGen 24  
 Noda Time project 557  
 Nodes método 350  
 nós, consultando 349–351  
 NodeType propriedade 242  
 not-a-number. *Consulte*  
 NaN NotSupportedException 543, 548  
 NuGet 357  
 operador de coalescência nula ( ?? ) 123–126, 130  
 referência nula  
   e digitação implícita 213  
   chamando o método de extensão em 269–271  
 valores nulos 8, 55, 106 e  
   operador condicional 130 e bancos de  
   dados 107  
   como valor padrão no genérico 78  
   comparando com 79 usando  
   parâmetros opcionais 376–378 Classe anulável 114  
 tipos anuláveis ?  
 modificador 115–  
   116 e operadores 119  
   atribuindo e comparando  
   com nulo 116–118 conversões envolvendo tipos  
   anuláveis 119 definidos 15 em C# 2 55 inteiros 120  
   operador de  
   coalescência  
   nulo 123–126  
   lógica anulável 121–123 operadores  
   envolvendo tipos anuláveis  
   119–121 padrões para representar nulo valores (C # 1)  
   sinalizador booleano extra 108–109 valor mágico 107–  
   108 wrapper de tipo de referência  
   108 System.Nullable<T>  
   boxing e unboxing 112–113  
   igualdade de instâncias  
   113 visão geral 109–112 suporte de  
   classe nula não genérica 114
- tipo de valor anulável, vs. tipo anulável 115  
 Nullable<T> struct 15 e ?  
   modificador 115 boxe e  
   unboxing 112–113 igualdade de  
   instâncias 113
- visão geral 109–112  
 suporte da classe anulável não genérica 114  
 NullReferenceException 79, 112, 269, 271, 415 restrições  
 numéricas 99–100
- 
- ## Ó
- Classe de objeto  
 74 padrão de  
   construtor de inicializadores  
   de objeto 223 coleções constantes 223  
   tipo de exemplo para visão geral  
   216–217 224  
   definindo propriedades 217–219  
   definindo propriedades em objetos incorporados  
   219–220  
   configurando testes de unidade  
 223 tipo de objeto, métodos retornando 457  
 mapeamento objeto-relacional. *Consulte*  
 objetos ORM 48–  
 49 ObservableCollection<T> classe 545–546  
 Operador OfType (LINQ) 97, 298–300, 529  
 Método OnCompleted 359–360, 501, 516  
 Método OnError 359  
 Método OnNext 359–360 tipos  
 abertos 65  
 OperationCanceledException 483, restrições de  
 operadores  
   488 , falta de 99–100 envolvendo  
   tipos anuláveis 119–121
- Agregação LINQ 527–528  
 concatenação 528  
 conversão 529–531  
 elemento 531–532  
 igualdade 532  
 filtragem 538  
 geração 532–533  
 agrupamento 533–534  
 junção 534  
 visão geral 290–291  
 particionamento 535  
 projeção 536–537  
 quantificadores 537  
 classificação baseada  
   em conjunto  
 538–539 classificação 5 39  
 otimização 364–365, 527 parâmetros  
   opcionais 16, 521 e imutabilidade  
   383–384 e versionamento 375–  
   376 declarando resolução  
   de sobrecarga 373–374 384–386  
   renomeando 386  
   restrições em 374–375 usos  
   para 372–373

parâmetros opcionais (*continuação*)  
 usando valores nulos 376–378  
 usando com argumentos nomeados 382–383  
 cláusula orderby (LINQ) 302–304  
 Operador OrderBy (LINQ) 277, 281, 304, 326, 539  
 Operador OrderByDecrescente (LINQ) 277, 304, 539  
 OrderedParallelQuery<TSource> ordenação da classe 355 em Parallel LINQ 356–357  
 Comparação de string OrdinalIgnoreCase 324  
 ORM (mapeamento objeto-relacional) 186  
 modificador de saída 379, 396–397, 402–403  
 parâmetro de saída 148, 189, 236 variável externa 148, 308 resolução de sobrecarga e herança 385 e argumentos nomeados 384–386 e parâmetros opcionais 384–386 visão geral 258–260  
 sobrecarga, tipos genéricos 66

