**TUXONE开发指南**

(针对TUXONE 2.0)

目录

[**TUXONE开发指南** 1](#_Toc289031182)

[第1章、 TUXONE简介 1](#_Toc289031183)

[1.1、 高度分布式处理 1](#_Toc289031184)

[1.2、 超大规模并发性 1](#_Toc289031185)

[1.3、 自动化负载均衡 1](#_Toc289031186)

[1.4、 稳健的故障管理 1](#_Toc289031187)

[1.5、 完善的灾备机制 1](#_Toc289031188)

[1.6、 强大的集群模式 2](#_Toc289031189)

[1.7、 丰富的数据类型 2](#_Toc289031190)

[第2章、 TUXONE系统的安装 2](#_Toc289031191)

[2.1、 系统组成部件 2](#_Toc289031192)

[2.2、 对系统的要求 2](#_Toc289031193)

[2.3、 对IPC的要求 3](#_Toc289031194)

[2.4、 对其他内核参数的要求 3](#_Toc289031195)

[2.5、 安装方法 3](#_Toc289031196)

[第3章、 TUXONE系统的体系结构 4](#_Toc289031197)

[3.1、 整个系统的体系结构 4](#_Toc289031198)

[3.2、 Gate组件的工作原理 5](#_Toc289031199)

[第4章、 TUXONE应用系统的配置 5](#_Toc289031200)

[4.1、 环境变量 6](#_Toc289031201)

[4.2、 GATE配置说明 6](#_Toc289031202)

[4.3、 应用配置说明 7](#_Toc289031203)

[4.4、 灾备与集群方式配置说明 8](#_Toc289031204)

[第5章、 TUXONE的缓冲区 8](#_Toc289031205)

[5.1、 CARRAY缓冲区 9](#_Toc289031206)

[5.2、 STRING缓冲区 10](#_Toc289031207)

[5.3、 VIEW32缓冲区 10](#_Toc289031208)

[5.4、 FML32缓冲区 11](#_Toc289031209)

[5.5、 XML缓冲区 11](#_Toc289031210)

[第6章、 TUXONE的通信方式 11](#_Toc289031211)

[6.1、 同步请求/应答方式 11](#_Toc289031212)

[6.2、 异步请求/应答方式 12](#_Toc289031213)

[6.3、 会话通信方式 12](#_Toc289031214)

[第7章、 TUXONE系统的服务器编程 12](#_Toc289031215)

[7.1、 编译服务器 12](#_Toc289031216)

[第8章、 TUXONE系统的客户机编程 13](#_Toc289031217)

[8.1、 编译客户机 13](#_Toc289031218)

[第9章、 TUXONE系统的线程管理与连接管理 13](#_Toc289031219)

[9.1、 关于线程池 13](#_Toc289031220)

[9.2、 关于网络连接管理 14](#_Toc289031221)

# TUXONE简介

TUXONE是一款分布式交易中间件，同ORACLE TUXEDO API级别兼容。现有的TUXEDO应用可以不需要修改任何代码，只要在TUXONE环境下重新编译，就可以平滑迁移至TUXONE平台。

## 高度分布式处理

TUXONE系统屏蔽了硬件、网络、数据库和操作系统的复杂性，提供了一套简单统一的编程接口，使程序员可以把精力集中在业务逻辑的实现上，而不必再为数据库、操作系统和网络通信协议的复杂性、异构性和可靠性担忧。

## 超大规模并发性

采用特别设计的网络程序架构，可支持超大规模的并发请求。经测试，单节点稳定并发数在10K以上。

## 自动化负载均衡

有别于传统式的集中负载均衡机制，TUXONE采用分布式负载均衡算法，大大缓解中央节点的压力，可在所有可用资源之间动态均衡请求，自动寻找可用服务，隔离故障节点，确保稳定的高吞吐量。

