第3章

编码规范

本章内容

- 概述
- 数据库设计及编码规范
- 客户端代码编码规范
- 服务器代码编码规范

3.1 概述

对于一个产品来说，开发人员的接口无非是代码、SQL。如果在开发过程中严格遵守编码规范，则会避免后期出现很多不必要的问题，这对缩短产品开发周期也是很有帮助的。

这些编码规范来源于工作中的不断积累和借鉴，涉及编码的方方面面，很多是平常工作中注意不到的内容，希望对读者有所帮助。

主要内容涉及三个基本方面：数据库设计及编码规范，服务器端代码规范，客户端代码规范。

3.2 编码规范

本节规范以 SQL Server 数据库为例进行说明。

3.2.1 数据库设计及编码规范

1．控制数据库碎片增长过快

碎片增长原因在数据库优化章节的专题研究中，有详细介绍。这里着重强调一下相关代码规范，如果想更详细了解数据库优化内容，请见第 6 章。

以下是笔者在工作中实际遇到的问题和解决方案。

（1）每个表至少保留一个聚集索引

表按存储方式分为聚集表——创建聚集索引的表和非聚集表（堆）——未创建聚集索引的表两种类型。

对于聚集表，可以通过重新创建或重新组织达到对表数据碎片清理的目的，因为聚集索引在完全二叉树的叶节点即是表数据位置。而对于堆，由于没有聚集索引，数据存储相对比较散列，即使碎片整理也只能对所有非聚集索引进行整理，不能对数据本身进行整理。

如果表不含有聚集索引，即使采用 SSMS 的数据库或文件收缩功能，也无法收缩，导致账套太大，不好管理，查询速度变慢。

（2）避免使用有可能被再次修改的字段作为聚集索引

虽然我们建议用“时间”字段作为聚焦索引，但聚集索引字段尽量用 CreatedTime 这种永远不改变的字段代替 updatedTime 这种后续可能被修改的字段。

因为一旦聚集索引字段被修改，由于索引的规则数据库要对字段重新排序，则会导致数据的批量移动、更新效率较低，也容易导致碎片产生。

说明：

CreateTime（创建时间）：在数据生成时写入当前时间作为初始值，后续不可修改。

UpdateTime（修改时间）：数据每被修改一次就更新一次时间，值为当前修改数据的时间。

（3）Guid 类型字段建立索引后会产生较多碎片

◆ 对表的主外键关系（主键和外键）避免使用 Guid 类型标识，一旦使用必然会创建索引。

◆ 对 Guid 类型的字段不建议创建索引。

◆ Guid 类型字段应该禁止建立聚集索引。

由于 Guid 值大小具有随机性，这样的字段建立聚集索引经常会导致即使一个很小的插入也会产生大量批量数据移动。

Guid 除了大小的随机性，它本身占用的字节也比较大，会导致索引用很少时间就产生大量碎片。

如果产品中违反了以上规则，并且已经在使用了，可以通过如下方式进行优化。

1）数据库端修正

SQL Server 也有内置的函数，即 NEWSEQUENTIALID()。目前默认的函数为 NEWGUID()，是无序的。这样可以有效避免在插入数据时索引碎片的产生。

2）代码端生成修正

根据 Guid+服务器网卡 ID+时间，自己生成有序的 Guid。

对 Guid 类型对碎片的影响，同样举以下几个例子。

首先，对比 int 和 Guid 对碎片影响，插入 10 万条数据后，碎片情况如表 3-1 所示。

表 3-1　磁盘碎片情况（1）

	表名	表数据量	索引名	索引类型	页填充度	索引碎片	页数
☑	T1	99999	I_ID	NONCLUSTERED INDEX	65.899629355...	99.011857707...	506
☑	T2	99999	I_ID	NONCLUSTERED INDEX	99.610032122...	2.1505376344...	186

Guid 类型的索引碎片约为 99%，而 int 类型的索引碎片仅为 2%左右。

再对比一下有序生成 Guid 和无序生成 Guid 对碎片的影响。

同样，插入 10 万条数据，碎片情况如表 3-2 所示。

表 3-2　磁盘碎片情况（2）

	表名	表数据量	索引名	索引类型	页填充度	索引碎片	页数
☑	T1	19998	I_ID	NONCLUSTERED INDEX	68.470471954...	99.009900990...	101
☑	T2	19998	I_ID	NONCLUSTERED INDEX	97.412107734...	52.112676056...	71

