ISE 使用手册

郝新庚 (haoxingeng@gmail.com)

最后更新：2013-05-30

一. ISE简介

1.1 ISE概述

ISE（Iris Server Engine）是一个基于现代 C++的跨平台（Linux 和 Windows）的高性能多线程并发网络服务器程序框架。它封装了琐碎的 socket 以及各种操作系统 APIs，以面向对象方式向开发者提供稳定、高效、易扩展、易配置、易维护的程序框架。ISE 的用户只需遵循接口的约定，挂接自己的业务程序，即可轻松开发出稳定、高效的网络服务器程序。

1.2 ISE的主要特点

- 跨平台。目前支持 Linux 和 Windows。
- 基于多线程并发，而非多进程。
- 支持 TCP 与 UDP 服务端程序开发。
- 基于 Proactor 模式的 TCP 服务接口。
- 基于请求分组与负载自适应的高并发 UDP 服务模型。
- 不支持 IPv6，只支持 IPv4。
- 提供通用数据库接口（DBI），并预置 MySQL 扩展。
- ISE 是一个程序框架（Framework），而非一个程序库（Library）。

1.3 为什么要跨平台

ISE 克服了许多平台差异带来的困难，在不牺牲性能的前提下，为开发者提供了一致的接口。ISE 跨平台的目标是 Linux 和 Windows，短期内不考虑其它平台。在这两种平台下，多线程网络服务程序需要面对的问题有：

- 多线程环境编程基础设施。比如线程实现方式、互斥锁、条件变量、信号量、原子整数等。两种平台对这些基础设施的实现差异甚大，比如 Windows 平台下，在 Vista 之前的系统并不提供对条件变量的支持，需要开发者通过多种同步机制自行模拟；再比如 Windows 中的事件对象（Event）在 Linux 中并无对应的机制。
- IO 多路复用机制（IO multiplexing）。Linux::EPoll 和 Windows::IOCP 分别代表了两种平台下最高效的 IO 多路复用机制，但是两者的设计理念完全不同，EPoll 以 Reactor 模式为基础，而 IOCP 则以 Proactor 模式为基础。在不同的平台下，要求开发者以不同的思维模式来设计自己的程序。ISE 解决了这个问题。
- 数据库访问机制。虽然不是全部，但有相当一部分服务端程序存在访问数据库的需求。在不同的平台下，访问数据库的途径五花八门，而如果要实现跨平台，则有必要实现一套统一的机制。
- 其它常用操作。比如文件读写、日期时间、内存管理、日志输出等等。平台的差异性将导致这些

最常用操作的实现方式大相径庭。ISE 封装了这些差异，为开发者提供了一致的接口。

从这些差异来看，ISE 为跨平台做了不少工作。但好在付出的这些代价只局限在 ISE 本身的开发，对于 ISE 的用户来说，跨平台并没有带来坏处。ISE 跨平台的前提是：不牺牲性能。

那么 ISE 为什么要跨平台？因为：

- 这是 ISE 的选择；
- 如果你正在做一个基于 Linux 的网络服务程序，而你的团队中有人并不擅长 Linux 编程，那么他可以拿起他最擅长的 Visual Studio 工具，写出在两种平台下行为一致的程序（ISE 保证了这一点）。
- 你的产品不只面向一个客户，有些客户要求基于 Linux 系统，而有些要求 Windows 系统。没关系，基于 ISE 的项目可以做到“一份代码，两处编译”。

1.4 编译与安装

ISE 是一个开源项目，它在 GitHub 中的地址是：

http://github.com/haoxingeng/ise

ISE 的主要开发环境：

- Linux 平台：
  Debian 6.0 Squeeze（内核版本：2.6.32）
  g++ 4.4
  32 位和 64 位环境。
- Windows 平台：
  Visual C++ 2005 SP1

ISE 依赖 Boost，但只用到了 boost::function/bind、shared_ptr、scoped_ptr、any 等这些无需编译的部分。安装 Boost 很简单：

