**Entitiy Framework Core中使用ChangeTracker持久化实体修改历史**

# 背景介绍

在我们的日常开发中，有时候需要记录数据库表中值的变化, 这时候我们通常会使用触发器或者使用关系型数据库中临时表(Temporal Table)或数据变更捕获(Change Data Capture)特性来记录数据库表中字段的值变化。原文的作者Gérald Barré讲解了如何使用Entity Freamwork Core上下文中的ChangeTracker来获取并保存实体的变化记录。

原文链接 [Entity Framework Core: History / Audit table](https://www.meziantou.net/2017/08/14/entity-framework-core-history-audit-table)

# ChangeTracker

ChangeTracker是Entity Framework Core记录实体变更的核心对象(这一点和以前版本的Entity Framework一致)。当你使用Entity Framework Core进行获取实体对象、添加实体对象、删除实体对象、更新实体对象、附加实体对象等操作时，ChangeTracker都会记录下来对应的实体引用和对应的实体状态。  
我们可以通过<code>ChangeTracker.Entries()</code>方法, 获取到当前上下文中使用的所有实体对象, 以及每个实体对象的状态属性State。

Entity Framework Core中可用的实体状态属性有以下几种

* Detached
* Unchanged
* Deleted
* Modified
* Added

所以如果我们要记录实体的变更，只需要从ChangeTracker中取出所有Added, Deleted, Modified状态的实体, 并将其记录到一个日志表中即可。

# 我们的目标

我们以下面这个例子为例。  
当前我们有一个顾客表Customer和一个日志表Audit, 其对应的实体对象及Entity Framework上下文如下：

#### Audit.cs

[Table("Audit")]

public class Audit

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string TableName { get; set; }

public DateTime DateTime { get; set; }

public string KeyValues { get; set; }

public string OldValues { get; set; }

public string NewValues { get; set; }

}

#### Customer.cs

[Table("Customer")]

public class Customer

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

}

#### SampleContext.cs

public class SampleContext : DbContext

{

public SampleContext()

{

}

public DbSet<Customer> Customers { get; set; }

public DbSet<Audit> Audits { get; set; }

}

我们希望当执行以下代码之后, 在Audit表中产生如下数据

class Program

{

static void Main(string[] args)

{

using (var context = new SampleContext())

{

// Insert a row

var customer = new Customer();

customer.FirstName = "John";

customer.LastName = "doe";

context.Customers.Add(customer);

context.SaveChangesAsync().Wait();

// Update the first customer

customer.LastName = "Doe";

context.SaveChangesAsync().Wait();

// Delete the customer

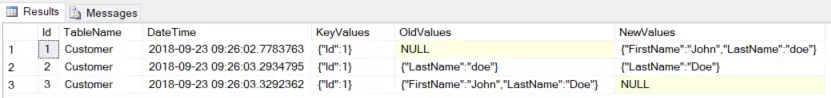
context.Customers.Remove(customer);

context.SaveChangesAsync().Wait();

}

}

}



# 实现步骤

## 复写上下文SaveChangeAsync方法

首先我们添加一个AuditEntry类, 来生成变更记录。

public class AuditEntry

{

public AuditEntry(EntityEntry entry)

{

Entry = entry;

}

public EntityEntry Entry { get; }

public string TableName { get; set; }

public Dictionary<string, object> KeyValues { get; } = new Dictionary<string, object>();

public Dictionary<string, object> OldValues { get; } = new Dictionary<string, object>();

public Dictionary<string, object> NewValues { get; } = new Dictionary<string, object>();

public List<PropertyEntry> TemporaryProperties { get; } = new List<PropertyEntry>();

public bool HasTemporaryProperties => TemporaryProperties.Any();

public Audit ToAudit()

{

var audit = new Audit();

audit.TableName = TableName;

audit.DateTime = DateTime.UtcNow;

audit.KeyValues = JsonConvert.SerializeObject(KeyValues);

audit.OldValues = OldValues.Count == 0 ? null : JsonConvert.SerializeObject(OldValues);

audit.NewValues = NewValues.Count == 0 ? null : JsonConvert.SerializeObject(NewValues);

return audit;

}

}

* Entry属性表示变更的实体
* TableName属性表示实体对应的数据库表名
* KeyValues属性表示所有的主键值
* OldValues属性表示当前实体所有变更属性的原始值
* NewValues属性表示当前实体所有变更属性的新值
* TemporaryProperties属性表示当前实体所有由数据库生成的属性集合

然后我们打开SampleContext.cs, 复写方法SaveChangeAsync代码如下。

public override async Task<int> SaveChangesAsync(bool acceptAllChangesOnSuccess, CancellationToken cancellationToken = default(CancellationToken))

{

var auditEntries = OnBeforeSaveChanges();

var result = await base.SaveChangesAsync(acceptAllChangesOnSuccess, cancellationToken);

await OnAfterSaveChanges(auditEntries);

return result;

}

private List<AuditEntry> OnBeforeSaveChanges()

{

throw new NotImplementedException();

}

private Task OnAfterSaveChanges(List<AuditEntry> auditEntries)

{

throw new NotImplementedException();