执行随机插入操作后，表 T1 和 T2 的碎片情况有了明显的差异，有序 Guid 索引产生的碎片大概是无序 Guid 索引碎片的一半。

在数据空间占用方面，采用有序 Guid 方式仅占用了 71 页，而无序 Guid 方式占用了 101 页。

（4）尽量用系统的基本类型表示相应的业务字段类型

有时候为了表示统一，一些业务字段不用系统基本类型表示。例如，某个产品中没有用 DateTime 基本日期类型字段表示日期，而是用了 char(20)表示日期，其实验数据如表 3-3 所示。

表 3-3　实验数据

	表名	表数据量	索引名	索引类型	页填充度	索引碎片	页数
☑	T2	99999	I_CreateTime2	CLUSTERED INDEX	61.795997034...	69.292742050...	14026
☑	T1	99999	I_CreateTime2	CLUSTERED INDEX	94.561984185...	0.6159938400...	9091

顺序插入 10 万条数据后，聚集索引列类型为 char(19)的索引碎片增加到 69%；而 DateTime 列的索引并没有增加，说明 char(19)类型列作为索引会导致大量碎片的产生。

因此要尽量用系统的基本类型表示相应的业务字段类型。

（5）数据库碎片归纳总结

顺序插入数据时，一般数据类型，如 int, DateTime 等不会产生碎片；Char(19)这样的类型会产生较多的碎片；Guid 类型更容易产生碎片，除了长度，更主要的原因是它产生值的随机性；索引列字段占字节越大，产生的索引碎片越大。

随机插入数据时，所有数据类型都易产生碎片；char(19)数据字段索引比基本类型（int、DateTime 等）更易产生碎片；聚集索引比非聚集索引更易产生碎片。

随机更新时，更新聚集索引列消耗的时间更长；无论是聚集表还是堆表随机更新比随机插入产生的碎片更多；更新聚集索引比更新非聚集索引代价更大，因为更新聚集索引，会对所有非聚集索引进行更新，而更新非聚集索引则仅对这一个索引进行修改。

执行删除操作对碎片影响非常小，基本可以忽略不计。

2．索引优化

（1）索引创建规则

- ◆ 为经常被用于查询的谓词创建索引，如下拉参照中被用于快速查找的 code 和 name 等。在平台现有下拉参照的查询 SQL 语句中，like 的条件语句要取消前置通配符。还需要关注 Order By 和 Group By 谓词的索引设计，Order By 和 Group By 的谓词是需要排序的，某些情况下为 Order By 和 Group By 的谓词建立索引，会忽略查询时的排序动作。
- ◆ 对于内容存在大量重复的列，比如只包含 1 和 0，应禁止建立索引。因为该索引选择性极差，在特定状况下会误导优化器做出错误的选择，导致查询速度大幅下降。
- ◆ 当一个索引由多个列构成时，应注意将选择性强的列放在前面。前后次序稍有不同，性能就可能出现数量级的差异。
- ◆ 对小表进行索引有时并不能产生优化效果，因为查询优化器在遍历用于搜索数据的索引时，花费的时间可能比扫描简单的表还长，因此设计索引时需要考虑表的大小。对于记录数不大于 100 的表禁止建立索引，频繁操作的小数据量表不建议建立索引（记录数不大于 3000）。

（2）聚集索引创建的必要性

实验表明，一旦创建聚集索引后再删除聚集索引，查询计划会由聚集搜索变为原来的表扫描，但扫描成本仍然为 15%，而不是之前的 41%。换句话说，由于聚集索引能够保证数据物理存储是连续的，所以一般查找大数据量时会比较快，磁盘轴每次物理寻道（移动）或逻辑 lookup/rid 操作时间较短；如果表中没有聚集索引，则可能数据是分散存储的，用一般的表扫描会使磁盘轴每次寻道（移动）或逻辑 lookup/rid 操作的时间比较长，整个表扫描时间就会随之增长很多。

聚集索引可以避免表中产生数据碎片，建议在表中保留一个聚集索引。产生数据碎片的主要原因是表中缺少聚集索引，如果表中含有聚集索引可以使数据在任何更新操作（插入/删除）后自动移动数据以保证数据的物理连续，很少会产生数据存储间的空闲碎片。因为聚集索引结构中的叶索引就是数据位置，而这些是非聚集索引做不到的。解决办法是，要在表中