Windows 平台下，可先下载 boost 压缩包后解压到某个目录下（比如 c:\boost），再在 Visual C++ 中“选项 -> 项目和解决方案 -> VC++目录 -> 包含文件”中添加 boost 目录（比如 c:\boost）即可。

Linux 平台下，可直接 apt-get 安装：

```
# sudo apt-get install libboost1.40-dev
```

在 Windows 平台下编译 ISE，请先安装 Visual C++ 2005 SP1，再打开 ise\build\windows\VC8\ 目录下的解决方案（*.sln）文件即可。

libise.sln - ISE 主程序。
libise_dbi_mysql.sln - ISE 数据库接口（DBI）的 MySQL 扩展。

libise_utils.sln - ISE 的 utils 扩展。
examples.sln - 各种示例程序。

在 Linux 平台下，ISE 采用 CMake 进行编译。如果系统未安装 CMake，可先安装它：

```
# sudo apt-get install cmake
```

编译 ISE 的方法很简单：

```
# cd ise/build/linux
# ./build.sh
# ./build.sh install
```

以上命令将 ISE 的头文件和生成的静态库文件安装到 ise/build/linux/debug-install/ 目录下。

如果要编译为 release 版，可执行：

```
# BUILD_TYPE=release ./build.sh
# BUILD_TYPE=release ./build.sh install
```

以上命令将 ISE 的头文件和生成的静态库文件安装到 ise/build/linux/release-install/ 目录下。

编译完成之后，可运行 ise/build/linux/debug/bin 目录下的示例程序，比如 server_inspector。然后在浏览器中输入 http://192.168.0.100:8080，其中 192.168.0.100 请替换成您的 Linux 机器的实际 IP 地址。

1.5 目录结构

ISE 的目录结构如下：

```
ise                             - ISE 根目录
├── build                       - 项目编译根目录
│    ├── linux                  - Linux 平台的编译目录
│    └── windows                - Windows 平台的编译目录
├── doc                         - 文档
├── examples                    - 示例程序
└── ise                         - ISE 源码根目录
     ├── ext                    - ISE 扩展
     │    ├── dbi               - 数据库扩展（DBI）
     │    │    └── mysql        - 预置的 MySQL 扩展
     │    └── utils             - 实用程序（utils）扩展
     │         ├── cipher       - 丰富的加解密、散列算法库
     │         └── xml          - 跨平台的 XML 操作封装库
     └── main                   - ISE 主程序
```

编译目录说明：

```
build                                   - 项目编译根目录
├── linux                               - Linux 平台的编译目录
│   ├── debug
│   │   └── bin                         - 存放 debug 模式下示例程序编译结果
│   ├── debug-install
│   │   ├── include                     - 存放 debug 模式下编译安装后的头文件
│   │   └── lib                         - 存放 debug 模式下编译安装后的静态库文件
│   ├── release
│   │   └── bin                         - 存放 release 模式下示例程序编译结果
│   └── release-install
│       ├── include                     - 存放 release 模式下编译安装后的头文件
│       └── lib                         - 存放 release 模式下编译安装后的静态库文件
│
└── windows                             - Windows 平台的编译目录
    └── VC8
        ├── bin                         - 存放示例程序编译结果
        └── lib                         - 存放编译生成的静态库文件
```