}

* 这里我们添加了2个方法<code>OnBeforeSaveChange()</code>和<code>OnAfterSaveChanges</code>。
* <code>OnBeforeSaveChanges</code>是用来获取所有需要记录的实体
* <code>OnAfterSaveChanges</code>是为了获得实体中数据库生成列的新值(例如自增列, 计算列)并持久化变更记录, 这一步必须放置在调用父类<code>SaveChangesAsync</code>之后，因为只有持久化之后，才能获取自增列和计算列的新值。
* 在<code>OnBeforeSaveChange</code>方法之后，<code>OnAfterSaveChanges</code>方法之前, 我们调用父类的<code>SaveChangesAsync</code>来保存实体变更。

然后我们来修改<code>OnBeforeSaveChanges</code>方法, 代码如下

private List<AuditEntry> OnBeforeSaveChanges()

{

ChangeTracker.DetectChanges();

var auditEntries = new List<AuditEntry>();

foreach (var entry in ChangeTracker.Entries())

{

if (entry.Entity is Audit || entry.State == EntityState.Detached || entry.State == EntityState.Unchanged)

continue;

var auditEntry = new AuditEntry(entry);

auditEntry.TableName = entry.Metadata.Relational().TableName;

auditEntries.Add(auditEntry);

foreach (var property in entry.Properties)

{

if (property.IsTemporary)

{

// value will be generated by the database, get the value after saving

auditEntry.TemporaryProperties.Add(property);

continue;

}

string propertyName = property.Metadata.Name;

if (property.Metadata.IsPrimaryKey())

{

auditEntry.KeyValues[propertyName] = property.CurrentValue;

continue;

}

switch (entry.State)

{

case EntityState.Added:

auditEntry.NewValues[propertyName] = property.CurrentValue;

break;

case EntityState.Deleted:

auditEntry.OldValues[propertyName] = property.OriginalValue;

break;

case EntityState.Modified:

if (property.IsModified)

{

auditEntry.OldValues[propertyName] = property.OriginalValue;

auditEntry.NewValues[propertyName] = property.CurrentValue;

}

break;

}

}

}

}

* <code>ChangeTracker.DetectChanges()</code>是强制上下文再做一次变更检查
* 由于Audit表也在ChangeTracker的管理中, 所以在<code>OnBeforeSaveChanges</code>方法中，我们需要将Audit表的实体排除掉，否则会出现死循环
* 这里我们只需要操作所有Added, Modified, Deleted状态的实体，所以Detached和Unchanged状态的实体需要排除掉
* ChangeTracker中记录的每个实体都有一个<code>Properties</code>集合，里面记录的每个实体所有属性的状态, 如果某个属性被修改了，则该属性的<code>IsModified</code>是true.
* 实体属性Property对象中的<code>IsTemporary</code>属性表明了该字段是不是数据库生成的。 我们将所有数据库生成的属性放到了<code>TemplateProperties</code>集合中，供<code>OnAfterSaveChanges</code>方法遍历
* 我们可以通过Property对象的<code>Metadata.IsPrimaryKey()</code>方法来获得当前字段是不是主键字段
* Property对象的CurrentValue属性表示当前字段的新值，OriginalValue属性表示当前字段的原始值

最后我们修改一下<code>OnAfterSaveChanges</code>, 代码如下

private Task OnAfterSaveChanges(List<AuditEntry> auditEntries)

{

if (auditEntries == null || auditEntries.Count == 0)

return Task.CompletedTask;

foreach (var auditEntry in auditEntries)

{

// Get the final value of the temporary properties

foreach (var prop in auditEntry.TemporaryProperties)

{

if (prop.Metadata.IsPrimaryKey())

{

auditEntry.KeyValues[prop.Metadata.Name] = prop.CurrentValue;

}

else

{

auditEntry.NewValues[prop.Metadata.Name] = prop.CurrentValue;

}

}

// Save the Audit entry

Audits.Add(auditEntry.ToAudit());

}

return SaveChangesAsync();

}

* 在<code>OnBeforeSaveChanges</code>中，我们记录下了当前实体所有需要数据库生成的属性。 在调用父类的<code>SaveChangesAsync</code>方法, 我们可以获取通过property的<code>CurrentValue</code>属性获得到这些数据库生成属性的新值
* 记录下新值，之后我们生成变更实体记录Audit，并添加到上下文中，再次调用SaveChangesAsync方法，将其持久化

# 当前方案的问题和适合的场景

* 这个方案中，整个数据库持久化并不在一个原子事务中，我们都知道Entity Framework的SaveChangesAsync方法是自带事务的，但是调用2次SaveChangeAsync就不是一个事务作用域了，可能出现实体保存成功，Audit实体保存失败的情况
* 由于调用了2次SaveChangeAsync方法，所以Audit实体中的DateTime属性并不能确切的反映保存实体操作的真正时间, 中间间隔了第一次SaveChangeAsync花费的时间(个人认为在<code>OnBeforeSaveChanges</code>中就可以生成这个DateTime让时间更精确一些)
* 如果所有实体属性值都是预生成的，非数据库生成的，作者这个方案还是非常好的，但是如果有数据库自增列或计算列, 还是使用关系型数据库中临时表(Temporal Table)或数据变更捕获(Change Data Capture)特性比较合理

