第1章

性能优化思路

本章内容

- 内存性能优化案例
- CPU 占用 100%分析案例
- 性能优化理论体系

1.1 两个优化实战案例

在本书的第一章，我并不喜欢先说一些理论上的东西，这样会让读者感到乏味，本书先以两个真实的性能问题案例开篇，让读者了解一下定位一个性能问题的过程，或许这样会更有趣。

1.1.1 内存性能问题案例

1．客户问题描述

客户反馈查询报表速度太慢，要十几分钟，而且是所有报表查询都慢，其他一些轻量级操作正常。客户服务器配置如表 1-1 所示。

表 1-1　客户服务器配置

硬 件 类 型	参　数　值
CPU	至强 8 核
内存	32GB
磁盘	SAS(> 10000RPM)
操作系统	Microsoft windows server 2003 sp2, 32 位
数据库	Microsoft SQL Server 2005 sp2, 32 位

2．诊断分析定位原因

根据客户的反馈及服务器环境，提取出以下三个线索：

- 只有报表查询慢，而报表查询对各硬件资源消耗比较大。
- 客户服务器当前环境安装的是 32 位的操作系统及 32 位的数据库版本。
- 客户服务器配置了 32GB 的内存。

首先会想到的是内存问题，先从内存入手解决。

用 Windows Performance 计数器对运行中的服务器做了日志跟踪，拿到本地并打开，如图 1-1 所示。

图 1-1　服务器日志跟踪

数据库目标内存（Target Server Memory）只有 1.6GB，指的是操作系统当前分配给数据库可用的最大内存（即使设置了 32GB，如果有内存不够或权限问题等原因，数据库也不会用到 32GB 内存，后面会说明）。当前总共使用内存（Total Server Memory）指的是当前数据库

已经使用了多少内存，本例监控到的值也是 1.6GB，也就是说已经用完了当前所有目标内存。

另外，数据库设置的最大内存是 30GB（并打开 AWE 功能），如图 1-2 所示。

图 1-2　内存设置

客户给 SQL Server 设定的最大使用内存是 30GB，并且开启了 AWE 功能，但为什么内存实际只用了 1GB 呢？

这是因为：虽然客户设置了 30GB，并且设置了 AWE，但这里的 AWE 并没有生效。

操作系统和数据库都安装的是 32 位的版本，尽管配置了 32GB 内存，但是 32 位操作系统最多只能识别 4GB 的内存，所以内存使用受到限制。在这 4GB 内存中，默认情况下操作系统会留 2GB 给自己（内核）用，剩下的 2GB 内存会给其他所有应用程序用，也就是说 SQL Server 不可能使用超过 2GB 的内存，跟之前客户服务器上内存使用的 1.6GB 相吻合。即使打开 3GB 开关，也仅有 3GB 给所有应用程序使用，32GB 物理内存也浪费了。

最佳解决方案：

以下经验仅针对数据库服务器内存配置建议：

■ 内存大于 4GB

对于内存超过 4GB 的服务器，建议安装 64 位的操作系统和 64 位的数据库版本，这样可以避免 32 位版本的 4GB 内存限制，也不需要做本节中的内存设置优化（PAE/AWE/内存锁定页权限）。

这个建议仅在开始部署环境时用得多，对当前已经上线的系统，则用得最多的还是进行内存设置优化（PAE/AWE/内存锁定页权限）。

■ 内存小于或等于 4GB

对于内存等于或小于 4GB 的物理内存配置，建议安装 32 位的操作系统和数据库版本并进行内存设置优化（PAE/AWE/内存锁定页权限）。64 位的操作系统下所有程序比较耗内存，4GB 物理内存有点小。比如数据库服务器在 32 位环境下并配置 4GB 物理内存，如果未开启 PAE 和 AWE，数据库最大会分配到 1.6GB 左右内存；如果开启了 PAE 和 AWE，则数据库使用内存大概在 2.8GB 内存，64 位操作系统下不会达到 2.8GB。

