Jarno Peschier is technisch specialist bij de Caesar Groep

BLOB + Stream = BlobStream

EENVOUDIG GROTE BLOB'S GEBRUIKEN ZONDER ZE GEHEEL IN HET GEHEUGEN TE HEBBEN

Dit artikel probeert een oplossing te geven voor een veelvoorkomend probleem: hoe ga je efficiënt om met blob-data in je applicaties zonder dat je code een onleesbare en moeilijk te onderhouden berg spaghetti wordt. In .NET is hiervoor een redelijk eenvoudige oplossing: combineer de T-SQL-commando's die SQL Server je biedt voor het werken met blobs (Binary Large OBjects) met de mogelijkheden van System.IO.Stream.

Deze combinatie kom je in de praktijk eigenlijk niet veel tegen, wat op zich erg jammer is. Reden genoeg om in dit artikel er eens nader op in te gaan. De C#-solution die bij dit artikel hoort geeft direct bruikbare code, waarbij plaatjes in een databasetabel worden opgeslagen. Met een kleine Windows Forms-beheerapplicatie (zie afbeelding 1) zijn deze data te beheren. Daarnaast kunnen gebruikers deze afbeeldingen met een ASP.NET-applicatie (zie afbeelding 2) via het web raadplegen.

Blobs zijn geweldig

Iedereen kent de zogenaamde Binary Large OBjects wel, BLOBs. In SQL Server hebben we het dan over de datatypes text, ntext en vooral image. Blobs zijn ontzettend handig. Alles wat je ergens op moet bergen zonder dat je er verder veel mee doet, kun je met behulp van blobs in je database kwijt. Als het maar minder dan de maximale twee gigabyte is. De laatste jaren is de hoeveelheid opslagruimte in computers drastisch toegenomen tegen dezelfde of een veel lagere prijs. In de praktijk zijn er echter maar weinig elementen die echt groter zijn dan twee gigabyte.

Dus wat komen we regelmatig in het wild tegen?

- Websites met allerlei plaatjes in een database, zoals het welbekende smoelenboek.
- Documentmanagementsystemen met grote Word-documenten erin, vaak nog verschillende revisies daarvan die alleen maar uit historisch oogpunt worden bewaard .
- Geautomatiseerde catalogi vol met PDF-bestanden van vele tientallen megabytes groot die via het web te raadplegen moeten zijn.

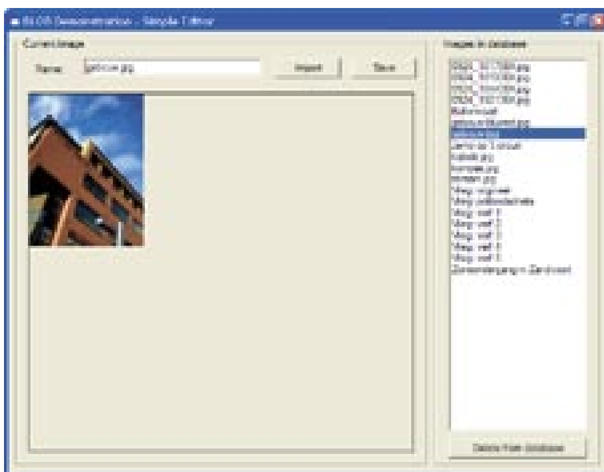
Volkomen vanzelfsprekend. En er is ook niets mis mee. Of toch…?

