Tối Ưu Hóa Chương Trình C#

Đây là một số kinh nghiệm về tối ưu hóa mã nguồn C# sau một khoảng thời gian làm việc với nó. Bạn có thể áp dụng một số thủ thuật này trong các ngôn ngữ khác như VB.Net, Java…

Để đo thời gian thực thi của các đoạn mã ví dụ bên dưới, bạn có thể dùng DateTime.Now.Ticks lưu thời điểm bắt đầu và kết thúc. Tuy nhiên .Net cung cấp cho bạn sẵn đối tượng Stopwatch (đồng hồ bấm giờ) nằm trong không gian tên System.Diagnostics để dùng cho những công việc dạng này.

Trong mỗi phần tôi sẽ so sánh hai phương pháp (đoạn mã), phương pháp thứ hai sẽ là phương pháp tối ưu hơn cho bạn lựa chọn. Mặc dù các giải pháp thay thế có thể tốt hơn nhưng không hẳn đã là tối ưu, việc tối ưu một đoạn mã đòi hỏi sự hiểu biết và phân tích khá sâu vào nền tảng .Net, hơn nữa còn phụ vào thuật toán bạn sử dụng trong từng trường hợp.

1. So sánh chuỗi:

Ở đây tôi dùng hai phương pháp so sánh chuỗi thường sử dụng (có phân biệt hoa thường). Điểm khác biệt giữa hai phương thức này là phương thức thứ 1 là tĩnh (static) nên ta có thể gọi trực tiếp từ lớp String.

- (1) int String.Compare(string strA, string strB, bool ignoreCase)
- (2) bool string.Equals(string value, StringComparison comparisonType)

string s1=”aaa”;

string s2=”AAA”;

Đoạn mã 1:

```
for (int i = 0; i < 100000; i++)
{
    bool b = String.Compare(s1, s2,true)==0;
}
```

Đoạn mã 2:

```
for (int i = 0; i < 100000; i++)
{
```

```
        bool b = s1.Equals(s2,StringComparison.OrdinalIgnoreCase);

    }
```

Đoạn mã thứ nhất chạy chậm hơn đoạn thứ hai hơn 3 lần. Tuy nhiên nếu bạn sử dụng tham số StringComparison.CurrentCultureIgnoreCase cho phương thức Equals thì tốc độ giữa hai đoạn mã là xấp xỉ. Một số người dùng cách chuyển cả hai chuỗi về dạng chữ hoa hoặc chữ thường rồi so sánh sẽ tốn thời gian lâu nhất (hơn 2 lần so với cách một).

2. Xây dựng chuỗi – String và StringBuilder:

Đây có lẽ là điều bạn thường gặp và cũng đã nắm bắt được sự khác biệt rõ ràng giữa chúng. Với số lần lặp tương đối lớn bạn sẽ có một khoảng thời gian chờ tương đối lâu khi làm việc với lớp String, vì thế tôi sẽ giảm số lần lặp xuống trong ví dụ này.

Đoạn mã 1:

```
    string str="";for (int i = 0; i < 10000; i++)

    {

        str += "a";

    }
```

Đoạn mã 2:

```
StringBuilder str=new StringBuilder();

    for (int i = 0; i < 10000; i++)

    {

        str.Append("a");

    }
```

Kết quả cho ta thấy đoạn mã một chạy chậm hơn khoảng từ 200 đến 300 lần đoạn mã hai. Nguyên nhân là toán tử + của lớp string sẽ tạo ra một đối tượng string mới trong mỗi lần lặp, trong khi phương thức Append của StringBuilder sẽ nối trực tiếp vào chuỗi hiện tại.