**P**

Par<T1, T2> classe 81–83  
 LINQ paralelo  
 Método AsParallel 355–356  
 mantendo a ordem em 356–357 Classe ParallelEnumerable 354–356 Classe ParallelQuery 354–356 Blog da equipe de programação paralela 504 Classe ParallelEnumerable 354–356 Classe ParallelQuery 354–356 matriz de parâmetros 374 tipo de parâmetro contravariance 43 classe ParameterExpression 246 propriedade parametrizada 390 digitação parametrizada. Consulte o delegado genérico ParameterizedThreadStart 147, 151 parâmetros em expressões lambda digitadas implicitamente 236 atalho de parâmetro único 237–238 opcional 372–407 parâmetros de tipo 64–67 vs. argumentos 372 modificador de parâmetros 374 Método de análise 351 palavra-chave parcial 184, 189 métodos parciais 188–190 tipos parciais e métodos parciais (C# 3) 188–190 criação 184–186 definidos 183–184 usos para 186–187

classes auxiliares parcialmente genéricas 83 operadores de particionamento (LINQ) 535 passagem por referência 48 padrões para operações assíncronas coleta de resultados à medida que chegam 508–511 coleta de resultados em chamada única 507–508 padrão assíncrono baseado em tarefa 504–507  
 Método Peek 550 PIAs (Primary Interop Assemblies)  
 definidos 21 vinculação 391–394 PLINQ (Parallel LINQ). Consulte a postagem do blog MSDN do Parallel LINQ “PLINQ Ordering” 356 interfaces de plug-in 90 operador de sinal de mais (+) e tipos anuláveis 119 combinando instâncias delegadas usando 35 Método pop 550 argumentos posicionais 380 Pós método 470 Coleções de energia 553 diretivas pragma checksum pragmas 198–199 definidos 197 pragmas de aviso 197–198 Delegado predicado 145 restrições primárias 76 Assemblies de interoperabilidade primárias. Consulte PIAs Método PrintCount 427 palavra-chave privada 193 classe ProcessStartInfo 223 inicializadores de projeção 228–229 operadores de projeção (LINQ) 536–537 projeções 293 propriedades implementadas automaticamente 208–211 para tipos anônimos 227 configuração com inicializadores de objeto 217–218 Classe PropertyChangedEventArgs 523 Delegado PropertyChangedEventHandler 523 palavra-chave pública 185 Método push 550 Pitão 3, 21, 420

**P**

operadores quantificadores (LINQ) 537 consultas coleções 12–14 LINQ to SQL junções explícitas 334–336 junções implícitas 336 visão geral 332–333 usando a cláusula let 333–334

consultas (*continuação*)  
 Operadores de  
 consulta nívelados LINQ to XML 351–352  
 em nós únicos 349–351 em  
 filtragem de  
 observáveis 360–361  
 nivelamento 362  
 agrupamento 361–362  
 limitações de 362–363  
 expressões de consulta 17–18  
 XML 18–19  
 continuações de consulta 318, 321–323  
 expressões de  
 consulta  
 definidas 17 da cláusula  
 314–317 fundamentos 293–  
 296 grupo...por cláusula 318–  
 321 grupo...por...na cláusula 321–323  
 cláusula join 307–311  
 join...into cláusula 311–314 let  
 cláusula 305–306  
 ordem de 296  
 cláusula orderby 302–304  
 variáveis de intervalo 296–300  
 cláusula select 293–296  
 select...into cláusula 321–323  
 identificadores transparentes 306–307  
 vs. notação de ponto 324–326  
 cláusula where 300–301  
*Consulte também*  
 provedores de consulta LINQ  
 342–344 Classe consultável 271,  
 341–342 Propriedade QueryProvider  
 337 Queue<T> classe 550

**R**

condição de corrida 488  
 Ragsdale 95  
 Rahien, Ayende 281  
 método de extensão de elemento aleatório 365–367  
 método de intervalo 272, 277, 357, 360, 533  
 variáveis de intervalo  
 292 digitadas explicitamente  
 298–300 visão geral  
 296–298 Pacote de extensões reativas. *Consulte leitura,*  
*avaliação e loop de impressão do pacote*  
 Rx. *Consulte o método REPL*  
 ReadAllText 556 Classe  
 ReaderWriterLockSlim 557  
 Método ReadFully 265 Método  
 ReadLines 173 interfaces de coleção somente  
 leitura  
 (.NET 4.5) 552–553 ReadOnlyCollection<T>  
 classe 223, 546 ReadOnlyDictionary< TKey, TValue > 548