继续回到客户问题。由于当前系统已经上线，并且客户正在使用，重装系统和数据库环境可能不太现实，最佳方案是进行内存设置优化（PAE/AWE/内存锁定页权限）。具体步骤如下：

第一步，开启操作系统 PAE（物理扩展内存），配置系统盘下的 boot.int 文件。

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(2)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(2)\WINDOWS="Windows Server 2003, Enterprise" /fastdetect /PAE
```

直接在 boot.ini 文件中增加 /PAE 参数。

如果是 Windows7，Windows Server 2008 及更高版本的操作系统，操作系统提供了专门的 BCDEdit 命令：

```
BCDEdit /set PAE forceenable
```

第二步，开启数据库的 AWE（Address Windowing Extensions）动态分配映射内存，勾选“使用 AWE 分配内存”，并设置最大服务器内存为 30720（30GB），如图 1-3 所示。

图 1-3　设置最大内存

第三步，设置内存锁定页权限。首先确定一下 SQL Server 当前运行账户，如图 1-4 所示。

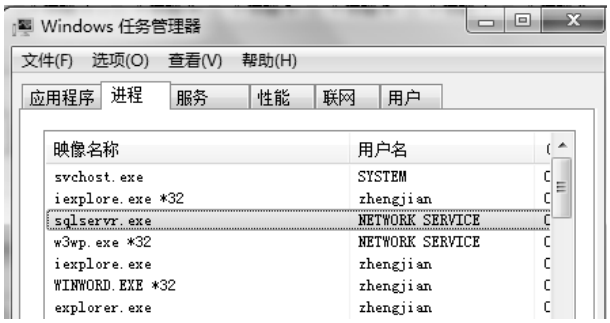

图 1-4　任务管理器

当前运行账户为：NETWORK SERVICE 网络用户，后面步骤就为此用户赋予内存锁定页权限。

运行 gpedit.msc 命令，如图 1-5 所示，打开“本地组策略编辑器”。

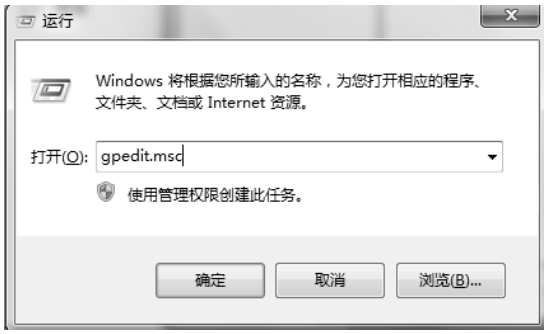

图 1-5　运行窗口

在“本地组策略编辑器”中依次展开左侧目录结点，找到“用户权限分配”结点，如图 1-6 所示。

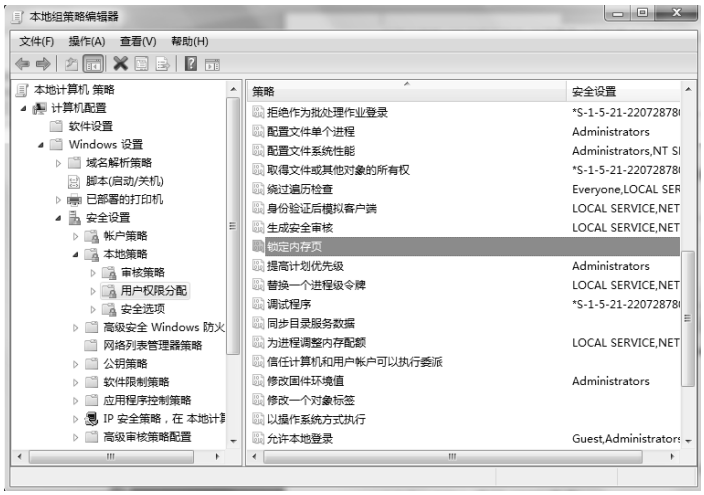

图 1-6　设置锁定内存页

右击“锁定内存页”，单击“属性”选项，打开“锁定内存页 属性”窗口，如图 1-7 所示。

图 1-7　添加锁定内存页权限账户

把 SQL Server 当前登录用户 NETWORK SERVICE 添加进来，单击“确定”按钮。

注意：这三个步骤缺一不可，设置好后要重启一下计算机。

经过此设置后，几分钟后再看客户服务器的计数器，如图 1-8 所示。

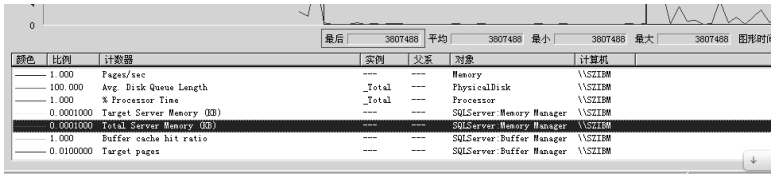

图 1-8　查看最大内存