Tuy nhiên cũng cần chú ý điều này có thể ngược lại nếu như bạn cần chuyển chuỗi StringBuilder thành String trong mỗi lần lặp. Tốc độ thực thi của đoạn mã thứ hai sẽ lâu hơn so với đoạn mã một. Hãy kiểm chứng bằng cách chạy thử đoạn mã hai sau khi sửa lại như sau:

Đoạn mã 2 (đã sửa):

```
StringBuilder str = new StringBuilder();

string strRet;

for (int i = 0; i < 10000; i++)

{

        str.Append(“a”);

        strRet = str.ToString();

}
```

3. Nối chuỗi – Phương thức Insert() và toán tử +:

Bạn thường sử dụng hai cách để nối hai chuỗi lại với nhau là dùng toán tử + và phương thức Insert() của đối tượng string. Cách đầu tiên được sử dụng nhiều hơn vì cách viết tiện lợi và dễ hiểu hơn, tuy nhiên bạn chỉ có thể nối vào đầu hoặc cuối chuỗi. Hãy thử so sánh xem phương pháp nào cho tốc độ thực thi nhanh hơn, giả sử bạn cần nối một chuỗi con vào đầu một chuỗi.

Đoạn mã 1:

```
string str=””";

for (int i = 0; i < 10000; i++)

{

        str = “string”;

        str= str.Insert(0, “my “);

}
```

Đoạn mã 2:

```
string str=””";

for (int i = 0; i < 10000; i++)

{

        str = “string”;

        str = “my ” + str;
```

```
}
```

Nếu chạy thử vài lần, bạn sẽ nhận thấy rằng đoạn mã thứ hai chạy nhanh gấp đôi đoạn mã một. Tuy nhiên nếu như không khởi tạo lại giá trị của biến str trong mỗi lần lặp, tốc độ của hai đoạn mã này là xấp xỉ nhau.

4. Cắt chuỗi – Substring() và Remove():

Hai phương thức trên của lớp string có chức năng khá giống nhau là cắt bỏ một phần chuỗi nguồn. Cả hai phương thức đều có 2 kiểu nạp chồng, và đều yêu cầu truyền vào vị trí bắt đầu của chuỗi. Tùy vào trường hợp, Substring() thích hợp cho việc cắt bỏ chuỗi phía trước, còn Remove() lại thường dùng để cắt bỏ phía sau chuỗi. Trong ví dụ này tôi sẽ dùng hai phương thức này để cắt bỏ 1 kí tự phía trước chuỗi:

Đoạn mã 1:

```
string str;
for (int i = 0; i < 10000; i++)
{
    str = “string”;
    str = str.Remove(0, 1);
}
```

Đoạn mã 2:

```
string str;
for (int i = 0; i < 10000; i++)
{
    str = “string”;
    str = str.Substring(1);
}
```

Sự khác biệt giữa đoạn mã hai so với đoạn mã một là tốc độ nhanh hơn khoảng 2 lần. Tuy nhiên sự khác biệt này sẽ trở nên khó phân biệt nếu như bạn dùng chúng để cắt ở vị trí khác, như là cuối chuỗi chẳng hạn.

5. Chuyển đối tượng về dạng chuỗi – Format() và ToString();

Format là một phương thức khá hiệu quả trong một số trường hợp bạn cần thêm các tham số và một chuỗi, sử dụng tương tự như cách bạn in một chuỗi ra màn hình console bằng phương thức WriteLine().

Đoạn mã 1:

```
string str = “”;

int obj = 1;

for (int i = 0; i < 100000; i++)

{

    str = String.Format(“{0}{0}{0}{0}{0}{0}{0}{0}{0}{0}”, obj);

}
```

Đoạn mã 2:

```
string str = “”;

int obj = 1;

for (int i = 0; i < 100000; i++)

{

    string s = obj.ToString();

    str = s + s + s + s + s + s + s + s + s + s;

}
```

Ở đây tôi tạo ra một đối tượng string với giá trị là mười số 1 trong mỗi lần lặp. Cả hai phương pháp này đều chạy khá lâu với số lần lặp là 100000. Tuy nhiên đoạn mã thứ nhất sẽ tốn thời gian gấp gần 5 lần so với đoạn mã thứ hai, mặc dù đoạn mã thứ hai còn sử dụng toán tử cộng chuỗi.