ReadOnlyObservableCollection<T> classe 546  
 ReadToEndAsync 486  
 receptores 432  
 parâmetros de referência 148, 236, 379,  
 402 conversão de referência 138, 299, 396, 402  
 tipos de referência  
 e tipos de valor, C# 1  
 boxing e unboxing 49–50  
 fundamentos 46–47 mitos  
 sobre 47–49 novos  
 recursos 55–56 restrições  
 71–72 definido 45  
 wrapper de  
 tipo de referência 108 reflexão e  
 tipagem  
 dinâmica 93 genéricos e  
 refletindo  
 métodos genéricos 93–94 System.Type,  
 métodos e propriedades de 92 typeof, usando com  
 tipos genéricos 90–91 usando árvores de  
 expressão com 251  
 Ferramenta refletor 118, 143, 436  
 operadores relacionais 119  
 Remover método 35–36, 541  
 Método RemoveAt 541  
 Método RemoveFirst 545  
 Método RemoveItem 545  
 RemoveLast método 545  
 Método RemoveWhere 549  
 remoção, delegados (C# 1) 35–36 método  
 de reordenação exige eficiência 274  
 Operador de repetição (LINQ) 533  
 REPL (ler, avaliar, imprimir loop) 411  
 Substitua o método 274  
 Método de reinicialização 171  
 declarações de retorno  
 tipos de retorno de métodos assíncronos 473–474,  
 481–482 produzem  
 instruções de retorno 164  
 covariância do tipo de retorno  
 43 Propriedade ReturnType 457  
 Operador reverso (LINQ) 272–273, 288, 324 Ruby 3  
 Método  
 de execução 505, 515, 517 tempo  
 de  
 execução  
 plataforma .NET  
 24 definido 24 RuntimeBinderException  
 415 Pacote Rx (Extensões Reativas) 290, 357, 505