## 稳健的故障管理

为了确保应用的不间断访问，TUXONE连续监控各种组件，以防应用、交易、网络和硬件发生故障。并可以自动重新启动和停止应用服务，消除了单点故障，无论何时何地，只要客户和合作伙伴需要，各种应用总是处于可用状态。

## 完善的灾备机制

TUXONE各节点之间采用实时互备机制，一旦某一节点发生故障，后备节点立刻接替故障节点，保证系统实时在线。

## 强大的集群模式

TUXONE从设计之初就为集群模式做好了准备，可以说TUXONE先天就具备集群方式。TUXONE的核心组件可以部署在多台机器上，用户的SERVER也可以同时部署到不同的机器上，它们互相协作，提高了整个系统的性能，并有效避免了单点故障。

## 丰富的数据类型

数据类型缓冲区是一块格式化的内存区域，它是TUXONE分布式应用程序之间交换数据的基本渠道。TUXONE支持5种数据类型缓冲区，它们是CARRAY、STRING、VIEW32、FML32、XML，不同数据类型缓冲区的存取方式和用途不一样，CARRAY适合于传递图片、声音等二进制数据，STRING适合于传递文本和字符串，VIEW32和FML32适合于传递记录集，XML适合于传递UTF-8的XML文档。

# TUXONE系统的安装

TUXONE的核心部件可以运行在包括LINUX、AIX、HPUX、SOLARIS等类UNIX服务器平台之上，WINDOWS NT平台暂不支持。

## 系统组成部件

TUXONE系统由若干部件组成，如表格 2‑1所示。

|  |  |
| --- | --- |
| 组件 | 描述 |
| ATMI Server | TUXONE ATMI服务端运行环境，编程语言为C/C++ |
| ATMI Client | TUXONE ATMI客户端运行环境，编程语言为C/C++ |
| txgate | TUXONE 核心进程，负责名字服务，负载均衡，集群管理，灾备恢复等核心功能 |
| 命令行工具 | TUXONE 管理控制台，编译VIEW32、FML32工具、以及编译TUXONE服务器与客户端工具等 |

表格 ‑2TUXONE的组件构成

## 对系统的要求

TUXONE安装时，磁盘的使用量一般在15MB左右，不同平台之间可能会有差异，TUXONE占用的这点磁盘空间可以完全忽略不计。

TUXONE运行时对内存的需求也不大，通常在64MB到128MB之间，每个SERVER都是多线程程序，所以SERVER数量不需要启动太多，这样对内存的要求也会比传统的多进程程序少。

TUXONE对CPU的数量和主频没有特别要求。

TUXONE对软件的要求主要就是C/C++编译器，最好使用操作系统提供的编译器，当然也可以使用免费的gcc，还可以在开发机上完成编译，再搬到生产机上来运行，这时生产机上就没有必要安装编译器了。

## 对IPC的要求

TUXONE少量使用了操作系统提供的IPC资源，一般使用默认配置即可。

## 对其他内核参数的要求

TUXONE大量使用了Socket，尤其是多线程服务器，会占用大量的Socket资源，每一个网络连接都会对应一个Socket资源，所以每个进程会要求最多可打开的文件数量，可通过配置NOFILES内核参数来改变。

## 安装方法

首先，通过telnet连接到主机，使用普通账号即可，以REDHAT ENTERPRISE LINUX 5.0为例，要执行的命令如下：

|  |
| --- |
| [jack@haw a]$ gunzip tuxone2.0.0.1188.linux.x86.beta.tar.gz  [jack@haw a]$ tar xvf tuxone2.0.0.1188.linux.x86.beta.tar  [jack@haw a]$ chmod +x install.sh  [jack@haw a]$ ./install.sh  Please input install directory:/home/jack/tuxone2.0  The directory /home/jack/tuxone2.0 does not exist!  Do you want to create it?(Y/N default:Y):y  Begin to install the software...  Congratulations. TUXONE has been successfully installed to:  /home/jack/tuxone2.0 |