6. Sự khác biệt giữa các phương thức nạp chồng (overloaded):

Bạn có thể không để ý rằng giữa các phương thức nạp chồng, thực hiện cùng một công việc lại có sự khác biệt về khoảng thời gian mà chúng thực thi. Điều này thường thấy trong các phương thức với tham số là một hoặc nhiều kiểu đối tượng được tạo ra từ những thành phần đơn giản hơn.

Chẳng hạn như phương thức khởi tạo của đối tượng Rectangle, ta sẽ thử xét hai phương thức:

- (1) Rectangle (Point location, Size size)

- (2) Rectangle (int x, int y, int width, int height)

Sự khác biệt về thời gian thực thi giữa 2 phương thức phụ thuộc vào đoạn mã chúng ta viết.

Nếu bạn so sánh giữa 2 đoạn mã sau:

Đoạn mã 1:

```
for (int i = 0; i < 100000; i++)
{
    Rectangle rec = new Rectangle(new Point(i,i), new Size(i,i));
}
```

Đoạn mã 2:

```
for (int i = 0; i < 100000; i++)
{
    Rectangle rec = new Rectangle(i,i,i,i);
}
```

Bạn có thể nhận thấy là đoạn mã thứ nhất chạy lâu hơn đoạn thứ hai khoảng 7 lần. Lý do xảy ra điều này bạn có thể đoán được là phương thức thứ nhất phải tạo ra 2 đối tượng trong mỗi lần lặp.

7. Hạn chế sử dụng khối try catch:

Việc xử lý các ngoại lệ trong C# thường chiếm một khoảng thời gian tương đối lâu. Một số ngoại lệ có thể dự đoán trước được ,và thay vì sử dụng khối try catch để bao đoạn mã không an toàn này lại, bạn hãy tự xử lý thông qua các dòng lệnh kiểm tra trước khi để đoạn mã tiếp tục thực hiện. Đây là một ví dụ:

```
int a = 0, b = 10;
```

Đoạn mã 1:

```
for (int i = 0; i < 1000; i++)
{
    try
    {
```

```
                int c = b / a;

        }

        catch { }

}
```

Đoạn mã 2:

```
for (int i = 0; i < 1000; i++)

{

        if (a == 0)
                continue;

        int c = b / a;

}
```

Dòng lệnh int c = b / a dĩ nhiên luôn ném ra ngoại lệ Divide by Zero (chia cho 0) nếu được thực thi. Tuy nhiên ngoại lệ này bạn có thể đoán trước và kiểm tra mẫu số phải khác 0 trước khi tiếp tục hay không. Sự cẩn trọng như trong đoạn mã 2 giúp bạn tiết kiệm được thời gian khoảng 30,000 (30 nghìn) lần so với đoạn mã 1.

8. Hạn chế việc gọi phương thức:

Việc gọi phương thức nhiều lần có thể làm giảm tốc độ thực thi của chương trình so với việc viết trực tiếp thân hàm vào nơi cần thiết. Vì thế đối với những phương thức có nội dung đơn giản và không được gọi ở nhiều nơi khác nhau, bạn nên loại bỏ bớt đi những phương thức này. Điều này thường ít xảy ra nhưng có thể giúp bạn cẩn thận hơn trong việc quyết định có nên tạo ra một phương thức mới hay không.

Giả sử bạn có 1 phương thức Method() đơn giản như sau:

```
private void Method()

{

        int a = 1, b = 2;

        int c = a + b;

}
```

Bạn có hai cách để thực thi hai phương thức trên như sau:

Đoạn mã 1:

```
for (int i = 0; i < 1000000; i++)
{
    Method();
}
```

Đoạn mã 2:

```
for (int i = 0; i < 1000000; i++)
{
    int a = 1, b = 2;
    int c = a + b;
}
```

Đoạn mã thứ hai chạy nhanh đoạn thứ nhất 3 lần. Các lệnh nhảy mặc dù rất tốn rất ít chi phí nhưng với số lượng nhiều có thể làm chậm tốc độ thực thi của chương trình khá rõ ràng.