ISE 主程序说明：

```
ise
└── main                                - ISE 主程序根目录
    ├── ise.h                           - ISE 的总头文件
    ├── ise_application.*               - 程序框架流程控制
    ├── ise_assert.*                    - 断言支持
    ├── ise_classes.*                   - 基础类库
    ├── ise_database.*                  - 数据库接口定义（DBI）
    ├── ise_err_msgs.*                  - 错误信息定义
    ├── ise_exceptions.*                - 各种异常定义
    ├── ise_global_defs.*               - 全局常量、类型定义
    ├── ise_http.*                      - HTTP 协议的实现
    ├── ise_inspector.*                 - 服务监视
    ├── ise_options.*                   - 编译条件定义
    ├── ise_scheduler.*                 - 简单的定时任务
    ├── ise_server_assistor.*           - 辅助线程管理
    ├── ise_server_tcp.*                - TCP 服务的实现
    ├── ise_server_udp.*                - UDP 服务的实现
    ├── ise_socket.*                    - 基础 socket 封装
    ├── ise_stldefs.*                   - STL 相关的头文件
    ├── ise_svr_mod.*                   - 服务模块支持
    ├── ise_svr_mod_msgs.*              - 服务模块消息支持
    ├── ise_sys_threads.*               - ISE 框架的后台线程
    ├── ise_sys_utils.*                 - 各种跨平台的实用函数
    └── ise_thread.*                    - 线程封装
```

1.6 Hello World

为了快速了解 ISE 的基本使用方法，现在让我们以最快的速度写一个基于 ISE 的“Hello World”程序。这个程序是一个 TCP 服务器，把它运行起来之后，用 telnet 作为客户端去连接它，会得到一行输出“Hello World!”然后连接断开，就这么简单。在 ise/examples/hello_world 目录下可以查看这个示例程序的源码。

hello_world.h 的源码如下：

hello_world.h

```
01 #include "ise/main/ise.h"
02
03 using namespace ise;
04
05 class AppBusiness : public IseBusiness
06 {
07 public:
08     virtual void initIseOptions(IseOptions& options);
09     virtual void onTcpConnected(const TcpConnectionPtr& connection);
10     virtual void onTcpSendComplete(const TcpConnectionPtr& connection,
11         const Context& context);
12 };
```

hello_world.cpp 的源码如下：

hello_world.cpp

```
01 #include "hello_world.h"
02
03 IseBusiness* createIseBusinessObject()
04 {
05     return new AppBusiness();
06 }
07
08 // class AppBusiness
09
10 void AppBusiness::initIseOptions(IseOptions& options)
11 {
12     options.setServerType(ST_TCP);
13     options.setTcpServerPort(12345);
14 }
15
16 void AppBusiness::onTcpConnected(const TcpConnectionPtr& connection)
17 {
18     string msg = "Hello World!\r\n";
19     connection->send(msg.c_str(), msg.length());
20 }
21
22 void AppBusiness::onTcpSendComplete(const TcpConnectionPtr& connection,
23     const Context& context)
24 {
25     connection->disconnect();
26 }
```

从示例程序可以看出，基于 ISE 的程序的开发步骤可简单归结为：

1） 继承 IseBusiness 类，写一个新的业务类 AppBusiness。

2） 写一个函数 createIseBusinessObject()，该函数创建了一个 AppBusiness 对象并返回。

3） 在 AppBusiness 类中根据需要覆写一些虚函数，实现自己的“业务逻辑”。

下面详细解说 hello_world 程序。

在ISE中，IseBusiness类是一条与“业务逻辑”联接的通道，任何程序都要首先继承它。IseBusiness提供了诸多虚函数 [1]提供给用户覆写（override），比如对TCP而言，有四个虚函数值得关注：onTcpConnected, onTcpDisconnected, onTcpRecvComplete, onTcpSendComplete。当相应的事件发生时，ISE会调用这些函数。对hello_world程序而言，它需要关注两个事件：

a. 客户端连接建立时，即 onTcpConnected;（此事件发生时须发送字符串。）

b. 字符串发送完毕时，即 onTcpSendComplete。（此事件发生时须断开连接。）

所以，hello_world.h 的第 *09*、*10* 行覆写了这两个虚函数。

hello_world.cpp 的第 *19* 行调用 connection->send() 发送了字符串“Hello World!”。当发送完毕时，第 *25* 行调用 connection->disconnect() 断开了连接。