目标内存已经由 1.6GB 变为 30GB，当前使用内存也由原来的最大使用 1.6GB 变为 3.8GB，接下来这个值还会增加，上限是目标内存最大值。

客户报表查询速度也非常快了，整体上操作都响应非常快了。问题原因主要在于客户配置了 32GB 物理内存，但数据库只使用了 1.6GB。

问题思路：

第一步，先确定环境问题（操作系统、数据库配置等）。

第二步，如果环境没问题，再进行报表服务端代码或报表查询 SQL 跟踪。

不要直接进入第二步，这样分析方向就错了。一个性能专业人员，除了要掌握专业的解决问题技能外，积累经验也非常重要。

下面是一个更有趣的案例。

1.1.2　CPU 占用 100%分析案例

1．客户问题描述

200 人并发使用某服务器，使用中出现所有客户端卡死，服务器无法接收客户端任何请求。

客户还提供了一个线索：此时服务器 CPU 利用率接近 100%。

2．定位分析

这台服务器运行着 ERP 系统，主要承载 Web 服务器，根据客户提供的线索很可能是 CPU 利用率 100%导致服务器繁忙，而不能及时响应所有客户端的请求，出现所有客户端“假死”现象，这是很常见的问题。

往往很多非专业计算机人员遇到这种问题就重启一下 Web 服务器或客户服务器，继续使用。这样既解决不了问题，又会丢失线索，而且问题还会重复出现。

先看一下服务器计数器，计数器中显示“% Time in GC”为 CPU 利用率 90%，如图 1-9 所示。

图 1-9　垃圾回收线程 CPU 利用率

说明：一般 % Time in GC > 10%，基本上就应该检查代码了，而这里达到 90%。

现象已经基本明确，是由于 w3wp 中的 GC 线程不断地在做垃圾回收工作，耗尽 CPU 资源，导致服务器不能处理其他客户端客户发来的请求。

到这里只是笼统的分析，还不能确定是什么问题，更不能做任何结论，要确定是什么导致 GC 这么忙碌才是最终目的。一般涉及 GC 问题都是服务端代码写法不正确导致的，找到代码并修改才是根本。下面就分析一下是哪句代码出的问题。

这里使用 WinDbg 工具从 webserver 的进程中寻找线索。WinDbg 是微软内部用来调试操作系统 bug 的一个工具，当然也能够调试应用程序软件。如果读者对这个工具不熟悉也没有关系，这里只是说明一下分析思路，这一章中还不需要对每个分析点具体了解。

接到客户问题后，使用 WinDbg 对服务器进程 w3wp.exe dump 了一个文件，把 dump 文件拿回本地分析，重启一下服务器，客户可继续使用系统。

Dump 文件的过程是把应用运行中某一时刻的运行信息及状态写到文件中，查看一下线程池：