9. Sử dụng cấu trúc thay cho lớp:

Đối với những kiểu dữ liệu đơn giản và không cần tham chiếu, bạn nên sử dụng cấu trúc (structure) thay cho lớp (class), điều này sẽ giúp giảm bớt chi phí tài nguyên tiêu hao cho chương trình của bạn. Tuy nhiên bạn cũng cần cân nhắc dựa vào yêu cầu của dự án trước khi quyết định dùng struct hay class cho phù hợp.

Giả sử bạn có một lớp và một cấu trúc có cùng thành viên và thuộc tính đơn giản như sau:

```
class MyClass
{
    private int value;
    public int Value
    {
        get { return this.value; }
        set { this.value = value; }
```

```
        }
    }
    struct MyStruct
    {
        private int value;
        public int Value
        {
            get { return this.value; }
            set { this.value = value; }
        }
    }
```

Hai đoạn mã cũng thực hiện cùng một nhiệm vụ:

Đoạn mã 1:

```
    for (int i = 0; i < 1000000; i++)
    {
        MyClass myClass = new MyClass();
        myClass.Value = 1;
    }
```

Đoạn mã 2:

```
    for (int i = 0; i < 1000000; i++)
    {
        MyStruct myStruct= new MyStruct();
        myStruct.Value = 1;
```

```
}
```

Khi chạy thử bạn có thể nhận thấy đoạn mã thứ hai chạy nhanh hơn khoảng trên 4 lần so với đoạn mã thứ nhất. Lý do chính khá đơn giản là cấu trúc đơn giản và có ít tính năng hơn lớp, vì thế như đã nói trước, bạn chỉ nên sử dụng đối với những kiểu dữ liệu đơn giản, thành viên của nó thường là những kiểu dữ liệu đã được xây dựng sẵn.

10. Sử dụng tập hợp (collection) có định kiểu:

Mục đích của việc này là giúp hạn chế tối đa việc ép kiểu khi bạn thao tác với các tập hợp này. Thường thì chúng ta chỉ chứa một loại đối tượng trong một tập hợp, vì thế thay vì dùng lớp ArrayList để lưu trữ, bạn hãy thử dùng List<T>. Các tập hợp có định kiểu được cung cấp trong namespace System.Collections.Generic.

11. Mảng (array) và tập hợp (collection) – tốc độ hay sự linh hoạt:

Bạn có thể đã biết rằng việc sử dụng mảng sẽ giúp truy xuất nhanh hơn tuy hơn lại không linh hoạt bằng tập hợp. Về mặt lý thuyết, tập hợp giống như một danh sách liên kết (linked list) với khả năng cấp phát động để cung cấp các thao tác thêm, xóa theo vị trí mà mảng không hỗ trợ. Trong một số trường hợp, nếu không cần đến các chức năng này của tập hợp, bạn nên sử dụng mảng để tối ưu cho chương trình của mình.

Đoạn mã 1:

```
List<int> list = new List<int>();

for (int i = 0; i < 1000000; i++)

{

        list.Add(1);

}
```

Đoạn mã 2:

```
int[] arr = new int[1000000];

for (int i = 0; i < arr.Length; i++)

{

        arr[i] = 1;

}
```

Tốc độ thực thi của đoạn mã thứ hai có thể nhanh hơn 4-6 lần đoạn mã một. Tuy nhiên nếu bạn sử dụng ArrayList trong đoạn mã một, sự khác biệt này có tăng lên đến 15 lần. Một lần nữa bạn thấy sự cải thiện đáng kể khi sử dụng tập hợp có định kiểu.

12. Nạp dữ liệu – DataSet và DataReader

Có một sự khác biệt dễ nhận thấy khi bạn thao tác với những loại cơ sở dữ liệu khác nhau. Chẳng hạn thử nghiệm trên SQL Server 2005 và Access thì tốc độ thực thi trên SQL Server sẽ nhanh hơn, sự khác biệt giữa các cách xử lý dữ liệu cũng nhiều hơn.