**S**

código seguro 38  
 Método Salvar como 21  
 Esquema 148

- escopo, variáveis capturadas em diferentes 155  
 Classe ScriptEngine 420  
 Classe ScriptHost 420  
 Classe ScriptRuntime 420  
 Classe ScriptScope 420  
 Classe ScriptSource 420  
 restrições secundárias 76  
 segurança e scripts incorporados 423  
 Classe SecurityContext 515  
 SecurityCriticalAttribute 516 cláusula  
 de seleção (LINQ) ordem  
   de 296 visão  
   geral 293  
 Selecionar o operador (LINQ) 276–277, 288, 341, 536  
 Operador SelectMany (LINQ) 351, 362, 536  
 Método de envio 470  
 Operador SequenceEqual (LINQ) 532 sequências  
   em LINQ 286–288 operadores  
   baseados em conjunto 538–539  
 Método SetItem 545  
 Método setResult 499  
 Método SetStateMachine 497–498  
 Método SetVariable 421  
 assemblies assinados 203–204  
 Silverlight 560–561  
 delegados simples, em C# 1  
   criando a instância 32–33  
   declarando o tipo de delegado 31  
   exemplo 33–35  
   encontrando o método para a ação da instância 31–32  
   invocando a instância 33  
 despacho único 427  
 Operador único (LINQ) 531  
 Operador SingleOrDefault (LINQ) 531  
 Ignorar operador (LINQ) 535  
 Operador SkipWhile (LINQ) 535 snippets  
 definiram  
   26  
   apresentando programas completos como 25–26  
 Ferramenta Snippy 26, 186  
 Método de classificação 9, 238  
 SortedDictionary< TKey, TValue > classe 548  
 SortedList< TKey, TValue > classe 547–548  
 SortedSet< T > classe 542, 549  
 classificação e filtragem  
   consulta de coleções 12–14  
   classificação de produtos por nome 9–12  
 operadores de classificação (LINQ)  
 539 SQL (Structured Query Language)  
   LINQ to SQL 19–20 vs.  
   pilha LINQ 18 e  
 operações assíncronas 501–503 e tipos de valor 48  
 Pilha< T > classe 550  
 campo de estado  
 498 máquina de  
   estado ponto de entrada único para  
   estrutura 498–500 do  
 estado 497–498 e iteradores  
 161 instruções 174, 193, 214, 268  
 classes estáticas, em C# 2 190–192  
 construtores estáticos 84–85  
 campos estáticos 84–  
   85 estáticos interfaces  
 100 palavra-chave estática  
 39, 191  
   digitação estática  
   definida 38–39 vs. tipagem dinâmica (C#  
 1) 39–40 Classe Stream  
 264, 269 streaming vs. buffering 273, 288  
 Classe StreamUtil 264  
 literais de string em Python 420 tipo  
 de string 39  
 Classe StringCollection 43  
 coleções  
   fortemente tipadas  
   6–7 definidas 38  
 Stroustrup, Bjarne 102  
 estruturas 45–48  
 comparações estruturais 558  
 Structured Query Language. Consulte o método  
 SQL Subscribe 358 assinando  
 eventos 36 Operador Sum  
   (LINQ) 528 Classe  
 SynchronizationContext 470, 500, 515 Indexador  
 SynonymInfo 390 Namespace  
 System 234 Namespace  
 System.Collections.Concurrent 550 Namespace  
 System.Collections.Generic 540 Namespace System.Linq  
 269 System.Linq  
 Namespace .Expressions 242 Namespace  
 System.Numeric 558 Namespace  
 System.Runtime.CompilerServices 475,  
   521  
 System.Runtime.WindowsRuntime.dll assembly  
   517 Namespace  
 System.Xml.Linq 345
- 
- T**
- Classe TableLayoutPanel 556  
 Pegue o operador (LINQ) 535  
 Operador TakeWhile (LINQ) 535 alvo, da  
 ação delegada 32  
 Classe de tarefa 466–468, 481  
   exceções para tipos de retorno  
   482–485 para métodos assíncronos 473 Task  
 Parallel Library. Consulte o padrão  
 assíncrono baseado em tarefas TPL 464, 504–507

Modelos TaskCanceledException  
483 , testes C++ 101–102 ,  
testes de divisão usando tipos parciais 187  
Classe de leitor de texto 175  
Operador ThenBy (LINQ) 277, 281, 304, 326  
Operador ThenByDescendente (LINQ) 277, 304 esta  
palavra-chave 266 e  
métodos anônimos 143 como variável  
externa 148  
Delegado ThreadStart 147  
Classe ThreadStaticAttribute 84  
Método ThrowIfCancellationRequested 488  
Classe TimeMachine 512  
Estrutura TimeSpan 426  
Classe TimeZoneInfo 557  
tlbimp (ferramenta Type Library Importer) 391  
Operador ToArray (LINQ) 529, 551  
Operador ToDictionary (LINQ) 324, 529  
OperadorToList (LINQ) 325, 529  
Operador ToLookup (LINQ) 529  
Método ToString  
para tipos anônimos 227  
Estrutura anulável<T> 110  
Toub, Estêvão 466, 516  
Método ToXml 445, 450  
TPL (Biblioteca Paralela de Tarefas) 178, 463, 488  
Traduções da biblioteca  
TPL Dataflow 507 em LINQ to SQL  
Interface IQueryble<T> 337–341  
Interface IQueryProvider<T> 338–341  
Métodos de extensão consultáveis 341–342  
usando provedor de consulta customizado 342–  
344 identificadores transparentes 306–307  
Método TrimExcess 542  
propriedade trivial 209  
tabelas  
verdade definidas 121  
para bool? tipo 122  
instruções try 481 e  
instruções de rendimento 164  
bloqueiam em 405  
Método TryAdd 551  
Método TryBinaryOperation 452  
Método TryConvert 452  
Método TryCreateInstance 452  
Método TryDeleteIndex 452  
Método TryDeleteMember 452  
Método TryGetIndex 452  
Método TryGetMember 452–453  
Método TryGetValue 78, 541  
Método TryInvoke 452  
Método TryInvokeMember 452  
Métodos TryParse 78  
Método TrySetIndex 452  
Método TrySetMember 452