```
0:025> !threadpool
CPU utilization: 99%
Worker Thread: Total: 47 Running: 5 Idle: 42 MaxLimit: 800 MinLimit: 8
Work Request in Queue: 0
--------------------------------------
Number of Timers: 46
--------------------------------------
Completion Port Thread:Total: 2 Free: 2 MaxFree: 16 CurrentLimit: 2 MaxLimit: 800 MinLimit: 8
```

说明：在本书中由于这样的分析展示比较多，关键数字我会以粗体进行标注，比如“99%”被标记为粗体显示。

CPU 利用率比较高，dump 文件这一刻 CPU 利用率是 99%，说明很可能在这个 dump 文件中能够找到线索。

w3wp 通过内部多线程方式来同时处理多客户端请求，线程池的数量根据请求数自动分配，一般有几十个线程在工作。随便选择一个线程：

```
0:025> kb
ChildEBP RetAddr    Args to Child
1b24f974 75430816 000003c0 00000000 00000000 ntdll!ZwWaitForSingleObject+0x15
1b24f9e0 76da1194 000003c0 ffffffff 00000000 KERNELBASE!WaitForSingleObjectEx+0x98
1b24f9f8 6c3f1030 000003c0 ffffffff 00000000 kernel32!WaitForSingleObjectExImplementation+0x75
1b24fa2c 6c3f1071 000003c0 ffffffff 00000000 clr!CLREvent::CreateManualEvent+0xf6
1b24fa7c 6c3ed3e8 00000000 75393cd7 00000000 clr!CLREvent::CreateManualEvent+0x137
1b24fabc 6c3ed409 ffffffff 00000000 00000000 clr!CLREvent::WaitEx+0x126
1b24fad0 6c4391dd ffffffff 00000000 00000000 clr!CLREvent::Wait+0x19
1b24faf0 6c43a296 1afa0048 00000002 6c43a370 clr!SVR::t_join::join+0xef
1b24fb10 6c43a08f 00000002 1b24fb30 00000001 clr!SVR::gc_heap::scan_dependent_handles+0x31
1b24fb58 6c439615 00000002 00000000 1afa057c clr!SVR::gc_heap::mark_phase+0x427
1b24fb84 6c439cbb 75393dab 00000004 1afa0048 clr!SVR::gc_heap::gc1+0x63
```

```
1b24fbc0 6c439328 00000000 00000000 1afa0048 clr!SVR::gc_heap::garbage_collect+0x30d
1b24fbe8 6c4998cb ffffffff 7754a11c 7754a0ca clr!SVR::gc_heap::gc_thread_function+0x73
1b24ff00 76da33ca 1afa0048 1b24ff4c 77549ed2 clr!SVR::gc_heap::gc_thread_stub+0x7e
1b24ff0c 77549ed2 1afa0048 6c4152e5 00000000 kernel32!BaseThreadInitThunk+0xe
1b24ff4c 77549ea5 6c499879 1afa0048 ffffffff ntdll!__RtlUserThreadStart+0x70
1b24ff64 00000000 6c499879 1afa0048 00000000 ntdll!_RtlUserThreadStart+0x1b
```

这个线程的调用堆栈如上所示，调用顺序从下往上。可以看到这个线程在等待 GC 操作。又看了其他几个不同的线程，也是如此。

其中有个 35 号线程有点问题，它的堆栈调用如下：