Ze hebben ook nadelen

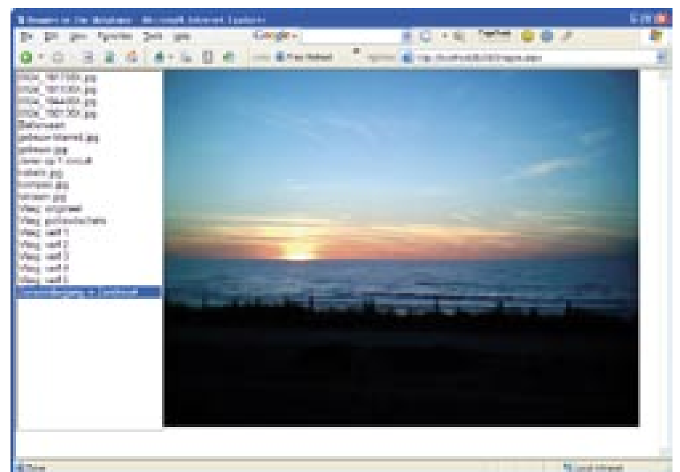
Veel applicaties werken met blobs alsof het gewone velden zijn. Dus net zoals elk ander veld in de database wordt er met INSERT-, SELECT- en UPDATE-statements gewerkt waarin dan de volledige inhoud van de blob wordt gebruikt. En daar gaat het vaak verkeerd, want het gegeven dat een blob twee gigabyte groot kan zijn, wil in de praktijk nog wel eens voor problemen zorgen. Laten we eens kijken naar een simpel voorbeeld. Stel, we hebben een applicatie met een SQL Server-tabel waarin afbeeldingen worden opgeslagen. De tabel is gedefinieerd zoals in codevoorbeeld 1, en wordt gebruikt als een 'filesysteem' voor afbeeldingen, waarbij een unieke naam wordt gebruikt om afbeeldingen terug te vinden.

In deze tabel zit een 24-bits BMP-afbeelding die 17 Mb groot is. Een gebruiker bekijkt deze afbeelding via een webapplicatie die blobs niet apart behandelt. Dit is wat er vervolgens gebeurt:

- De gebruiker kiest 'Grote BMP' om te bekijken.
- De applicatie voert SELECT blob FROM tblBlob WHERE blobId = 'Grote BMP' uit.
- De applicatie krijgt 17 Mb aan data van SQL Server terug.
- De applicatie stuurt daarna deze 17 Mb naar de browser van de gebruiker.



Afbeelding 1. Afbeelding opslaan met een Windows Forms-beheerapplicatie



Afbeelding 2. Afbeelding via het web bekijken met een ASP.NET-applicatie

Dit klinkt logisch, maar hierbij is wel minstens 17 Mb aan geheugen nodig op de webserver voor het afhandelen van de request van de gebruiker. Waarschijnlijk gaat dat wel goed, zeker in de ontwikkel- en testfase van het project. Maar is dat ook nog zo in een productieomgeving waar de applicatie misschien wel 100 gebruikers per seconde te verwerken krijgt en veel van de afbeeldingen zo groot zijn? Die 1,66 Gb aan geheugen die dan in gebruik kan zijn per seconde is misschien wel te veel van het goede. Erg schaalbaar is dit niet.

Wat nu…?

Gelukkig is dit probleem al veel langer bekend en biedt SQL Server een oplossing. De commando's READTEXT en UPDATETEXT zijn waarschijnlijk bij veel van jullie wel bekend[1]. Deze bieden je als programmeur de mogelijkheid specifieke stukjes uit een blob te benaderen in plaats van alle bytes in één keer. Maar waarom worden deze commando's dan niet vaker gebruikt? Het gebruik ervan is in de praktijk helaas een beetje lastig. In plaats van een enkele query moet je met byte-buffers werken en administratief het een en ander bijhouden om te zorgen dat alles tot op de byte goed gaat. Daar moet je zelf in je code moeite voor doen; kijk bijvoorbeeld maar op http://msdn2.microsoft.com/en-us/library/3517w44b.aspx. En in de praktijk komt het er vaak niet van. Vooral niet onder tijdsdruk.