保留一个聚集索引，这样在更新（插入/修改/删除）时就可以尽量避免数据碎片的产生，而不是等到账套大了再缩小。那些不含有聚集索引且插入更新频繁的表是最容易导致体积变大的。建议使用 CreateTime，用户一般是按时间插入单据的，按索引排序原则会插入到数据最后，这样也不会导致数据匹配移动；尽量不要用 ID(Guid)，由于 Guid 值的大小具有随机性，这样的字段建立聚集索引经常会发生即使一个很小的插入也会导致大量批量数据移动的情况。用 CreateTime 作聚集索引也便于在日后数据量很大时进行数据分区。

最重要一点，虽然大部分时候聚集索引扫描与表扫描的查询速度是一样的，但聚集索引查找是所有操作中最快的，比非聚集索引查找还快。

（3）索引类型的选择

在不同情况下，该创建聚集索引还是非聚集索引，如图 3-1 所示。

动作描述	使用聚集索引	使用非聚集索引
列经常被分组排序	√	√
返回某范围内的数据	√	×
一个或极少不同值	×	×
小数目的不同值	√	×
大数目的不同值	×	√
频繁更新的列	×	√
外键列	√	√
主键列	√	√
频繁修改索引列	×	√

图 3-1　创建索引类型建议

3．避免阻塞设计原则

（1）整体原则

总结起来性能优化也依赖于代码和 SQL 的优化，逻辑代码要优化到最快，否则一个耗时较长的 SQL 语句将会阻塞全部用户等待几十秒甚至几分钟，所以针对查询 SQL 语句的优化是最重要的。

修改批量操作的需求，产品中单记录操作的消耗不小，批量操作耗时和记录数量成正比。因此，设计时要避免在一个自动事务服务方法中做批量的循环操作，这里可以将循环操作放到 UI 控制器端，这样就使一个长事务变成多个短小的事务，减少阻塞机会。

减少写操作使用锁的数量，比如减少批量更新操作中修改行的数量，使行锁定减少，同时减少锁升级至表锁的概率。

一些耗时长、锁定数据多的操作需要避免和正常业务操作冲突，可以使用调度计划在系统闲置的时间内运行，或者使用互斥机制来保证其他用户暂时退出操作以独立运行，具体视业务情况而定。

（2）处理原则

1）减少读操作需要的共享锁

a）将事务隔离级别由默认读提交（ReadCommited）修改为读未提交（ReadUn Commited），这样将不会有读/写阻塞，但是会造成读取其他事务未提交的数据。指定 Isolation Level 属性为 ReadUn Commited。

b）在 SELECT 语句中带 NoLock 提示，事务内无锁提示也不会有读/写阻塞，与上面一样也会有脏读。SELECT 上加无锁控制，在执行 SELECT 操作时加 with(NoLock)，如：SELECT * FROM person WITH(NoLock)。

这两种方式都会出现脏读，对数据要求非常严格的场景要慎用。

2）在 SQL Server 中使用基于行版本的快照隔离模式

从 SQL Server 2005 版本开始，数据库支持行版本快照隔离功能，主要是用于解决读/写阻塞问题，提供了以下两种方式。

◆ 方式一：已提交读快照打开

ALTER DATABASE 数据库名 SET READ_COMMITTED_SNAPSHOT ON;

◆ 方式二：快照隔离打开

ALTER DATABASE 数据库名 SET ALLOW_SNAPSHOT_ISOLATION ON;

[SET TRANSACTION ISOLATION LEVEL SNAPSHOT]

方式一和方式二在使用时略有不同。方式二除了要打开快照隔离，在每个事务中使用时都要配合 SET TRANSACTION ISOLATION LEVEL SNAPSHOT 子句使用才能生效；而方式一只要打开快照即可生效。方式一主要是针对物理对象（如表、行、键、索引等），方式二针对逻辑事务，所以方式一控制粒度更小。

4．死锁最小化原则

（1）死锁产生原因

死锁是由两个互相阻塞的线程组成的，它们互相等待对方完成。一般死锁情况下，两个数据库事务之间存在着反向的操作。SQL Server 中死锁监视器定期检查死锁，如果发现死锁，将选择其中回滚消耗最小的任务，这时易发生 1205 数据库错误。