```
0:025> ~35s
eax=00000000 ebx=00000000 ecx=27591b10 edx=1576c8c0 esi=000003f8 edi=00000000
eip=7752f8c1 esp=1c67d518 ebp=1c67d584 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!ZwWaitForSingleObject+0x15:
7752f8c1 83c404          add     esp,4
0:035> !clrstack
OS Thread Id: 0x17e8 (35)
Child SP IP        Call Site
1c67d820 7752f8c1 [HelperMethodFrame: 1c67d820]
1c67d870 6b87781c System.String.Concat(System.String, System.String)
1c67d888 0100cb90 U.King.EE.FF.EEDoc.DocFF.GetAllSubEmployee(System.String)
1c67d8b8 0100ca6d U.King.EE.Service.EEDoc.EEService.GetAllSubEmployee(System.String)
1c67de14 6c3921db [DebuggerU2MCatchHandlerFrame: 1c67de14]
1c67e08c 6b87d37c System.Reflection.RuntimeMethodInfo.Invoke(System.Object, System.Reflection
BindingFlags, System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo, Boolean)
……（省略完整代码）
69c52cdd System.Web.HttpApplication.System.Web.IHttpAsyncHandler.BeginProcessRequest(System.
Web.HttpContext, System.AsyncCallback, System.Object)
1c67f0ac 69c9a8f2 System.Web.HttpRuntime.ProcessRequestInternal(System.Web.HttpWorkerRequest)
1c67f0e0 69c9a63d System.Web.HttpRuntime.ProcessRequestNoDemand(System.Web.HttpWorker
Request)
1c67f0f0 69c99c3d System.Web.Hosting.ISAPIRuntime.ProcessRequest(IntPtr, Int32)
1c67f0f4 6a2b5a7c [InlinedCallFrame: 1c67f0f4]
1c67f168 6a2b5a7c DomainNeutralILStubClass.IL_STUB_COMtoCLR(Int32, Int32, IntPtr)
1c67f2fc 6c3925c1 [GCFrame: 1c67f2fc]
1c67f36c 6c3925c1 [ContextTransitionFrame: 1c67f36c]
1c67f3a0 6c3925c1 [GCFrame: 1c67f3a0]
1c67f4f8 6c3925c1 [ComMethodFrame: 1c67f4f8]
```

一般%Time in GC 消耗时间比较大的原因是，方法长时间执行，并且产生很多对象，导致 GC 线程不断地释放空引用对象。而这里的 GetAllSubEmployee 方法内部存在连接运算 String.Concat（字符“+”），可以推测很可能是 GetAllSubEmployee 方法中有大量循环调用“+”的操作，导致不断地创建对象，不断地被 GC 线程回收，所以 GC 线程忙碌。到目前这只是推测。

进一步看一下 35 号线程中方法的参数值，如下：

```
0:035> !clrstack -a
OS Thread Id: 0x17e8 (35)
Child SP IP        Call Site
1c67d820 7752f8c1 [HelperMethodFrame: 1c67d820]
1c67d870 6b87781c System.String.Concat(System.String,     System.String)
    PARAMETERS:
        str0 (<CLR reg>) = 0x29c20038
        str1 (<CLR reg>) = 0x0f5c59f4
    LOCALS:
        0x1c67d870 = 0x003c2e10
        <no data>

1c67d888 0100cb90 U.King.EE.FF.GLDoc.DocFF.GetAllSubEmployee(System.String)
    PARAMETERS:
        this = <no data>
        strWhere = <no data>
    LOCALS:
        <no data>
        <no data>
        <no data>
        0x1c67d88c = 0x29c20038
        <no data>
        0x1c67d888 = 0x0f20fa84
        <no data>

1c67d8b8 0100ca6d U.King.EE.Service.GLDoc.DocService.GetAllSubEmployee(System.String)
    PARAMETERS:
        this = <no data>
        strWhere = <no data>
……（省略完整代码）
```

可以看到，String.Concat 的两个参数的地址分别为 0x29c20038 和 0x0f5c59f4，它们是 CLR 寄存器中存储数据的内存地址，通过这两个地址我们能够知道存储的是什么数据。

先看一下 0x0f5c59f4 指针中的数据：

```
0:035> !do 0x0f5c59f4
Name:        System.String
MethodTable: 6b8df9ac
EEClass:     6b618bb0
Size:        92(0x5c) bytes
File:        C:\Windows\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\
mscorlib.dll
String:      ,'237fb7ed-99d8-4b8c-8ee5-49d4b848dc3d'
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
6b8e2978  40000ed        4        System.Int32  1 instance       39 m_stringLength
6b8e1dc8  40000ee        8         System.Char  1 instance       2c m_firstChar
6b8df9ac  40000ef        8       System.String  0     shared  static Empty
```

```
>> Domain:Value    01132680:0aaa0260 1afe6ae8:0aaa0260 <<
```

这个地址中存储的数据为粗体标注的 String 节部分，值为一个 guid 类型的数据。

再看一下 0x29c20038 指针存储的数据：