Streams to the rescue

Gelukkig biedt het .NET Framework een prima oplossing voor dit probleem: streams. Simpel gezegd is een stream de invulling van het abstracte idee van 'een rijtje bytes'. Aangezien bijna alles in een computer onder die noemer is te vangen, is het bijna vreemd dat subclasses van System.IO.Stream niet vaker in code opduiken. Een stream heeft een bepaalde lengte (Length) en een soort cursor die de huidige plek in de stream onthoudt (Position) en die mogelijk verplaatst kan worden (CanSeek/Seek). Vanaf die 'huidige' positie zijn vervolgens bytes te lezen (CanRead/Read) en/of te schrijven (CanWrite/Write). Het .NET Framework definieert zelf al een groot aantal concrete subclasses van System.IO.Stream. Een aantal voorbeelden is:

- System.IO.MemoryStream: een stuk geheugen
- System.IO.FileStream: de inhoud van een bestand op schijf
- System.Net.Sockets.NetworkStream: bytes die over een netwerk zijn verstuurd/ontvangen
- System.Security.Cryptography.CryptoStream: een set gecodeerde bytes
- System.IO.BufferedStream: voeg buffering toe aan een andere stream die zelf niet buffert

Het is overigens helemaal niet moeilijk om zelf een stream te schrijven. En zeg nou zelf: een blob voldoet perfect aan het abstracte idee van een stream. De rest van dit artikel zullen we kijken hoe we zo'n stream zelf moeten bouwen en hoe we die vervolgens kunnen gebruiken.

We bouwen een BlobStream

Om te beginnen zullen we een subclass van System.IO.Stream moeten definiëren, zoals in codevoorbeeld 2 is te zien. Deze code doet nog vrijwel niets, maar bevat wel alle verplichte onderdelen van een stream. Er is al wel de aanname gedaan dat we de huidige positie in onze blob zelf gaan bijhouden. Aangezien we met READTEXT en UPDATETEXT overal in de blob in de database kunnen lezen en schrijven heb ik CanSeek, CanRead en CanWrite alvast allemaal op true gezet. De implementatie van Seek is te zien in afbeelding 3 en is eigenlijk heel simpel zodra Length geïmplementeerd is.

Onze stream class zal in de constructor een SqlConnection en een blobId mee moeten krijgen waarmee de juiste blob in de database kan worden gevonden. Ook definiëren we een private GetMetaData-functie die, gegeven de blobId en een SqlCommand, de TEXTPTR-functie in een query gebruikt om de zogenaamde blobpointer en de lengte van de blob op te halen. Hiermee wordt op eenvoudige wijze de Length-functie gedefinieerd. Daarna is het alleen nog maar nodig om de belangrijkste operaties te implementeren:

- Read: roep GetMetaData aan en gebruik de pointer in een aanroep van READTEXT om de juiste bytes te lezen
- Write: roep GetMetaData aan en gebruik de pointer in een aanroep van UPDATETEXT om de juiste bytes te schrijven

De drie SqlCommands (GetMetaData, Read en Write) worden voor

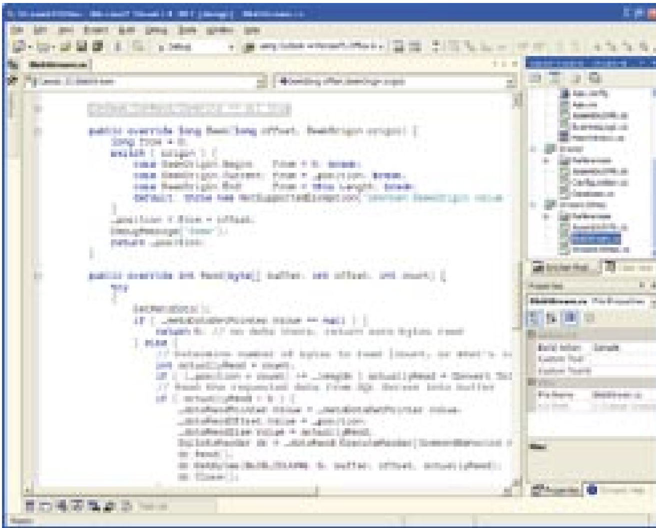
```
CREATE TABLE tblBlob (
  blobId varchar (50),
  blob image NOT NULL
)

ALTER TABLE
  tblBlob
ADD CONSTRAINT
  PK_tblBlob
PRIMARY KEY CLUSTERED
  (blobId)
```