TUXONE已经成功安装到了/home/jack/tuxone2.0目录下，接着执行以下命令来配置TUXONE运行环境

|  |
| --- |
| [jack@haw a]$ cd /home/jack/tuxone2.0  [jack@haw tuxone2.0]$ . ./tux.env |

# TUXONE系统的体系结构

## 整个系统的体系结构

在TUXONE系统中，存在三种进程，Gate进程，Client进程，Server进程。其中Client与Server可以是同一个进程。Gate进程是整个TUXONE系统中的关键进程，它负责名字服务，负载均衡，集群管理等关键功**能，整个**系统如图所示

当Server启动后，会将Server自身需要公布的Service信息注册到Gate上，Client调用tpinit连接服务器时，首先连接Gate，从Gate下载相应的服务信息（服务信息包含相应Server的监听地址），然后Client调用tpcall，直接向对应的Server发起调用。



图表 ‑1TUXONE整体架构

## Gate组件的工作原理

Gate在整个系统中处于最关键的地位，它负责名字服务，负载均衡，集群管理，灾备处理等功能。Gate由若干个进程组成，如图所示。

Gate使用公告板来提供命名服务。公告板是一块共享内存区域，它保存着服务进程、服务、运行环境的配置和统计信息。为了便于快速访问，在运行时系统中，它会被复制到TUXONE系统中的每一个成员节点上。



图表 ‑2Gate的组成

Gate真正对外提供服务的是WSH进程池，WSH进程池由多个WSH进程组成，进程数量会随着并发大小自动增减。每个WSH进程是由一个网络线程池来对外提供服务，线程池中线程数量会随着当前WSH进程的负载大小而自动调整。Gate中同时使用了进程迟与线程池技术，大大增强了其对外提供网络并发访问的能力。

Gate监控进程主要负责调整WSH进程池中进程数量，监控Gate同步进程。当有进程意外退出时，Gate监控进程负责重启。

Gate同步进程是用来与其他Gate同步公告板信息的，它可以同时监控多个Gate的公告板信息变化，当有任何Gate公告板发生变化时，Gate同步进程负责更新本地公告板，使整个TUXONE系统中各个节点的公告板信息同步更新。

# TUXONE应用系统的配置

每个TUXONE应用程序都有一个配置文件，它告诉TUXONE系统，应该如何配置和部署服务器进程，应为为服务器进程提供什么样的运行环境。这个配置文件的作用和J2EE规范中的部署描述符是一样的，它告诉TUXONE系统（相当于J2EE规范中的容器），应该如何配置和调度服务进程（相当于J2EE规范中的EJB组件）。

可以通过环境变量、默认配置文件、命令行指定配置文件、命令行配置项来配置TUXONE应用程序，它们的优先级如下：

* + 命令行配置项优先级最高；（例如txgate -D tuxone.gate.listener.addr=0.0.0.0:26000）
  + 命令行指定配置文件次之；（例如 txgate -c gate.config）
  + 默认配置文件最低；（通过TUXCONFIG环境变量指定）

其中环境变量中的配置与其他形式不冲突。

## 环境变量

TUXONE应用程序在建立、启动和运行的过程中，要从环境变量中读取一些配置参数。下面就意表格的形式对这些变量的含义和用法进行说明，如表格所示。

|  |  |
| --- | --- |
| 环境变量 | 含义和用途 |
| TUXDIR | TUXONE系统的安装路径。只要设置了TUXDIR，TUXONE的build系统命令就会自动到$TUXDIR/include和$TUXDIR/lib目录下去找头文件和库文件。用法举例（UNIX）:TUXDIR/=/usr/tuxone2.0;eoport TUXDIR |
| TUXCONFIG | 应用程序配置文件名。 |
| LD\_LIBRARY\_PATH | 这些变量的作用都是一样的，用于指定除了$TUXDIR/lib目录以外的其他公共库路径 |
| SHLIB\_PATH | LD\_LIBRARY\_PATH用于一般UNIX、LINUX系统，SHLIB\_PATH用于HP-UX， |
| LIBPATH | LIBPATH用于AIX. |
| PATH | PATH的作用就不用解释了，它应该包含TUXONE的bin目录。用法（UNIX平台）：PATH=$PATH:$TUXDIR/bin |
| CC、CFLAGS | 仅用于UNIT平台，指定C编译器和编译选项，不是必须的。如果没有使用操作系统默认的C编译器（如使用gcc）就需要指定CC变量：CC=gcc;export CC |
| TUXLOGDIR | 指定TUXONE应用程序运行过程中产生的日志文件目录 |