如前文所述，hello_world 这个程序是一个 TCP 服务端，并且打算绑定本机端口 12345。这些信息需要“告诉”ISE，途径是覆写 initIseOptions() 这个虚函数。

在 hello_world.cpp 的第 *12*、*13* 行，调用 options.setServerType(ST_TCP) 设定本程序是一个 TCP 服务端，调用 options.setTcpServerPort(12345) 指定绑定的端口号。

程序编译：

在 Linux 下，执行 build.sh 后，可在 ise/build/linux/debug/bin 中找到可执行文件 hello_world。

在 Windows 下，可在 VC++ 2005 中打开 ise/build/windows/VC8/examples.sln，编译完毕后，可在 ise/build/windows/VC8/bin/debug 目录中找到可执行文件 hello_world.exe。

1.7 IseBusiness接口

ISE 提供了一个重要的接口类：IseBusiness。这个类是 ISE 与“业务逻辑”相联的通道。正如从 hello_world 示例程序中看到的，基于 ISE 开发任务程序，第一件事情便是继承 IseBusiness 类。开发者可以认为 IseBusiness 是 ISE 的回调集合，ISE 通过 IseBusiness 将各种事件通知应用程序。

下面是 IseBusiness 的声明：

[1] “面向对象”的怀疑论者可能不太赞成这种“继承+虚函数”的做法。本人不反对“基于对象”，但更喜欢适度继承带来的层次美感。“方法论”的优劣见仁见智。

```
01 class IseBusiness :
02     boost::noncopyable,
03     public UdpCallbacks,
04     public TcpCallbacks
05 {
06 public:
07     IseBusiness() {}
08     virtual ~IseBusiness() {}
09 
10     // 初始化 (失败则抛出异常)
11     virtual void initialize() {}
12     // 结束化 (无论初始化是否有异常, 结束时都会执行)
13     virtual void finalize() {}
14 
15     // 解释命令行参数, 参数不正确则返回 false
16     virtual bool parseArguments(int argc, char *argv[]) { return true; }
17     // 返回程序的当前版本号
18     virtual string getAppVersion() { return "0.0.0.0"; }
19     // 返回程序的帮助信息
20     virtual string getAppHelp() { return ""; }
21 
22     // 初始化 ISE 配置信息
23     virtual void initIseOptions(IseOptions& options) {}
24 
25     // 初始化之前
26     virtual void beforeInit() {}
27     // 初始化成功之后
28     virtual void afterInit() {}
29     // 初始化失败 (不应抛出异常)
30     virtual void onInitFailed(Exception& e) {}
31 
32 public:  /* interface UdpCallbacks */
33     // UDP 数据包分类
34     virtual void classifyUdpPacket(void *packetBuffer, int packetSize, int& groupIndex)
35         { groupIndex = 0; }
36     // 收到了 UDP 数据包
37     virtual void onRecvedUdpPacket(UdpWorkerThread& workerThread, int groupIndex,
38         UdpPacket& packet) {}
39 
40 public:  /* interface TcpCallbacks */
41     // 接受了一个新的 TCP 连接
42     virtual void onTcpConnected(const TcpConnectionPtr& connection) {}
43     // 断开了一个 TCP 连接
44     virtual void onTcpDisconnected(const TcpConnectionPtr& connection) {}
45     // TCP 连接上的一个接收任务已完成
46     virtual void onTcpRecvComplete(const TcpConnectionPtr& connection, void *packetBuffer,
47         int packetSize, const Context& context) {}
48     // TCP 连接上的一个发送任务已完成
49     virtual void onTcpSendComplete(const TcpConnectionPtr& connection,
50         const Context& context) {}
51 
52 public:
53     // 辅助服务线程执行(assistorIndex: 0-based)
54     virtual void assistorThreadExecute(AssistorThread& assistorThread,
55         int assistorIndex) {}
```

```
56      // 系统守护线程执行 (secondCount: 0-based)
57      virtual void daemonThreadExecute(Thread& thread, int secondCount) {}
58  };
```