Trong minh họa này, tôi sẽ sử dụng SQL Server để nạp dữ liệu từ một bảng và lặp qua từng dòng để thêm dữ liệu của một cột nào đó vào tập hợp List<string>. Cách đầu tiên sử dụng một DataSet để lưu dữ liệu và cách thứ hai sử dụng một SqlDataReader để đọc dữ liệu.

Ở đây tôi sử dụng lớp SqlAccess với hai phương thức chính để lấy dữ liệu từ cơ sở dữ liệu SQL Server 2005 thông qua DataSet và reader. Nội dung của chúng như sau:

```
public DataSet GetDataSet(string tableName)
{
        string strQuery= “select * from ” + tableName;
        SqlCommand cmd = new SqlCommand(strQuery,conn);
        SqlDataAdapter da = new SqlDataAdapter(cmd);
        DataSet ds = new DataSet();
        da.Fill(ds, tableName);
        return ds;
}
public SqlDataReader GetReader(string tableName)
{
        string strQuery = “select * from ” + tableName;
```

```
        SqlCommand cmd = new SqlCommand(strQuery,conn);

        return cmd.ExecuteReader();

    }
```

Đoạn mã 1 - Sử dụng DataSet:

```
    List<string> list = new List<string>();

    DataSet ds= sqlAccess.GetDataSet("tblUser");

    foreach(DataRow row in ds.Tables[0].Rows)

    list.Add(row["Alias"].ToString());
```

Đoạn mã 2 – Sử dụng SqlDataReader:

```
    List<string> list = new List<string>();

    SqlDataReader reader = sqlAccess.GetReader("tblUser");

    while (reader.Read())

    list.Add(reader["Alias"].ToString());

    reader.Dispose();
```

Theo kết quả thử nghiệm thì tốc độ của SqlDataReader nhanh gấp 2 lần so với khi sử dụng DataSet. Chính vì thế trong những trường hợp không cần phải lưu dữ liệu tạm thời để làm việc, bạn nên tránh sử dụng DataSet để nạp dữ liệu. Việc sử dụng reader mặc dù có nhiều hạn chế hơn DataSet như chỉ có thể đọc dữ liệu, chỉ đi tới theo kiểu "thà chết chớ lui" nhưng trong nhiều trường hợp, bạn cũng cần đánh đổi những chức năng đó để đạt được tốc độ cần thiết cho ứng dụng của mình.

Nếu trong hai ví dụ trên bạn sử dụng kết nối đến tập tin Access thì tốc độ sẽ chậm hơn đồng thời sự khác biệt giữa tốc độ thực thi của cách dùng DataSet với OleDbDataReader sẽ chỉ khoảng 1.5 (nghiêng về phía reader).

13. Thuộc tính CommandType của đối tượng Command

Có một sự khác biệt khi chọn lựa giá trị cho thuộc tính CommandType của đối tượng Command. Mặc định thì CommandType là Text tức là dựa trên câu lệnh Sql để thực thi lệnh. Tuy nhiên nếu chuyển CommandType sang kiểu StoredProcedure hoặc TableDirect thì tốc độ sẽ nhanh hơn.

Đáng tiếc là không thể sử dụng kiểu CommandType.TableDirect khi kết nối đến SQL Server, chính vì thể tôi sẽ sử dụng đối tượng OleDbDataReader để kết nối và đọc 1 bảng dữ liệu từ file Access. Các phương thức sau lần lượt sử dụng từng giá trị CommandType khác nhau để cùng lấy dữ liệu từ bảng SinhVien:

Phương thức 1 – Sử dụng CommandType.Text

```
public OleDbDataReader GetReader_Text()
{
    string strQuery = “select * from SinhVien”;
    OleDbCommand cmd = new OleDbCommand(strQuery,con);
    return cmd.ExecuteReader();
}
```

Phương thức 2 – Sử dụng CommandType.StoredProcedure

```
public OleDbDataReader GetReader_StoredProcedure()
{
    OleDbCommand cmd = new OleDbCommand(“qryGetUsers”,con);
    cmd.CommandType = CommandType.StoredProcedure;
    return cmd.ExecuteReader();
}
```