Método TryTake 551  
Método TryUnaryOperation 452  
Métodos TryXXX 78  
substituindo 452–454  
análise de inteiro exemplo 127  
Tupla classe 558  
tuplas 81, 128  
Turing, Alan 117  
inferência de tipo em duas fases 254–258  
argumentos de tipo 64  
conversão de tipo 41  
Classe de tipo 90,  
92 restrições de  
tipo combinando restrições 75–76  
restrições de tipo de construtor 72–73  
restrições de tipo de conversão 73–75  
definidas 70–71  
restrições de tipo de referência 71–72  
restrições de tipo de valor 72  
apagamento de tipo  
103 inferência de tipo  
298 em tempo de execução 424–  
425 chamando métodos genéricos 252–  
253 definidos  
71 tipos de retorno de funções anônimas 253–254  
inferência de tipo em duas fases 254–258  
usando com método não genérico 82  
restrição de  
parâmetros de  
tipo 73 definida 64  
convenções de  
nomenclatura genéricas 64–  
67 67 irrestritas, definidas 71  
TypedReference struct 435  
operador typeof 65, 78, 90–91, 443  
  
digitação dinâmica  
20–22 COM e 20–  
22 vs. digitação estática (C# 1) 39–  
40 explícita  
40 implícita  
40 estática 39–  
40 *Veja também* genéricos  
  
você  

---

  
Thread de UI 467  
operadores unários 119  
tipos genéricos não vinculados  
64 conversão  
de unboxing 299  
definidos 50  
Nullable<T> 112–113 tipos  
de valor e tipos de referência, C# 1 49–50  
Projeto Melodia Irrestrita 74

parâmetros de tipo irrestrito 71 tipo  
subjacente 109 variável  
de tipo não fixo 254  
Operador Union (LINQ) 538 testes  
de unidade e  
  initializadores de objetos 223  
  operações assíncronas  
    avançando o tempo programaticamente 511–513  
    executando testes 513–514  
  estendendo LINQ para objetos 364  
Método UnsafeOnCompleted 501, 516 nomes  
indizíveis 143 cancelando  
inscrição em eventos 36  
desembrulhando 110  
usando instruções 4, 131

**V**

Propriedade de valor 109,  
448 restrições de tipo de  
valor 72  
  tipos de valor e  
  mutabilidade 110 e discrepância  
  de padrões 73 boxing e  
  unboxing 49–  
  50 restrições 72  
fundamentos 46–49 valores  
  expressões de valor padrão 77–79  
  representando o par de 81–83  
Classe ValueType 72, 74  
var palavra-chave  
  17 visão geral 211–213  
  vs. palavra-chave dinâmica  
416  
variáveis acessando fora de métodos anônimos  
150 capturando em métodos anônimos  
  fechamentos 148–149  
  examinando o comportamento de 149–  
  151 vida útil estendida de 152–153  
  instâncias de variáveis locais 153–155  
  misturas de variáveis compartilhadas  
    e distintas 155–156  
  propósito de 151  
práticas recomendadas  
  locais digitadas implicitamente  
  para visão geral  
  de 215–216 211 prós e contras  
  de 214–215 restrições em  
  213–214 usando a palavra-chave  
var 211–213 do tipo  
delegado  
  33 variância  
definida 94  
  genérica, falta de suporte para  
  covariância 95 onde contravariância seria útil 97–99  
  onde a covariância seria útil 95–97

VARIANT tipo 212, 412  
vetores 544  
versões  
  e parâmetros opcionais 375–376  
  Linguagem C# 555–556  
  A biblioteca da estrutura  
  Compact Framework 560 apresenta versões  
  principais da estrutura 556–558 554–555  
  O tempo de execução do  
  Micro Framework 561 (CLR) apresenta 559  
  Luz Prateada 560–561  
  WinRT 561–562  
  Veja também versões individuais  
Visual Básico 228, 492  
Métodos de  
  extensão do Visual Studio nas  
  versões 268 554–555  
  especificação no  
  visualizador 27  
246 palavra-chave nula 189, 473