```
0:035> !do 0x29c20038
Name:          System.String
MethodTable: 6b8df9ac
EEClass:       6b618bb0
Size:          7887918(0x785c2e) bytes
File:          C:\Windows\Microsoft.Net\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\
mscorlib.dll
String:        <String is invalid or too large to print>

Fields:
        MT    Field   Offset                 Type VT     Attr    Value Name
6b8e2978  40000ed      4        System.Int32  1 instance  3943952 m_stringLength
6b8e1dc8  40000ee      8         System.Char  1 instance       27 m_firstChar
6b8df9ac  40000ef      8       System.String  0   shared   static Empty
>> Domain:Value    01132680:0aaa0260 1afe6ae8:0aaa0260 <<
```

值为“<String is invalid or too large to print>”，值已经太大了，不能显示，估算了一下字符串长度为近 4MB（长度：3943952）。

还是得看一下这个地址 0x29c20038 的值才行，性能优化分析不能凭猜，根据我的经验一般去猜十有九错，要用数据说话。用另一个内存查看命令显示一下它的值：

```
0:035> du 0x29c20038 0x29c20038+1000
29c20038  "18c35a15-6a48-40fd-b4ad-001"
29c20078  "7ddafa85d','18c35a15-6a48-40fd-b"
29c200b8  "4ad-0017ddafa85d','18c35a15-6a48"
29c200f8  "-40fd-b4ad-0017ddafa85d','18c35a"
29c206b8  "-6a48-40fd-b4ad-0017ddafa85d','1"
29c206f8  "8c35a15-6a48-40fd-b4ad-0017ddafa"
29c20738  "85d','18c35a15-6a48-40fd-b4ad-00"
29c20a38  "99012f2b6','cda9b452-8ec0-416c-a"
29c20a78  "436-00299012f2b6','cda9b452-8ec0"
29c20ab8  "-416c-a436-00299012f2b6','cda9b4"
29c20af8  "52-8ec0-416c-a436-00299012f2b6',"
29c20b38  "'cda9b452-8ec0-416c-a436-0029901"
29c20b78  "2f2b6','cda9b452-8ec0-416c-a436-"
29c20bb8  "00299012f2b6','cda9b452-8ec0-416"
29c20bf8  "c-a436-00299012f2b6','cda9b452-8"
29c20c38  "ec0-416c-a436-00299012f2b6','cda"
29c20c78  "9b452-8ec0-416c-a436-00299012f2b"
29c20cb8  "6','cda9b452-8ec0-416c-a436-0029"
29c20cf8  "9012f2b6','cda9b452-8ec0-416c-a4"
29c20d38  "36-00299012f2b6','cda9b452-8ec0-"
```

```
29c20d78   "416c-a436-00299012f2b6','cda9b45"
29c20db8   "2-8ec0-416c-a436-00299012f2b6','"
29c20df8   "cda9b452-8ec0-416c-a436-00299012"
29c20e38   "f2b6','cda9b452-8ec0-416c-a436-0"
29c20e78   "0299012f2b6','cda9b452-8ec0-416c"
29c20eb8   "-a436-00299012f2b6','cda9b452-8e"
29c20ef8   "c0-416c-a436-00299012f2b6','cda9"
29c20f38   "b452-8ec0-416c-a436-00299012f2b6"
29c20f78   "','cda9b452-8ec0-416c-a436-00299"
29c20fb8   "012f2b6','cda9b452-8ec0-416c-a43"
29c20ff8   "6-00299012f2b6','cda9b452-8ec0-4"
……（省略完整代码）
……（省略完整代码）
```

表格中显示的即为指针 0x29c20038 的值，跟猜测完全一致。这里 4MB 长度的字符串不会有多大问题，问题是这里会产生成千上万个 guid 对象（**3943952/36=产生 109000 多个对象，也相当于这个 for 循环了 10.9 万次以上**），让 w3wp 的 GC 线程一直忙碌，不能处理客户端请求，并且消耗了 99%的 CPU 资源。

到现在问题原因已经非常明确，35 号线程 GetAllSubEmployee 方法中代码的原因导致此问题。接下来的工作就好做了，根据 GetAllSubEmployee 这个线索用 Reflector 反射代码看一下具体是怎么写的：