表格 ‑1TUXONE环境变量

## GATE配置说明

txgate是tuxone系统中的核心进程，负责名字服务，负载均衡等核心功能，其配置参数如表格所示：

|  |  |  |
| --- | --- | --- |
| 配置项 | 默认值 | 含义和用途 |
| tuxone.gate.listener.addr | 0.0.0.0:26000 | Gate 对外公布的监听地址 |
| tuxone.gate.other.addr | NONE | Gate 其他Gate地址 |
| tuxone.gate.sync.interval | 12 | Gate 之间同步公告板间隔 |
| tuxone.gate.concurrent.max | 512 | Gate 支持的最大并发数 |
| tuxone.gate.concurrent.lowWater | 128 | Gate 并发低水位 |
| tuxone.gate.concurrent.highWater | 2048 | Gate 并发高水位 |
| tuxone.gate.housekeeping.interval | 10 | Gate housekeeping工作间隔时间 |
| tuxone.gate.bbl.berthMax | 128 | 公告板支持的最大泊位数（存储Server信息与Service信息） |
| tuxone.gate.wsh.min | 1 | WSH 进程最小数 |
| tuxone.gate.wsh.max | 32 | WSH 进程最大数 |
| tuxone.gate.wsh.accessMax | 512 | 每个WSH进程允许被访问的最大并发数 |
| tuxone.gate.wsh.threadPool.threadMin | 3 | WSH 线程池最小数 |
| tuxone.gate.wsh.threadPool.threadMax | 128 | WSH 线程池最大数 |
| tuxone.gate.wsh.threadPool.threadMaxIdle | 10 | WSH 线程池最大空闲数 |
| tuxone.gate.wsh.serverValidTime | 30 | WSH housekeeping线程工作间隔时间 |
| tuxone.gate.wsh.housekeeping.interval | 10 | WSH housekeeping线程工作间隔时间 |
| tuxone.gate.wsh.app.heart.interval | 12 | WSH 向APP下发的心跳间隔参数 |
| tuxone.gate.log.enable | true | 是否开启日志打印 |
| tuxone.gate.log.level | 4 | 日志打印级别 |
| tuxone.gate.log.printFile | true | 是否打印至文件 |
| tuxone.gate.log.printConsole | false | 是否打印至标准输出 |

表格 ‑2Gate配置项说明

## 应用配置说明

|  |  |  |
| --- | --- | --- |
| 配置项 | 默认值 | 含义和用途 |
| tuxone.app.gate.addr | 127.0.0.1:26000 | Gate 对外公布的监听地址 |
| tuxone.app.server.listener.addr | 0.0.0.0:0 | SERVER 对外公布的监听地址 |
| tuxone.app.server.concurrentMax | 512 | SERVER 进程的最大并发数 |
| tuxone.app.server.maxLongConnections | 256 | SERVER 最大长连接数 |
| tuxone.app.server.threadMode | MT | SERVER 网络线程模型 |
| tuxone.app.server.threadPool.threadMin | 3 | SERVER 线程池最小数 |
| tuxone.app.server.threadPool.threadMax | 128 | SERVER 线程池最大数 |
| tuxone.app.server.threadPool.threadMaxIdle | 10 | SERVER 线程池最大空闲数 |
| tuxone.app.client.cachedConnections | 128 | CLIENT 缓存的网络连接 |
| tuxone.app.heart.interval | 3 | APP 心跳间隔（如果连接上Gate，则从Gate上获取此参数） |
| tuxone.app.buffer.cachedSize | 1024000 | APP Buffer cached size（=0表示无限制） |
| tuxone.app.socket.rwtimeout | 90 | app socket读写超时时间 |
| tuxone.app.log.enable | true | 是否开启日志打印 |
| tuxone.app.log.level | 4 | 日志打印级别 |
| tuxone.app.log.printFile | true | 是否打印至文件 |
| tuxone.app.log.printConsole | false | 是否打印至标准输出 |