Codevoorbeeld 1.

```
public class BlobStream : System.IO.Stream
{
  private long _position;
  // TODO: Constructor
  public override bool CanSeek
  {
    get
    {
      return true;
    }
  }
  public override bool CanRead
  {
    get
    {
      return true;
    }
  }
  public override bool CanWrite
  {
    get
    {
      return true;
    }
  }
  public override long Seek(long offset, SeekOrigin origin)
  {
    // TODO: _position aanpassen
  }
  public override int Read(byte[] buffer, int offset, int count)
  {
    // TODO: Nog schrijven...
  }
  public override void Write(byte[] buffer, int offset, int count)
  {
    // TODO: Nog schrijven...
  }
  public override void Flush()
  {
    // TODO: Nog schrijven...
  }
  public override long Position
  {
    get
    {
      return _position;
    }
    set
    {
      Seek(value, SeekOrigin.Begin);
    }
  }
  public override void SetLength(long value)
  {
    throw new NotSupportedException();
  }
  public override long Length
  {
    // TODO: Nog schrijven...
  }
}
```

Codevoorbeeld 2.



Afbeelding 3. De implementatie van Seek

efficiëntie als private members van BlobStream geïnitialiseerd in de constructor en daarna hergebruikt als ze nodig zijn. In afbeelding 3 is ook een deel van de code van de Read-methode te zien. Voor nadere details verwijs ik graag naar downloadbare code van de aan het eind van dit artikel genoemde voorbeeldapplicaties.

Klaar

Nu we een BlobStream-class hebben is de vraag natuurlijk: hoe gebruiken we die nu? Als voorbeeld laat ik in codevoorbeeld 3 de vrijwel volledige code zien van de Image.aspx-pagina uit onze webapplicatie van afbeelding 2. Deze is bijvoorbeeld op de volgende manier te gebruiken als SRC-attribuut van een <IMG> tag in HTML: http://localhost/BLOB/Image.aspx?id=gebouw.jpg. Zoals je ziet stelt efficiënt gebruik van een blob niet zoveel meer voor als je een class als BlobStream hebt. Je maakt de juiste instantie van de class aan en gebruikt hem waar je de bytes van de afbeelding nodig hebt.

Dit wordt gemakkelijk gemaakt doordat het .NET Framework op allerlei plaatsen met streams overweg kan. Op de paar plaatsen waar dit misschien iets minder gemakkelijk gaat, zoals bij het kopiëren van de inhoud van een stream naar een andere stream of het converteren tussen streams en strings, levert de code in de bij dit voorbeeld meegeleverde StreamUtilities-class uitkomst. In de Windows Forms-editor uit afbeelding 1 is de code om een afbeelding uit de database te halen en te laten zien in een PictureBox zo simpel als in codevoorbeeld 4. Immers, System.Drawing.Image bevat een static method FromStream die uit een stream een plaatje kan inlezen in allerlei formaten zoals JPEG, GIF en BMP. En omgekeerd is een plaatje heel eenvoudig in een blob op te slaan:

1. Lees het plaatje van disk met Image.FromDisk().
2. Maak een nieuw record in tblBlob aan met een lege blobwaarde.
3. Maak een BlobStream-instantie aan met hetzelfde blobId.
4. Gebruik Image.Save() om het plaatje direct in de BlobStream op te slaan in het gewenste formaat.

Voor de precieze details verwijs ik weer graag naar de voorbeeldsolution. Bedenk dat je eventueel tussen stap 1 en 4 hierboven met GDI+ op de Image kunt 'tekenen': een kader toevoegen bijvoorbeeld, of een copyright message in een van de hoeken kunt zetten. Ook kun je de afbeelding met de Image.GetThumbnailImage-methode verkleinen en zo automatisch ook een thumbnail opslaan van alle plaatjes in je database. De mogelijkheden zijn eindeloos.