SQL Server 有几十种锁，以下是两个最典型的：

共享锁（S 锁）：如果事务 T 对数据 A 加上共享锁，则其他事务只能对 A 再加共享锁，不能加排他锁。获准共享锁的事务只能读取数据，不能修改数据。

排他锁（X 锁）：如果事务 T 对数据 A 加上排他锁，则其他事务不能再对 A 加任何类型的锁。获准排他锁的事务既能读取数据，又能修改数据。

（2）死锁最小化

1）写/写死锁

用相同的顺序访问对象，如果涉及多于一张表的操作，要保证事务中都按照相同的顺序访问这些表。

另外，要减少一个事务中大批量更新的操作，大批量的写操作涉及记录独占锁太多且一直到事务结束才能释放，更容易与其他事务造成死锁。

2）读/写死锁

去掉读操作共享锁的最佳方式是使用 SQL Server 的快照模式，其次是使用读取未提交隔离模式或使用 with no lock 提示。

5．高性能 SQL 建议

（1）确保查询索引有效

1）索引使用原则

创建了索引，并不能确保 SQL 语句在执行时一定用到。以下是能有效使用索引的条件

语句：

```
[fieldname]=8
[fieldname]>200
[fieldname] between 0 and 100
[fieldname] like 'abc%'
```

以下是不能有效利用索引的一些语句：

```
SUM（[fieldname]）=8
[fieldname] + 1 =8
[fieldname] like '%abc%'
```

2）避免对表达式左侧索引字段进行运算或函数处理

低效的写法为：

```
select name from table where substring(name,1,3) = 'king'
```

高效的写法为：

```
select name from table where name like 'king%'
```

3）复合索引使用原则

- 选择性强的条件放在前面。
- 尽量用复合索引，速度快而且不会导致表中索引个数太多，还能节省空间。

（2）高效率 SQL 语句

1）查询时不返回不需要的行和列

在查询时，尽量使用通配符，如 SELECT * FROM T1 语句，要用到几列就选择几列，如 SELECT C1,C2 FROM T1。在可能的情况下尽量限制结果集行数，如 SELECT TOP 100 C1,C2,C3 FROM T1，因为某些情况下用户是不需要那么多数据的。

2）合理使用 EXISTS,NOT EXISTS 子句

```
SELECT SUM(T1.C1)FROM T1 WHERE((SELECT COUNT(*) FROM T2 WHERE T2.C2=T1.C2>0)
SELECT SUM(T1.C1) FROM T1 WHERE EXISTS(SELECT * FROM T2 WHERE T2.C2=T1.C2)
```

以上两种写法的结果相同，但是后者的效率显然要高于前者。因为后者不会产生大量锁定的表扫描或索引扫描。

还有以下几个 SQL 语句：

```
SELECT a.hdr_key    FROM hdr_tbl a            //tbl a  表示 tbl 用别名 a 代替
WHERE NOT EXISTS (SELECT * FROM dtl_tbl b WHERE a.hdr_key = b.hdr_key)

SELECT a.hdr_key    FROM hdr_tbl a
LEFT JOIN dtl_tbl b ON a.hdr_key = b.hdr_key    WHERE b.hdr_key IS NULL

SELECT hdr_key    FROM hdr_tbl
WHERE hdr_key NOT IN (SELECT hdr_key FROM dtl_tbl)
```

以上三种写法都可以得到同样正确的结果，但是效率依次降低。

3）最大化使用可能的查询条件

例如，以下两个 SQL 中第二句很可能比第一句执行快得多。

```
SELECT SUM(A.Money)
FROM T1 A left join T2 B on A.ID = B.ID

SELECT SUM(A.Money)
FROM T1 A left join T2 B on A.ID = B.ID    AND
    A.CODE=B.CODE
```

4）使用 UNION ALL 代替 UNION

合并两个结果集时，尽量使用 UNION ALL，因为它在执行的时候不会进行“确定唯一排序”，而 UNION 则要排序。

如果对结果集不需要排序，或者数据肯定是唯一的，则要用 UNION ALL 代替 UNION 合并运算符。SQL 语句如下：

```
select c1, c2, c3 from t1 where code = '001'
union all
select c1, c2, c3 from t1 where code = '002'
```