表格 ‑3应用配置项说明

## 灾备与集群方式配置说明

如何配置两个Gate为互备模式，假设Gate1监听的地址为192.168.0.100:26000。Gate2监听的地址为192.168.0.100:27000，则通过以下方式启动Gate，即可成为互备方式。

|  |
| --- |
| txgate -D tuxone.gate.listener.addr=192.168.0.100:26000 \  -D tuxone.gate.other.addr=192.168.0.101:27000  txgate -D tuxone.gate.listener.addr=192.168.0.101:27000 \  -D tuxone.gate.other.addr=192.168.0.100:26000 |

另外，Client与Server的启动也需要配置相应的Gate地址，才能起到灾备作用

|  |
| --- |
| client -D tuxone.app.gate.addr=192.168.0.100:26000;192.168.0.101:27000  server -D tuxone.app.gate.addr=192.168.0.100:26000;192.168.0.101:27000 |

集群方式部署同灾备类似，假设有三台机器需要以集群方式部署TUXONE系统，则配置方式如下

|  |
| --- |
| txgate -D tuxone.gate.listener.addr=192.168.0.100:26000 \  -D tuxone.gate.other.addr=192.168.0.101:27000;192.168.0.102:28000  txgate -D tuxone.gate.listener.addr=192.168.0.101:27000 \  -D tuxone.gate.other.addr=192.168.0.100:26000;192.168.0.102:28000  txgate -D tuxone.gate.listener.addr=192.168.0.102:28000 \  -D tuxone.gate.other.addr=192.168.0.100:26000;192.168.0.101:27000 |

|  |
| --- |
| client -D tuxone.app.gate.addr=192.168.0.100:26000;192.168.0.101:27000;192.168.0.102:28000  server -D tuxone.app.gate.addr=192.168.0.100:26000;192.168.0.101:27000;192.168.0.102:28000 |

# TUXONE的缓冲区

跨平台通信应用程序一般使用缓冲区来在两个通信实体之间交换数据。以UNIX下SOCKET编程为例，客户进程通常使用char sendBuf[1024]来分配一个大小为1KB的缓冲区，然后将要传递的数据复制到sendBuf首地址指向的缓冲区中，最后调用write()函数把缓冲区内容写到发送窗口。运行在服务器上的守护进程检查到socket上有数据读取时，就调用read()函数从接收窗口读取数据。在数据发送和接收的过程中，程序员需要考虑并解决下面一些问题。