De voordelen op een rij

We hebben nu een eigen stream-class geschreven die weet hoe hij met blobs in een SQL Server-database moet omgaan. Hiermee zijn we in één klap de eerder genoemde nadelen kwijt. De hele blob is niet meer ineens in het geheugen nodig, omdat BlobStream alleen de bytes ophaalt/wegschrijft waar de gebruiker van de stream om vraagt. Ook is de code goed leesbaar, doordat de details van het gebruik van READTEXT en UPDATETEXT netjes zijn verborgen in de stream-class. Dit gebruik van blobs is hiermee ook gemakkelijker herbruikbaar in andere applicaties; of aan te passen als dat nodig is, bijvoorbeeld als er op een ander DBMS dan SQL Server gewerkt moet worden. En tot slot sluiten we ook prima aan bij het .NET Framework dat (terecht) een voorkeur voor streams lijkt te hebben.

```
private void Page_Load(object sender, System.EventArgs e)
{
  string blobId = Request.Params["id"];
  try {
    SqlConnection database = Configuration.GetConnection();
    BlobStream blob = new BlobStream(blobId, database);
    Response.ContentType = "image/jpeg";
    StreamUtilities.Copy(blob, Response.OutputStream);
  } catch {
    Response.Clear();
    Response.StatusCode = 404; // "Not Found"
    Response.ContentType = "text/plain";
    Response.Write("Requested content not available");
  } finally {
    Response.End();
  }
}
```

Codevoorbeeld 3.

```
SqlConnection database = Configuration.GetConnection();
BlobStream existingBlob = new BlobStream(blobId, database);
PictureBox.Image = Image.FromStream(existingBlob);
```

Codevoorbeeld 4.

Tot slot

Hopelijk heeft dit artikel je aan het denken gezet. Het beschrijft een elegante manier om efficiënt met blobs om te gaan, en die je op eenvoudige wijze direct zelf kunt gebruiken. Daarnaast is het ook een voorbeeld van het praktisch nut van de System.IO.Stream-abstractie. Zoals je ziet is het helemaal niet zo moeilijk om zelf streams te schrijven om operaties op 'rijen bytes' netjes herbruikbaar weg te stoppen op een manier die aansluit bij wat het .NET Framework zelf doet.

Jarno Peschier heeft informatica gestudeerd aan de Universiteit Utrecht en werkt sinds 1999 als technisch specialist bij de Caesar Groep in Utrecht (www.caesar.nl). Jarno is te bereiken via email op j.peschier@caesar.nl

Referenties

Gebruikte T-SQL-commando's:

- TEXTPTR - msdn2.microsoft.com/en-us/library/ms176068.aspx
- READTEXT - msdn2.microsoft.com/en-us/library/ms187365.aspx
- UPDATETEXT - msdn2.microsoft.com/en-us/library/ms189466.aspx

Voorbeelden van omgaan met blobs zonder Stream-objecten:

Conserving Resources When Writing BLOB Values to SQL Server - msdn2.microsoft.com/en-us/library/3517w44b.aspx

Obtaining BLOB Values from a Database - msdn2.microsoft.com/en-us/library/87z0hy49.aspx

Writing BLOB Values to a Data Source - msdn2.microsoft.com/en-us/library/4f5s1we0.aspx

Noten:

1 Let op: In de documentatie van SQL Server 2005 staan deze commando's gemarkeerd als 'Zullen in toekomstige versies verdwijnen. Gebruik SUBSTRING vanaf nu in nieuwbouw.' Hiermee is in dit artikel nog geen rekening gehouden, maar de BlobStream class is natuurlijk zonder veel problemen om te bouwen zodra dit echt nodig is.