上的 IseBusiness 类的声明中，每个接口的注释阐述了它的含义。IseBusiness 由以下几大部分组成：

- 程序初始化和结束化。当程序启动时，ISE 会调用 initialize()；程序退出时，会调用 finalize()。应用程序可以将与业务逻辑相关的初始化和结束化在这两个函数中实现。

```
// 初始化 (失败则抛出异常)
virtual void initialize() {}
// 结束化 (无论初始化是否有异常，结束时都会执行)
virtual void finalize() {}
```

- 命令行参数解析、程序版本、命令行帮助信息。

```
// 解释命令行参数，参数不正确则返回 false
virtual bool parseArguments(int argc, char *argv[]) { return true; }
// 返回程序的当前版本号
virtual string getAppVersion() { return "0.0.0.0"; }
// 返回程序的帮助信息
virtual string getAppHelp() { return ""; }
```

- ISE 配置接口。应用程序通过这个接口对 ISE 的所有参数进行配置。通过参数配置，可以设置服务器类型（TCP/UDP）、指定 TCP 监听端口号、选择 TCP 并发模型、进行 UDP 性能调优等等。

```
// 初始化 ISE 配置信息
virtual void initIseOptions(IseOptions& options) {}
```

- 程序初始化过程。应用程序一般会在这些回调中输出一些信息。

```
// 初始化之前
virtual void beforeInit() {}
// 初始化成功之后
virtual void afterInit() {}
// 初始化失败 (不应抛出异常)
virtual void onInitFailed(Exception& e) {}
```

- UDP 回调。classifyUdpPacket() 用于对收到的 UDP 包进行分组；每收到一个 UDP 包时，都会调用 onRecvedUdpPacket()。详见「高并发 UDP 服务程序设计」。

```
// UDP 数据包分类
virtual void classifyUdpPacket(void *packetBuffer, int packetSize, int& groupIndex)
    { groupIndex = 0; }
// 收到了 UDP 数据包
virtual void onRecvedUdpPacket(UdpWorkerThread& workerThread, int groupIndex,
    UdpPacket& packet) {}
```

- TCP 回调。详见「基于 ISE 开发 TCP 服务端程序」。

```
// 接受了一个新的 TCP 连接
virtual void onTcpConnected(const TcpConnectionPtr& connection) {}
// 断开了一个 TCP 连接
virtual void onTcpDisconnected(const TcpConnectionPtr& connection) {}
// TCP 连接上的一个接收任务已完成
virtual void onTcpRecvComplete(const TcpConnectionPtr& connection,
    void *packetBuffer, int packetSize, const Context& context) {}
```

```
// TCP 连接上的一个发送任务已完成
virtual void onTcpSendComplete(const TcpConnectionPtr& connection,
    const Context& context) {}
```

- 辅助线程和系统守护线程回调。详见「辅助线程（Assistor Threads）」。

```
// 辅助服务线程执行(assistorIndex: 0-based)
virtual void assistorThreadExecute(AssistorThread& assistorThread, int assistorIndex) {}
// 系统守护线程执行 (secondCount: 0-based)
virtual void daemonThreadExecute(Thread& thread, int secondCount) {}
```

二. 使用ISE开发TCP服务端程序

基于 ISE，开发 TCP 服务端程序将变得非常简单和高效。ISE 封装了琐碎的 socket APIs，解决了网络编程中的各种麻烦和问题，ISE 使开发者能从复杂易错的 socket 调用中解放出来，把精力集中在业务逻辑的实现上。在 ISE 中，开发者只需关注四个事件：

1) 连接的建立（IseBusiness::onTcpConnected）；
2) 连接的断开（IseBusiness::onTcpDisconnected）；
3) 读操作完成（IseBusiness::onTcpRecvComplete）；
4) 写操作完成（IseBusiness::onTcpSendComplete）。