* 缓冲区大小。由于在设计时无法预知运行时可能产生的消息的大小，通常定义一个MAX\_BUFSIZE常量来指定最大缓冲区长度，程序中使用char sendBuf[MAX\_BUFSIZE]来静态分配发送和接收缓冲区。这印出来一个问题就是，MAX\_BUFSIZE定义多大比较合适。MAX\_BUFSIZE定义得太小会影响性能，定义得太大可能会造成内存浪费。
* 字节序问题。大多数数据通信发生在异构系统之间，这时就不得不考虑字节序的问题，否则通过网络传输整数将会产生混乱。有的操作系统使用小端字节序（LITTLE\_ENDIAN，低序字节存储在起始地址），如AIX，WINDOWS等；有的操作系统使用大端字节序（BIG\_ENDIAN，高序字节存储在起始地址），如SUN SOLARIS。通常的做法是在发送端将主机字节序转化成网络字节序，在接收端再把网络字节序转化成主机字节序。
* 字符集和编码问题。有的系统只支持单字节字符集（SBCS），它的所有字符都只有一个字节的长度，如ASCII。有的系统支持多字节字符集（MBCS），它包含的字符中有单字节长的字符，也有多字节长的字符。有的系统支持统一字符集（UNICODE），在UNICODE编码标准中的所有字符都是双字节。
* 数据压缩问题。大多数通信系统中交换的数据都不大，因此都不涉及到数据压缩问题。而在有的系统中，通信实体之间常常需要交换大量的数据流，这时使用压缩算法将大大提供性能。
* 加密/解密问题。如果安全对应用来说十分重要，这时就需要考虑采用某种算法来进行数据加密和解密。

对于一个普通的程序员来说，要完全考虑以上这些因素几乎是不可能的。然而在TUXONE系统中，这些原本应该由程序员考虑的因素都成为了“平台特性”，这就使程序员可从底层通信细节实现中解放出来，把精力集中在应用逻辑上。TUXONE就是这样，它以自身实现的复杂性换来了应用实现的简单。

TUXONE的客户端与服务端之间的数据传送是通过数据缓冲区来进行的。TUXONE数据缓冲区主要包括CARRAY、STRING、VIEW32、FML32，XML数据缓冲区。

在TUXONE中客户端与服务端之间进行数据交换的缓冲区（如：tpcall()中的输入，输出缓冲区等），都要用TUXONE自己提供的API进行操作。不能采用C语言的函数（如：mallloc()，free等分配）分配释放这些缓冲区。同时在程序中要自己管理这些缓冲区，像C语言中的缓冲区一样，在用tpalloc()分配一块缓冲区之后，在不在需要该缓冲区时，用tpfree释放掉。

## CARRAY缓冲区

CARRAY是定长的字符串缓冲区，它与C语言中的字符缓冲区是等价的。NULL字符一般被认为是字符串的结束符，但在CARRAY缓冲区中，NULL是有意义的，这就是为什么使用CARRAY缓冲区时，需要指定长度的原因。TUXONE原封不动地将CARRAY缓冲区传递给目的主机，而不对其中的数据进行任何解释。也不进行编码/解码处理，所以它比较适合于传输二进制数据，如位图、视频剪辑等。由此可见，CARRAY缓冲区不支持DDR是理所当然的，CARRAY最大的特点就是速度快，效率高。

CARRAY缓冲区使用方式，如代码清单所示：

|  |
| --- |
| char \*sendbuf = NULL;  sendbuf = (char \*) tpalloc("CARRAY", NULL, 1024);  if(sendbuf == NULL) {  (void) fprintf(stderr, "Error allocating send buffer\n");  return -1;  }  tpfree(sendbuf); |

## STRING缓冲区

在STRING缓冲区中，NULL字符被认为是字符串的结尾，这个特征是STRING与CARRAY的主要区别之一，这也是为什么STRING缓冲区不必指定长度的原因，因为从缓冲区的首地址开始，到第一个NULL字符之间的数据被认为是缓冲区的内容。STRING缓冲区是自描述的，因此在字符集不同的主机之间传输时，TUXONE系统会自动进行数据转换。STRING缓冲区不支持编码/解码特性，也不支持DDR。STRING缓冲区具有使用简单，效率高的特点，适合于跨平台传输大量的文本数据，也是实际编程中用得最多的缓冲区类型。

STRING缓冲区使用方式，如代码清单所示：

|  |
| --- |
| char \*sendbuf = NULL;  sendbuf = (char \*) tpalloc("STRING", NULL, 1024);  if(sendbuf == NULL) {  (void) fprintf(stderr, "Error allocating send buffer\n");  return -1;  }  tpfree(sendbuf); |