肯定要比以下 SQL 语句高效。

```
select c1, c2, c3 from t1 where code = '001'
union
select c1, c2, c3 from t1 where code = '002'
```

3.2.2 客户端代码编码规范

下面这些规范以 JavaScript 语言为例进行说明。

1．预先计算 elements 的长度

JavaScript 与服务端代码一样，不要重复计算当前遍历集合元素的长度。

低效的写法如图 3-2 所示：

高效的写法如图 3-3 所示：

```
function C1 ()
{
    var ements = document.getElementsByTagName("SPAN") :
    var index = 0:
    while(index < 1000 )
    {
        for ( var i = 0: i < ements.length:  i++ )
        {
            var a = i.toString():
        }
        index++:
    }
}
```

图 3-2　低效写法

```
function C2 ()
{
    var ements = document.getElementsByTagName("SPAN") :
    var index = 0:
    while(index < 1000)
    {
        for ( var i = 0, k=ements.length:i<k: i++)
        {
            var a = i.toString():
        }
        index++:
    }
}
```

图 3-3　高效写法

分别用两种方式访问 collection.length 属性，实验统计结果如图 3-4 所示。

函数	计数	包含时间(毫秒)
C1	1	124.99
C2	1	46.87

图 3-4　统计结果

2．高效率变量的定义方式

低效的写法为：

```
function Compute()
{
    var   i;
    var   j;
    var   k;
    for(i=0; i < X.Length; i++)
        for(j=0; j < Y.Length; j++)
            for(k=0; k < Z.Length; k++)
    {
            ……（省略完整代码）
    }
}
```

高效的写法为：

```
function Compute()
{
    for(var i=0; i < X.Length; i++)
        for(var j=0; j < Y.Length; j++)
            for(var k=0; k < Z.Length; k++)
    {
            ……（省略完整代码）
    }
}
```

再如，低效的写法为：

```
p = new Person();
p.Name = "King";
p.Age = "20";
p.Sex = "male";
```

高效的写法为：

```
p = { Name: "King", Age: "20", Sex: "male" }
```

JavaScript 是解释执行语言，并且每条代码都会作为一个单位进行提交执行，因此要减少提交的次数，尽量用最少的语句实现相同的功能。

3．对象缓存优化

由于 JavaScript 的解释性，a.b.c.d.e 需要进行至少 4 次查询操作，先检查 a 再检查 a 中的 b，再检查 b 中的 c，等等。

请看图 3-5 中的示例，这种写法每次访问元素都要挂上“document.all…”。如图 3-6 所示为高效率写法，这种写法是先把 document.all 和 document.all.length 存储到变量中，使用时直接访问变量，避免再次查询的过程。

```
function Q1()
{
    var n=0;
    while(3000)
    {
        for(Var i=0:i< document.all.length; i++)
        {
            if( document.all(i).tagName ==="SPAN")
            {
                a+= document.all(i).id ;
            }
        }
        n++;
        if(n>10000)break;
    }
}
```

图 3-5　低效写法

```
function Q2()
{
    var n=0;
    while(3000)
    {
        var all=document.all;
        var len=all.length;
        for(var i=0;i<len;i++)
        {
            if( all(i).tagName ==="SPAN")
            {
                a+= all(i).id ;
            }
        }
        n++;
        if(n>10000)break;
    }
}
```

图 3-6　高效写法

执行 document.all 检索大约 15 万次，两种方式时间消耗差 3 倍，实验统计数据如图 3-7 所示。

Q1	1		15,073.97
Q2	1		5,128.90

图 3-7　统计结果

4．字符串连接高效率写法

如果要连接多个字符串，应该使用“+”，如：

```
s+=a;
s+=b;
s+=c;
```

应该写成：

```
s+=a + b + c;
```

连接大量的字符串，应使用 Array 的 join 方法。

如果是收集字符串，最好使用 JavaScript 数组缓存，最后使用 join 方法连接起来，如下：

```
var array = new Array();
for(var I = 0; I < 100; i++)
{
    array.push(i.toString());
}
var result = array.join("");
```

看一个实验例子，低效的写法如图 3-8 所示，高效的写法如图 3-9 所示。

两种字符串连接方式各执行 1000000 次，时间统计数据如图 3-10 所示。

两种写法时间相差 100ms 左右。

```
function AppendString1 ()
{
    var a="ABCDEFGHIJKLMNOPQRSTUVWXYZZ1";
    var b="ABCDEFGHIJKLMNOPQRSTUVWXYZZ2";
    var c="ABCDEFGHIJKLMNOPQRSTUVWXYZZ2";
    var s2;
    for(var i=0,i<1000000,i++)
    {
        s2+=a;
        s2+=b;
        s2+=c;
    }
}
```