在 ISE 中，通过 TCP 连接发送和接收数据也很简单，发送和接收操作都以异步模式进行。向 ISE 提交操作请求后，只需关注“读操作完成事件”和“写操作完成事件”即可，不用担心数据只发送了一半，或者只接收到半个数据包。

ISE 非常重视系统性能。ISE 以 Linux::EPoll 和 Windows::IOCP 为基础，经过合理的封装后，结合事件循环线程模型（每个线程轮循一个事件循环 + 多线程），并将基于 Reactor 的 Linux::EPoll 和基于 Proactor 的 Windows::IOCP 统一，最终向开发者提供 Proactor 模式的编程接口。

2.1 详述ISE的TCP并发模型

为了解释 ISE 的 TCP 并发机制，先从 IO 模型、Reactor 与 Proactor 两种模式的比较，以及常见 TCP 并发服务设计模型说起。

2.1.1 IO模型

系统 IO 可分为三种：

- 阻塞型；
- 非阻塞同步型；
- 非阻塞异步型。

阻塞型 IO 意味着控制权直到调用操作结束后才会回到调用者手中，在等待 IO 结果的过程中，调用者做不了其它任何事情。

相比之下，非阻塞同步型操作会立即返还控制权给调用者，调用者不需要等待，它从调用操作中获取两种结果，要么成功了，要么还需继续重试。比如 read() 操作，如果当前 socket 无数据可读，则立即返回 EWOULBLOCK/EAGAIN，告诉调用者“数据还未就绪，请稍后重试”。

非阻塞异步型操作和前两者不同，调用操作除了会立即返回控制权，还告诉调用者：“此申请已提交给系统，系统会使用另外的资源或线程去完成这项任务，请注意接收任务完成通知”。典型的例子是 Windows

的完成端口（IOCP）。

对比这三种 IO 模型，“非阻塞异步型”是性能最优、伸缩性最好的。

2.1.2 Reactor与Proactor

在高性能 IO 设计中，有两个著名的模式：Reactor 和 Proactor。其中 Reactor 模式基于同步 IO，而 Proactor 模式则基于异步 IO。

这两种模式都运用了IO多路复用（IO multiplexing）技术，而IO多路复用机制的核心是“多路事件选择器[1]（demultiplexor）”。多路事件选择器的作用是分发事件处理者感兴趣的事件。开发者首先要在多路事件选择器那里注册感兴趣的事件，并提供相应的事件处理器（event handlers）；当事件发生时，多路事件选择器会将这些事件分发给事件处理器。

在 Reactor 模式中，多路事件选择器等待某个事件发生（比如文件描述符可读写），当事件发生时，多路事件选择器把这个事件传给事先注册的事件处理器，再由后者来做实际的读写操作。

而在 Proactor 模式中，程序发起一个异步操作请求，而实际的工作由操作系统来完成。发起请求时，以读操作为例，需提供读数据大小、缓存区、以及这个请求完成后的事件处理器等信息。多路事件选择器得知这个请求后，它等待这个请求的完成，然后发送完成事件给相应的事件处理器。

下面再以读操作为例，以步骤分解方式看看这两种模式的区别。

Reactor 模式的做法：

- 事件处理者宣称它对某个 socket 上的“可读”事件感兴趣；
- 多路事件选择器等待这个事件的发生；
- 当事件发生时，多路事件选择器通知事件处理者；
- 事件处理者收到了通知，于是去执行 socket 的读操作。如果需要，再次重复以上步骤。

Proactor 模式的做法：

- 事件处理者直接提交一个读操作，并只对该操何时完成感兴趣；
- 多路事件选择器等待这个读操作的完成；
- 在多路事件选择器等待期间，操作系统已经开始了实际的读操作，它从目标读取数据并放入用户提供的缓存区，最后通知多路事件选择器：“任务完成”；
- 多路事件选择器通知事件处理者：“读操作已完成”；
- 事件处理者这时会发现，想要读的数据已经放在了缓存区中。如果有需要，再次重复以上步骤。