## VIEW32缓冲区

VIEW32缓冲区允许在异构平台之间传输C语言的结构（structures）。使用tpalloc来分配VIEW32缓冲区时，需要使用subtypes参数来指定和它关联的C语言结构名。VIEW32支持DDR，在不同的主机之间传递时，还支持自动编码/解码特性。

使用VIEW32缓冲区时，需要先定义一个描述文件，再通过TUXONE系统提供的VIEW32编译器viewc32生成一个.h文件和一个扩展名为.V的二进制文件，其中.h文件定义了VIEW32对应的C语言结构，.V文件定义了只有TUXONE系统才能识别的运行时描述文件，需要加到客户机和服务器的环境变量VIEWFILES32中，另外还需要设置VIEWDIR32环境变量来指定.V文件所在目录。

## FML32缓冲区

FML（Field Manipulation Language）是一组用于定义和管理字段缓冲区（Fielded Buffers）的C语言函数，它提供了一种更为高层的内存访问方法。字段缓冲区是一块格式化的内存区域。

FML32字段支持的数据类型有：short、long、float、double、char、STRING、CARRAY。对于需要指定长度的字段，FLDID32后面会自动添加一个FLDLEN域来保存字段的长度。

FML32字段支持DDR，在异构平台之间传输时支持编码和解码特性。FML32缓冲区是可变长的，当使用tpalloc()分配时，如果不指定长度，则默认分配的大小是1024KB。

另外，程序员只能通过FML32函数来存取FML缓冲区。分配缓冲区代码如下所示

|  |
| --- |
| char \*sendbuf = NULL;  sendbuf = (char \*) tpalloc("FML32", NULL, 1024);  if(sendbuf == NULL) {  (void) fprintf(stderr, "Error allocating send buffer\n");  return -1;  }  tpfree(sendbuf); |

## XML缓冲区

TUXONE内部集成了 Apache Xerces C++ Parser 2.8，这样TUXONE服务器和客户端就可以直接分析XML文档了，分配方式同CARRAY类似。

# TUXONE的通信方式

## 同步请求/应答方式

在同步请求/应答方式中，客户端使用tpcall向远程服务器（由TUXONE系统根据公告板信息确定）发送服务请求，此时客户端将传送请求服务的名字、用于请求服务的输入参数和输出参数。tpcall发出后，客户的数据被传送至服务器，得到相应的服务处理。在此方式下，服务器处理请求时，客户端将等待，不继续运行，直到服务器返回相应结果。

图表 ‑1同步通信



## 异步请求/应答方式

注：此功能将在下一版本增加。

## 会话通信方式

注：此功能将在下一版本增加。

# TUXONE系统的服务器编程

## 编译服务器

TUXONE提供了buildserver命令来编译服务器进程。事实上buildserver只完成预编译，它会调用当前操作系统中已经安装的默认C编译器来完成进一步的编译和连接，最终生成可执行代码，这就是为什么在安装TUXONE时，要确保已经安装了C编译器的原因。

建议使用与操作系统一起发布的C编译器。由于在大多数UNIX平台（比如AIX、SOLARIS、HP-UX等）上C编译器不是免费的，所以在没有配套编译器的情况下，也可以使用免费的gcc。

buildserver命令的使用方法如下：

|  |
| --- |
| buildserver [-s services] [-v] [-o outfile] [-f firstfiles] [-l lastfiles] |

* -v 表示打开VERBOSE模式，即编译过程中打印更多的信息。
* -s services定义服务与函数的映射关系。通常情况下，函数名与服务名一致。
* -f firstfiles指定要优先于TUXONE系统库之前连接的文件名。则文件名之间应以空格分割，并用引号把整个串引起来（如-f “f1.c f2.c f3.c”），也可以对每个文件使用一个-f选项（如-f f1.c –f f2.c –f f3.c）
* -l lastfiles 指定要在TUXONE系统库之后连接的文件名，用法同-f。
* -o outfile 指定要生成的可执行文件名。