Phương thức 3 – Sử dụng CommandType.TableDirect

```
public OleDbDataReader GetReader_TableDirect()
```

```
{
        OleDbCommand cmd = new OleDbCommand("SinhVien", con);
        cmd.CommandType = CommandType.TableDirect;
        return cmd.ExecuteReader();
}
```

Kết quả sau khi thực thi 3 phương thức này như sau (tốc độ hoàn thành tác vụ):

- **CommandType.Text: 2618**
- **CommandType.StoredProcedure: 2444**
- **CommandType.TableDirect 1231**

Như bạn có thể thấy là kiểu TableDirect có tốc độ thực thi nhanh nhất. Cách này rất thích hợp trong trường hợp bạn chỉ cần nạp một bảng dữ liệu đơn lẻ, đồng thời thao tác viết lệnh cũng nhanh hơn. Đối với những trường hợp nạp dữ liệu phức tạp hoặc để thực thi các lệnh Insert, Delete, Update,… thì hãy sử dụng StoredProcedure bất cứ khi nào có thể.

Theo Microsoft thì Stored Procedure có thể được thực thi mà không cần phải qua bước thông dịch, đồng thời giúp làm giảm lượng dữ liệu truyền tải giữa client và server trong các ứng dụng liên quan đến mạng.

14. Chỉ nạp những cột dữ liệu cần thiết

Điều này có lẽ ai cũng có thể hiểu, việc truy vấn dữ liệu với số cột ít hơn sẽ đồng nghĩa với lượng dữ liệu trả về ít và tốc độ cũng nhanh hơn. Một số lập trình viên do thói quen thường sử dụng dấu '*' để nạp dữ liệu ngay cả trong trường hợp họ chỉ cần 1 đến 2 trường là đủ. Sự khác biệt về tốc độ này phụ thuộc vào số cột mà bạn muốn truy vấn nên sẽ không có giá trị so sánh nào trong phần này.

15. Tìm kiếm dữ liệu – DataTable và DataView

Một chức năng không thể thiếu trong các ứng dụng có kết nối cơ sở dữ liệu đó là tìm kiếm. Ở đây ta chỉ so sánh hai phương pháp tìm kiếm được hỗ trợ trong đối tượng DataTable và DataView.

Để sử dụng được phương thức Find này, điều cần làm là phải xác định tập khóa chính cho bảng. Giả sử tôi đọc dữ liệu từ bảng tblUser có cột Alias có các giá trị không trùng nhau, vì thế tôi có thể sử dụng cột này làm khóa chính.

```
DataTable table = sqlAccess.GetDataSet(“tblUser”).Tables[0];
table.PrimaryKey = new DataColumn[] { table.Columns["Alias"] };
```

Đối với DataView thì “yêu sách” hơn, tức là để tìm kiếm được ta cần phải làm một bước nữa là sắp xếp cho nó.

```
DataTable table = sqlAccess.GetDataSet(“tblUser”).Tables[0];
table.PrimaryKey = new DataColumn[] { table.Columns["Alias"] };
DataView dv = table.DefaultView;
dv.Sort = “Alias”;
```

Ta sẽ sử dụng hai phương thức sau đây để kiểm tra tốc độ của hai phương pháp này:

Phương thức 1 – Tìm kiếm bằng DataTable

```
long Test1()
{
    DataTable table = sqlAccess.GetDataSet(“tblUser”).Tables[0];
    table.PrimaryKey = new DataColumn[] { table.Columns["Alias"] };
    string email = “”;
    Stopwatch sw = Stopwatch.StartNew();
    for (int i = 0; i < 100000; i++)
    {
        DataRow row = table.Rows.Find(“YinYang”);
        email = row["Email"].ToString();
```

```
        }
        sw.Stop();
        return sw.ElapsedMilliseconds;
    }
```