```
public string GetAllSubEmployee(string strWhere)
{
    string strSQL = this.GetWhereSql(strWhere);
    DataTable table = execute.query(strSQL);
    string strResult = "";
    foreach (DataRow row in table.Rows)
    {
        strResult = strResult + ((strResult != "") ? (",'" + row["ID"].ToString() + "'") : ("'" +
row["ID_EE"].ToString() + "'"));
    }
    if (strResult == "")
    {
        strResult = "'00000000-0000-0000-0000-000000000000'";
    }
    return strResult;
    }
```

上面 foreach 中果真是采用“+”号进行连接的。循环了 10.9 万次以上，并且每次循环还用了多个字符串“+”连接符，而且 dump 文件时它还在循环过程中，可能实际循环次数还要多。仅仅这一句代码就导致 GC 线程占用 90%的 CPU，并导致所有客户端卡死及 CPU 占用率 100%。

另外，根据这个循环次数可以推断，可能是开发人员把数据库表中的所有数据追加起来，仅在客户使用过程中数据量增大后才呈现，该问题非常隐蔽。

还有，用 guid 数据库类型表示这样的业务，就不如用 int 或 string，guid 更浪费空间。

修改方案：采用 StringBuilder 代替 String.Concat。这再简单不过了，只要是开发人员都会知道怎么修改。

小提示：

为什么 StringBuilder 代替 String.Concat 会解决此问题呢？

这与 StringBuilder 的实现原理有关，它的类定义如下（伪代码）：