图 3-8　低效写法

```
function AppendString2 ()
{
    var a="ABCDEFGHIJKLMNOPQRSTUVWXYZZ4";
    var b="ABCDEFGHIJKLMNOPQRSTUVWXYZZ5";
    var c="ABCDEFGHIJKLMNOPQRSTUVWXYZZ6";

    var s3;
    for(var i=0;i<1000000;i++)
    {
        s3+=a+b+c;
    }
}
```

图 3-9　高效写法

函数	计数	包含时间 (毫秒)
AppendString1	1	1,312.47
AppendString2	1	1,249.97

图 3-10　统计结果

再看一个实验，高效的写法如图 3-11 所示，低效的写法如图 3-12 所示。

```
function AppendBigString1()
{
    var buffer=new Array();
    for(var i=0;i<100000;i++)
    {
        buffer.push(i);
    }
    var s=buffer.join('-');
    //alert(s);
}
```

图 3-11　高效写法

```
function AppendBigString2()
{
    var buffer=new Array();
    var s;
    for (var i=0;i<100000;i++)
    {
        s+=i.toString()+'-';
    }
    //alert(s);
}
```

图 3-12　低效写法

两种方式（使用 Array.join 和串连接符）各自执行 100000 次，时间统计数据如图 3-13 所示。

函数	计数	包含时间 (毫秒)
AppendBigString1	1	125.00
AppendBigString2	1	187.50

图 3-13　统计结果

5．类型转换函数

使用 Math.floor()或者 Math.round()将浮点数转换成整型。

浮点数转换成整型很容易出错，很多人喜欢使用 parseInt()，其实 parseInt()适用于将字符串转换成数字，而不是浮点数和整型之间的转换。如果是纯数值之间的转换，应该使用 Math.floor() 或者 Math.round()。

再看下面的实验数据，低效的写法如图 3-14 所示，高效的写法如图 3-15 所示。

```
function ChangeToInt1()
{
    var r;
    for(var i=0;I<100000;i++)
    {
        r= parseInt(i);
    }
}
```

图 3-14　低效写法

```
function ChangeToInt2()
{
    var r;
    for(var i=0;I<100000;i++)
    {
        r= Math.floor(i);
    }
}
```

图 3-15　高效写法

使用 parseInt()和 Math.floor()两种方法分别转换 100000 次，统计数据如图 3-16 所示

函数	计数	包含时间(毫秒)
ChangeToInt1	1	93.75
ChangeToInt2	1	62.50

图 3-16　统计结果

6．避免闭包写法

使用 prototype 代替 closure，可以有效地避免闭包。使用 closure 在性能和内存消耗上都是不利的，如果 closure 使用量过大，这就会成为一个问题。所以，尽量将：

```
this.method1 = function()
{
        ……（省略完整代码）
}
```

替换成：

```
Class1.protoype.methodFoo = function()
{
      ……（省略完整代码）
}
```

和 closure 存在于对象实例之中不同，prototype 存在于类中，被该类的所有的对象实例共享，可以代码重用。

另外，闭包在有些浏览器（或一些浏览器版本）中会出现内存泄漏，需要特别注意。

7．高效操作 DOM——离线操作

在添加一个复杂的 DOM 树时，可以先构造，构造结束后再将其添加到 DOM 树的适当节点；如果操作已经填加到节点的 DOM，可以先移除(脱离)，构造完成后再添加到节点上。

看一下以下代码：

```
var u1 = document.getElementById("ul");
for(var i=0; i < 100; i++)
{
      ul.appendChild(document.createElement("li"));
}
```

下面这种写法会更高效：

```
var u1 = document.getElementById("ul");
var li = document. createElement ("li");
var parent = ul.parentNode;
parent.removeChild(ul);
for(var i=0; i<100; i++)
{
      ul.appendChild(li.cloneNode(true));
}
parent.appendChild(ul);
```