# TUXONE系统的客户机编程

## 编译客户机

TUXONE系统提供了buildclient命令来编译C语言客户机程序。与buildserver一样，buildclient也只完成预编译，它会调用当前操作系统中已经安装的默认C编译器来完成进一步编译和连接，最终生成可执行代码。如果TUXONE客户端不是用C语言编写的，而是使用Visual Basic这样的快速开发工具来编写的，那么编译工作就由IDE来完成，与buildclient没有什么关系。

buildclient命令的使用方法如下：

|  |
| --- |
| buildclient [-v] [-o outfile] [-f firstfiles] [-l lastfiles] |

buildclient中选项的含义与buildserver完全一致。

# TUXONE系统的线程管理与连接管理

线程（或一个进程中一小段连续控制流）也称为轻量级进程，它通过与其它线程共享基础部件减少资源开销。线程是轻量级的，因此一个进程中可以包含许多线程。利用多线程可在应用程序中提供并发能力并改善性能。利用多个线程同时执行多个独立的计算，可以使应用程序结构更有效率。例如，数据库系统可以在支持多个用户互动操作的同时执行几个文件和网络操作。虽然可以将软件编写成从一个请求到另一个请求异步移动的一个控制线程，但是，如果将每个请求编写为一个单独序列，而由基本系统处理不同操作的同步交错，就可以简化代码。处理不同操作的同步交错，就可以简化代码。

## 关于线程池

网络服务器的线程池模型会定义可分配用于处理客户端请求的最大线程数。每个客户端请求会得到一个工作线程，但是仅用于该特定请求的时段。请求完成后，分配给该请求的工作线程就会放回可用线程池内，准备重新分配，处理以后来自任何客户端的请求。

线程池模型中，线程是根据向服务器发出的请求的通信量分配的。这意味着同时向服务器发出很多请求的非常活跃的客户端会分配到多个线程（保证请求得到迅速执行），而不活跃的多个客户端可能共享一个线程，但是其请求仍能够立即得到处理。另外，这样可以减少与创建和撤消工作线程相关的资源开销，因为线程得到重新利用而不是撤消了，而且线程可以分配给多个连接。

TUXONE根据并发客户端请求的数量动态分配线程池中的线程数，从而节约系统资源。如果客户端很活跃，可以分配线程满足其需要。如果线程使用效率不高，TUXONE会释放它们，只保留足够的线程，满足客户端的当前需要。这样使服务器中的线程数始终保持在最佳状态。线程池的大小根据服务器活动情况变化，而且可以按照特定分布式系统的需要自由配置（无论是在执行前，还是在执行后）。

TUXONE中的用户应用SERVER端使用了线程池模型，用户可以通过以下参数来配置线程池：

* tuxone.app.server.threadPool.threadMin  
  线程池中最小线程数
* tuxone.app.server.threadPool.threadMax  
  线程池中最大线程数
* tuxone.app.server.threadPool.threadMaxIdle  
  线程池中最大空闲线程数

每次收到客户端请求时，都会尝试从线程池分配线程来处理请求。如果这是第一个客户端请求且池为空，则创建一个线程。同样，如果所有线程均忙，就会创建新线程来处理请求。服务器可以定义可分配用于处理客户端请求的最大线程数。如果池中没有线程可用且已经创建最大线程数，请求就会受到阻挡，直到有当前正在使用的线程被释放回池。

## 关于网络连接管理

Client与Server之间的网络连接通常采用长连接方式，当并发量超过Server配置参数tuxone.app.server.maxLongConnections的值后，后续的网络连接请求将自动变为短连接，也就是说Client与Server之间采用长短结合的方式，并发量小的情况下采用短连接，并发量大的情况下采用长连接。

Client本地会有一个缓存的网络连接池，连接的最大数量由tuxone.app.client.cachedConnections决定。超过配置数量后，新建立的连接自动变为短连接，即调用完成后，自动断开连接。