Phương thức 2 – Tìm kiếm bằng DataView

```
    long Test2()
    {
    DataTable table = sqlAccess.GetDataSet(“tblUser”).Tables[0];
    table.PrimaryKey = new DataColumn[] { table.Columns["Alias"] };
    DataView dv = table.DefaultView;
    dv.Sort = “Alias”;
    string email = “”;
    Stopwatch sw = Stopwatch.StartNew();
    for (int i = 0; i < 100000; i++)
    {
        int index = dv.Find(“YinYang”);
        email = dv[index].Row["Email"].ToString();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
    }
```

Quá trình hai phương thức trên không xảy ra bất kì ngoại lệ nào, có nghĩa là cả hai phương thức Find() đều tìm thấy dữ liệu trong bảng. Tuy nhiên xét về tốc độ thì phương thức 2 nhanh hơn khoảng 1.5 lần.

16. Lọc dữ liệu – DataTable và DataView

Bên cạnh tìm kiếm thì lọc dữ liệu cũng là một chức năng rất quan trọng trong nhiều trường hợp như tìm kiếm kết quả tương đối, hiển thị dữ liệu theo từng mục chọn,… Cả DataTable và DataView đều có phương pháp để lọc dữ liệu, tuy nhiên cách sử dụng không giống nhau. Để lọc bằng DataTable, ta sử dụng phương thức Select(), còn đối với DataView thì phải gán biểu thức lọc cho thuộc tính RowFilter.

Đối với phương thức Select() thì DataTable trả về một mảng các đối tượng DataRow, trong khi đó, DataView thay đổi lại các phần tử của nó dựa vào biểu thức lọc với dữ liệu được lưu trong thuộc tính Table của nó.

Xét hai phương thức sau:

Phương thức 1 – Lọc bằng DataTable

```
long Test1()
{
    DataTable table = sqlAccess.GetDataSet(“tblUser”).Tables[0];
    string email = “”;
    Stopwatch sw = Stopwatch.StartNew();
    for (int i = 0; i < 100000; i++)
    {
    DataRow[] row = table.Select(“Alias like ‘Yin%’”);
    email = row[0]["Email"].ToString();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
```

```
}
```

Phương thức 2 – Lọc bằng DataView

```
long Test2()
{
    DataTable table = sqlAccess.GetDataSet("tblUser").Tables[0];
    DataView dv = table.DefaultView;
    string email = "";
    Stopwatch sw = Stopwatch.StartNew();
    for (int i = 0; i < 100000; i++)
    {
        dv.RowFilter = "Alias like 'Yin%'";
        email = dv[0].Row["Email"].ToString();
    }
    sw.Stop();
    return sw.ElapsedMilliseconds;
}
```

Khi chạy thử thì phương thức 1 chiếm thời gian lâu hơn khoảng 100 lần so với phương thức 2. Ví dụ như đối với cơ sở dữ liệu của tôi thì kết quả là (tương đối):

-Phương thức 1 mất 6000ms (mili giây)

-Phương thức 2 mất 60ms

Nếu như thay biểu thức lọc trên với biểu thức lọc chính xác hơn bằng cách sử dụng dấu "=" thì tốc độ sẽ nhanh hơn. Ví dụ ta sửa lại biểu thức lọc ở hai ví dụ trên thành:

Alias = 'YinYang'

Thì kết quả là:

-Phương thức 1 mất 1900ms

-Phương thức 2 mất 55ms

Khoảng cách về tốc độ giữa hai phương thức được rút ngắn lại còn khoảng 34.5 lần.

Dĩ nhiên trong thực tế khi giải quyết các vấn đề về lọc thì người ta thường sử dụng DataView. Còn phương thức Select() của DataTable thì giống như một kiểu truy vấn đơn giản để lấy về những dòng thích hợp cho mục đích nào đó. Ngoài ra Select() còn dùng để chuyển nhanh một DataTable thành một mảng các DataRow, sắp xếp, và còn có thể lọc để lấy ra những dòng có trạng thái đặc biệt như các dòng Deleted, Added, Unchanged,…thông qua một tham số kiểu enum DataViewRowState. Bạn hãy coi thử 4 overload của phương thức này để hiểu rõ hơn cách sử dụng.