

# 上下文与模型映射配置

2019年3月28日 21:32

## 配置DbContext上下文

配置 DbContext 以使用特定的 EF Core 提供程序和可选行为来连接到数据库的基本模式。

### 配置上下文选项

DbContextOptionsBuilder -> DbContextOptions -> DbContext

### 构造函数参数方式

```
public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        : base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}
```

*注: DbContextOptions 可以使用非泛型版本, 但不推荐。*

```
var optionsBuilder = new DbContextOptionsBuilder<BloggingContext>();
optionsBuilder.UseSqlite("Data Source=blog.db");

using (var context = new BloggingContext(optionsBuilder.Options))
{
    // do stuff
}
```

### 重写 OnConfiguring 方法

```
public class BloggingContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blog.db");
    }
}
```

```

}

using (var context = new BloggingContext())
{
    // do stuff
}

```

注: 无法进行单元测试

## 依赖注入方式

```

public class BloggingContext : DbContext
{
    public BloggingContext(DbContextOptions<BloggingContext> options)
        :base(options)
    { }

    public DbSet<Blog> Blogs { get; set; }
}

public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<BloggingContext>(options =>
options.UseSqlite("Data Source=blog.db"));
}

public class MyController
{
    private readonly BloggingContext _context;

    public MyController(BloggingContext context)
    {
        _context = context;
    }
}

using (var context = serviceProvider.GetService<BloggingContext>())
{
    // do stuff
}

var options = serviceProvider.GetService<DbContextOptions<BloggingContext>>
();

```

## 测试数据提供程序

### SQLite 具有内存中模式

```
// In-memory database only exists while the connection is open
var connection = new SqlConnection("DataSource=:memory:");
connection.Open();
var options = new DbContextOptionsBuilder<BloggngContext>()
                .UseSqlite(connection)
                .Options;
```

### InMemory 数据库进行测试

```
var options = new DbContextOptionsBuilder<BloggngContext>()
                .UseInMemoryDatabase(databaseName:
"Add_writes_to_database")
                .Options;
```

## 创建并配置模型

- 重写 OnModelCreating 使用 Fluent API 配置模型

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(b => b.Url)
            .IsRequired();
    }
}
```

- 使用 Data Annotations 数据注释来配置模型

```
public class Blog
{
    public int BlogId { get; set; }

    [Required]
    public string Url { get; set; }
}
```

## 忽略不必要的类型（类）

### 约定

首先：DbSet类型中的 Public 公共属性将包含在最终的模型之中。

此外：OnModelCreating 方法提及到的类型将包括在模型之中。

最后：通过递归发现的导航属性也包括在模型中。

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AuditEntry>();
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public Blog Blog { get; set; }
}

public class AuditEntry
{
    public int AuditEntryId { get; set; }
    public string Username { get; set; }
    public string Action { get; set; }
}
```

以上代码：blog 在 DbSet 中公开，Post 作为 Blog 的导航属性被发现，还有在 OnModelCreating 中配置的类型。

## 排除类型 (Data Annotations)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogMetadata Metadata { get; set; }
}

[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDatabase { get; set; }
}
```

## 排除类型 (Fluent API)

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<BlogMetadata>();
    }
}
```

## 忽略不必要的属性

### 约定

按照约定，模型所含的那些公共属性都拥有一个 getter 和一个 setter 访问器。

## 排除属性 (Data Annotations)

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
```

```
}
```

## 排除属性 (Fluent API)

```
class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Ignore(b => b.LoadedFromDatabase);
    }
}
```

## 数据库主键

主键是数据库每行数据的唯一标识符。

### 约定

名为 Id 或 <type name>Id 的属性会配置为实体的主键。

### Data Annotations

```
[Key]
public string LicensePlate { get; set; }
```

### Fluent API

```
modelBuilder.Entity<Car>().HasKey(c => c.LicensePlate);
```

### 复合主键

无法使用约定和 Data Annotations 设置复合键，只能使用 Fluent API 设置。

```
modelBuilder.Entity<Car>().HasKey(c => new { c.State, c.LicensePlate });
```