```
Public class StringBuilder
{
    String strCurrentString;
    StringBuilder orinal;
}
```

当调用 StringBuilder 的 Append 方法时，StringBuilder 对象不会复制新的对象，而是生成新的 StringBuilder 对象，并用里面的 orinal 指向原来的对象。相当于用引用（指针）指向所有零散字符串的地址。简单地说，每次有新对象追加，只是改变一下指针，指向原来的对象地址。

而 String.Concat 则不一样，c=a+b 中的“=”运算符每运算一次总是会生成新的对象；另外“=”运算符也比较耗时，不如 StringBuilder.Append 方法效率高。

上面是对成功解决一个性能问题后的过程回放，事实上在找到问题之前分析的过程就像大海捞针，甚至毫无头绪，因为产生同一种现象的原因太复杂了。

这是一个根据现场痕迹寻找线索，再逆向推断定位，最终找到问题代码的过程，同时也是一个很有趣、很具有挑战性的过程。

1.2 性能优化理论体系

性能优化的理论体系如图 1-10 所示。

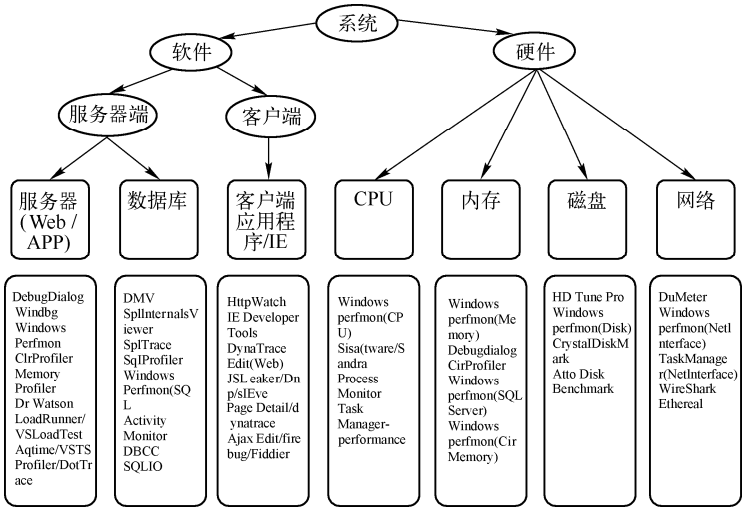

图 1-10　性能优化理论体系

说明：图 1-10 所示的这些工具集的应用场景在第 2 章还会有介绍。

当一个系统出现性能问题，概括地讲，一般主要由两种类型原因导致：系统上运行的软件和硬件。

软件部分又分为：Web 服务器、应用服务器、数据库、客户端应用程序或浏览器等；硬件部分一般出现问题较多的有 CPU、内存、磁盘、网络这 4 个硬件。每一个部件出现的问题也是各种各样的：

- Web 服务器出现的问题比较典型，如：运行中崩溃/异常、内存泄露、CPU High、线程死锁挂起及常用的服务端代码性能问题等。
- 数据库服务器问题也较多，SQL 脚本效率低下及缺少索引都可能会导致 CPU、内存、磁盘、网络出现问题。

不管是 Web 服务器还是数据库服务器，它们都会导致 CPU、内存、磁盘、网络等出现问题，并且出现问题的现象是一样的。可以说性能优化工程师就是一个医生，只是他诊断的不是病人，而是计算机系统；或者说是一个特工，在服务器崩溃现场寻找线索。

不管是软件还是硬件出现问题，作为一个做优化的专业人员，都应当有一些手段和工具去寻找、定位。

事实上在分析过程中，并不是如图 1-10 中标注的每个组件出现问题就用下方对应的手段（工具）去解决，也会遇到以下几种情况：

- 软件和硬件是互相影响的

软件有问题会导致硬件出现问题，硬件有问题会导致软件性能出现问题。比如我们在做产品监控时，监控到内存出现问题后，相应的 CPU、磁盘等也会出现问题，要通过经验正确地找到哪个硬件是主要问题。

不管是软件还是硬件出现问题，都表现在硬件负载异常。比如磁盘出现问题了，有可能是缺少索引导致大量 I/O 所致，有可能是 SQL 效率低下所致，也有可能是数据库目标内存设置得太小导致内存与磁盘之间大量页交换所致……。一般来讲，大部分性能问题要先从软件入手，大部分问题可以通过软件部件优化解决。如果仍无法解决，再通过一些工具检测一下出现负载异常的硬件是否真的有问题，如 CPU 是否运算能力比较差，磁盘是否读写能力差，升级硬件解决即可。

- 杠杆平衡原理（内存⇔CPU 和磁盘）

杠杆平衡原理主要是针对图 1-10 中右边的几个硬件而言。在一个系统中，内存配置越大，则系统对 CPU 和磁盘资源占用就越少；反之，内存配置越小，则系统对 CPU 和磁盘资源占用就越多。以数据库服务器为例，内存配置小了，系统内存不够用，就无法利用缓存，每次取数据都要从磁盘读取，并不断用置换算法把内存空间腾出来，缓存新的数据，就会不断导致大量的页交换，进而导致对 CPU 和磁盘更多的占用。

了解这个原理不管对我们做产品优化还是对客户资金投入都是有参考意义的。内存价格非常便宜，而 CPU 和磁盘都比内存贵得多，所以能够根据此经验给客户提供最佳性价比的服务器配置建议非常重要。

最后再补充一点，性能优化工作不仅仅是解决代码和 SQL 问题，要掌握一整套性能分析方法，包括对各种硬件和系统运行的环境都要熟悉。

例如数据库方面，开发工程师只要了解 SQL 语法基本上就可以开发了，但优化工程师除了要掌握这些，还要了解数据库的各个层面，如：存储引擎、编译优化、大并发下的阻塞/死锁诊断，及 CPU/内存/磁盘负载下的各种诊断方法。

在分析问题时最重要的不是工具应用得多么熟练，而是思考。本章的两个案例分别从不同的角度入手，最终解决问题，分析时不要搞错方向。

性能优化 = 思考 + 经验 + 方法