同样做了一个实验，数据如下。一般写法如图 3-17 所示，更高效的写法如图 3-18 所示。

```
function CreateDom1()
{
    var divParent=document.createElement("DIV");
    document.body.appendChild(divParent);
    for(var;=0;I<10000;i++)
    {
        var span=document.createElement("SPAN");
        divParent.appendChild(span);
    }
}
```

图 3-17　低效写法

```
function CreateDom2 ()
{
    var divParent=document.createElement("DIV");
    for(var;=0;I<10000;i++)
    {
        var span=document.createElement("SPAN");
        divParent.appendChild(span);
    }
}   document.body.appendChild(divParent);
```

图 3-18　高效写法

构造完后再追加到 DOM 元素上，防止单次构造操作导致浏览器回流操作。

在上述实验中，用两种方式分别创建 10000 个 SPAN，并追加到 body 中，执行时间比较如图 3-19 所示。

函数	计数	包含时间 (毫秒)
CreateDom1	1	234.37
CreateDom2	1	203.12

图 3-19　统计结果

3.2.3　服务器代码编码规范

本节规范以 C#语言为例进行说明。

1．预先计算 collection 的 length

不要在循环语句中循环计算当前集合遍历长度。

如将

```
for(var i=0; i < collection.length; i++)
{
        ……（省略完整代码）
wq}
```

替换成

```
for(var i=0; len=collection.length; i<len; i++)
    {
        ……（省略完整代码）
    }
```

效果会更好，尤其是在大循环中。

2．避免循环体内重复创建对象

避免循环体内重复创建对象的代码如下。

```
List< BusinessObj > Compute(Collection sets)
{
    List< BusinessObj > list = new List<Results>();
    foreach(RelateObj obj in set.Length)
    {
```

```
            Creater creater = new Creater();
            BusinessObj obj = creater.Create(obj);
            list.Add(obj);
        }
        return list;
    }
```

高效的做法是将 builder 对象提到循环外面创建。

3．优先使用系统常量

看一下这段代码有没有优化的余地？

```
If(object1.Amount == new decimal(0))
{
    Object1.Status = AmountStatus.None;
}
```

正确的做法是使用 Decimal.Zero 常量，使用常量避免创建对象。

4．使用 StringBuilder 做字符串连接

第 1 章中第二个案例就是因为“+”（String.Concat 方法）连接导致了 GC 99% CPU 利用率。

当连接字符串个数超过 10 个时要用 StringBuilder，如 StringBuilder sb = new StringBuilder **(256)**。

另外，后面的参数为 256，StringBuilder 类默认申请 Buffer 的长度为 16，一旦超出这个范围 Buffer 就会重新分配，这比较消耗资源。在我们已知要连接字符串长度的情况下，要传一个初始值参数，这样首次就会分配指定的空间大小。

5．避免不必要的调用 ToUpper 或 ToLower 方法

```
Pubic bool GetTrueOrFalse
{
    Get
    {
        return bool.Parse(object1.ToString().ToLower());
    }
}
```

这里的 ToLower 方法完全没必要。

6．用 String.Compare 方法代替 ToLower 和==运算

请看如下代码：

```
Foreach(XmlElements element in a.Elements)
{
    If(element.Value.ToLower()== "a")
        {
            … …
        }
        If(element.Value.ToLower()== "b")
        {
            … …
```

```
        }
        If(element.Value.ToLower()== "c")
        {
            ……（省略完整代码）
        }
    }
```

高效的做法是使用 Compare 方法，这个方法可以做大小写忽略的比较，并且不会创建新字符串，如下：

```
If(String.compare(element.Value, "a", StringComparison.OrdinalIgnoreCase) == 0)
```

7．多线程代码优化

（1）线程同步：同步方法

以下是多线程控制同步的实现方法。

```
[MethodImpl(MethodImplOptions.Synchronized)]
public static PerformanManager GetInstance()
{
    if(instance == null)
    {
        PM pm    = new PerfmonManager();
    }
    return pm;
}
```

请注意：如果能控制同步语句尽量同步语句，并且遵循同步粒度最小原则。

（2）线程同步：同步类型

锁定 Type 对象会影响同一进程中所有应用程序域的所有实例，这不仅可能导致严重的性能问题，也会导致各种错误的发生。应额外构造一个 static 的成员变量，让此类成员变量作为锁定对象。

```
public static PerformanManager GetInstance()
{
    if(instance == null)
    {
        lock(typeof(Manager))
        {
            PM pm    = new PerfmonManager();
        }
    }
    return pm;
}
```