**C**

Método de espera 490  
pragmas de advertência 197–198  
WCF (Comunicação do Windows  
  Fundação) 557  
Classe WebClient 466  
WebException 483  
Método WhenAll 508  
Método WhenAny 508  
cláusula where (LINQ)  
  ordem da visão  
  geral 296 300–301  
onde palavra-chave 71, 75  
Operador Where (LINQ) 288, 538 método  
  de extensão 273–276  
  implementando usando blocos iteradores 176–178  
espaço em branco 220  
janelas 8 516  
Windows CardSpace 557  
Windows Communication Foundation. Consulte  
WCF Windows Presentation Foundation. Veja WPF  
Windows WF (Fundação de Fluxo de Trabalho) 557  
Classe WindowsRuntimeSystemExtensions 517  
Operações assíncronas do WinRT  
  (Windows Runtime) em 464, 516–517 visão geral  
  561–562  
Método WithCancellation 356  
Método WithDegreeOfParallelism 356  
Método WithExecutionMode 356  
Método WithMergeOptions 357  
WPF (Windows Presentation Foundation) 557 envolvendo  
  110

**X**

Xamarin.iOS 24 XAML  
(Extensible Application Markup Language) 186 Classe  
XAttribute 345–347,  
350 Classe XCData 346 Classe XComment  
345 Classe XContainer  
346 Classe XDeclaration 345  
Classe XDocument 346  
Classe XElement 345–347, 349–  
350, 449 XML (Extensible  
Markup Language ) 18–19 Classe XmlElement 345  
Classe XmlReader 373 Classe XName 345–346 Classe  
XNamespace 345–346

Classe XNode 345, 347  
Classe XObject 345  
Classe XText 346

**S**

instruções de  
rendimento terminando com instrução de interrupção de  
rendimento 167–168 fluxo de  
código para 165–167 visão geral 163–165  
peculiaridades para 170–  
171 usando blocos finalmente com 168–170

**Z**

Operador Zip (LINQ) 536

## C#IN DEPTH, Terceira Edição

Jon Skeet



criar um aplicativo empresarial avançado ou apenas lançar um aplicativo Se você for um desenvolvedor .NET usará C# quer esteja construindo expressões lambda, digitação dinâmica, LINQ, blocos iteradores e outros recursos. Mas primeiro você tem que aprender isso em profundidade.

**C# em profundidade, terceira edição** foi completamente revisado para cobrir os novos recursos do C# 5, incluindo as sutilezas de escrever código assíncrono sustentável. Você verá o poder do C# em ação, aprendendo como trabalhar com recursos de alto valor que ficará feliz em ter em seu kit de ferramentas. E você aprenderá a evitar as armadilhas ocultas da programação em C# com a ajuda de explicações claras sobre questões “nos bastidores”.

### O que há dentro

- Atualizado para C# 5
- O novo recurso assíncrono/aguardado
- Como o C# funciona e por quê

Este livro pressupõe que você tenha digerido seu primeiro livro sobre C# e esteja ansioso por mais!

**Jon Skeet** é engenheiro de software sênior no Google e um participante altamente visível de grupos de notícias, grupos de usuários, conferências internacionais e do site Stack Overflow Q&A. Jon passa grande parte do dia programando em Java, mas seu coração pertence ao C#.

Para baixar seu e-book gratuito nos formatos PDF, ePUB e Kindle, os proprietários deste livro devem visitar [manning.com/CSharpinDepthThirdEdition](http://manning.com/CSharpinDepthThirdEdition)

“O definitivo o que, como,  
e por que de C #.”

—Do prefácio de Eric  
Lippert, Coverity

“A melhor fonte para aprender recursos  
da linguagem C#.”

—Andy Kirsch, Venga

“Levei meu conhecimento em C#  
para o próximo nível.”

—Dustin Laine, Colheita de Código

“Um livro obrigatório para todo  
desenvolvedor .NET”  
deveria ler pelo menos uma vez.

—Dror Helper, lugar melhor

“Facilmente o melhor C#  
referência que encontrei.”

—Jon Parish, Datasift

ISBN 13: 978-1-617291-34-0  
ISBN 10: 1-617291-34-X



9 781617 291340



TRIPULAÇÃO

\$ 49,99 / lata \$ 52,99 [INCLUINDO e-BOOK]