（3）线程同步：同步 this 对象

要避免锁定 this 对象，因为锁定 this 会影响该实例的所有方法。假设对象 obj 有 A 和 B 两个方法，其中 A 方法使用 lock(this)对方法中的某段代码设置同步保护。现在，因为某种原

因，B 方法也开始使用 lock(this)来设置同步保护，并且可能为了完全不同的目的。这样，A 方法就被干扰了，其行为可能无法预知。所以，基于良好的习惯，建议避免使用 lock(this)这种方式。

（4）线程同步：同步策略

采用静态对象的方式实现同步，代码如下。

```
public class UserOnlineStatusManager
{
    private Hashtable htOnlineUsers = null;
    private static Object _lockObject = new Object();
    public static UserOnlineStatusManager GetInstance()
        {
            lock(_lockObject)
            {
                if (instance == null)
                    {
                        instance = new UserOnlineStatusManager();
                    }
                }
            }
            return instance;
        }
    }
```

这里把同步对象_lockObject 定义为静态的原因在于，所有线程都要能够对同步对象_lockObject 进行访问才能实现同步功能，也就是_lockObject 对象起到线程之间数据传递作用，而线程之间共享数据常用的方法之一就是通过静态变量，如果定义成非静态变量则每个线程都会有自己的一个_lockObject，也就无法实现同步功能。

（5）线程同步：Double Check 机制

以上代码其实还不够严谨，在某些情况下还是会出现问题。

一般在 lock 语句后就会直接创建对象了，但这不够安全。因为在 lock 锁定对象之前，可能已经有多个线程进入第一个 if 语句中了。如果不加第二个 if 语句，则单例对象会被重复创建，新的实例替代旧的实例。如果单例对象中已有数据不允许被破坏或者存在其他原因，则应考虑使用 Double Check 技术。修正后的代码如下：

```
public class UserOnlineStatusManager
{
    private Hashtable htOnlineUsers = null;
    private static UserOnlineStatusManager instance = null;
    public static UserOnlineStatusManager GetInstance()
        {
            if (instance == null)
            {
                lock(_lockObject)
                {
```

```
                    if (instance == null)
                    {
                        instance = new UserOnlineStatusManager();
                    }
                }
            }
        return instance;
    }
}
```

8．使用泛型集合避免装箱/拆箱

这里直接看实例。

低效的写法如图 3-20 所示。

高效的写法如图 3-21 所示。

```
private void ChangedTypel()
{
    ArrayList arr=new ArrayList();
    for(int i=0;i<10000;i++)
    {
        arr.Add(i);
    }
    int j=0;
    for(int i=0;i<10000;i++)
    {
        j=(int)arr[i];
    }
}
```

图 3-20　低效写法

```
private void ChangedType2()
{
    List<int>list=new List<int>();
    for(int i=0;i<10000;i++)
    {
        list.Add(i);
    }
    int j=0;
    for(int i=0;i<10000;i++)
    {
        j=list[i];
    }

}
```

图 3-21　高效写法

分别用非泛型集合和泛型集合对 int 类型的数据各装箱和拆箱 10000 次，得出的比较数据（ms）如图 3-22 所示。

_Default.ChangedType1()		5.39
_Default.ChangedType2()		2.55

图 3-22　统计结果

9．避免两次检索集合元素

获取集合元素时，有时需要检查元素是否存在。通常的做法是先调用 ContainsKey（或 Contains），然后再获取集合元素，这种写法非常符合逻辑。

以下这段代码有没有优化的余地？

```
if(HashTable.Exist(Key))
{
    return HashTable[key];
}
```

如果考虑效率，可以先直接获取对象，然后判断对象是否为 null 来确定元素是否存在。对于 Hashtable，则可以节省一次 GetHashCode 调用和 n 次 Equals 比较，避免两次检索集合元素。如下：

```
object value = HashTable[key];
if(value != null)
{
        ……（省略完整代码）//code logic
}
```

10．避免两次类型转换

请看一下以下这段代码有没有优化的余地？

```
if(obj is Employee)
{
    Employee employee = (Employee)obj;
    employee.Do();
}
```

效率更高的写法如下（避免两次类型转换）：

```
Employee employee = obj as Employee;
if(employee    != null)
{
    employee.Do();
}
```