对比可以发现，Reactor 与 Proactor 的根本区别是：**真正的读写操作是由谁来完成的**。Reactor 模式需要应用程序自己读写数据；而 Proactor 模式中，真正的读写操作由操作系统代劳。

如前文所述，Linux::EPoll 基于 Reactor 模式，而 Windows::IOCP 基于 Proactor 模式，各有特色。因为模式的差异，在不同的平台下，开发者必须以不同的思维模式去设计程序。ISE 提供了一种融合了 Reactor 与 Proactor 两种模式的解决方案，对 Reactor 稍做演变，在“真正的读写操作是由谁来完成的”这个核心问题上，把真正读写操作任务由原本的应用程序交给 ISE，即由 ISE 来担任 Proactor 模式中操作系统的角色，这样 Reactor 便摇身变成了 Proactor。

通过这样的模式演变，ISE 便可以在两种平台下向开发者提供统一的编程模式，即 Proactor 模式。下面列出了 ISE 执行一个读操作的流程：

[1]又名“事件分离器”。

- 应用程序提交了一个读请求：connection->recv(...)；
- ISE 内部执行实际的读操作。在 Linux 平台下，由事件循环线程驱动，当 EPoll 显示为可读状态时，执行读操作，直到读取完用户期望的数据；在 Windows 平台下，直接向 IOCP 提交读操作请求，并由事件循环线程驱动，接收完成通知并酌情重试，直到读取完用户期望的数据；
- ISE 读取完用户期望的数据后，通过 IseBusiness::onTcpRecvComplete 通知应用程序；
- 应用程序这时发现，缓存中已经有了期望的数据，并且是完整的。

2.1.3 常见TCP并发模型

在网络服务器程序的运行环境中，一般都是一个服务器对应多个客户端。为了处理客户端的请求，对服务端程序的设计有很多要求。服务器程序为了能同时响应不同的客户端，产生了不同的并发策略，这些策略被归纳总结之后，称为并发模型。下面列出几种常见的基于 TCP 的并发模型。

(0) 循环服务器

(1) 即时创建型多进程并发模型

(2) 即时创建型多线程并发模型

(3) 预创建型多进程并发模型

(4) 预创建型多线程并发模型

(5) 基于 select 的小规模 IO 多路复用并发模型

(6) 基于 IO 多路复用的单线程并发模型

(7) 基于 IO 多路复用的单线程 IO 加工作线程池并发模型

(8) 基于 IO 多路复用的单事件循环多线程并发模型

(9) 基于 IO 多路复用的多事件循环多线程并发模型

(10) 基于 IO 多路复用的多事件循环多进程并发模型

(11) 基于 IO 多路复用的多事件循环多线程加工作线程池并发模型

模型（0） 循环服务器

该方案其实并不能实现并发，它只是一个循环服务器，一次只能服务于一个客户端。只有在这个客户端的所有请求满足后，服务器才可以继续后面的请求。如果有一个客户端占住服务器不放，其它客户端就都不能工作了，因此，TCP 服务器很少采用循环服务器模型，之所以在这儿列出来是为了与其它模型对比。该模型的工作流程为：

```
01 socket(...);
02 bind(...);
03 listen(...);
04 while(true)
05 {
06    accept(...);
07    process(...);
08    close(...);
09 }
```

模型(1) 即时创建型多进程并发模型

这是传统的 Unix 网络编程并发模型。每当接收到一个客户端的连接后，便立即 fork 出一个新的子进程来为这个客户端服务。服务完毕后，子进程退出。所以，这种模型也叫“process per connection”。这种模型并不适合并发量较大的场合，因为 fork 的开销太大。

模型(2) 即时创建型多线程并发模型

与即时创建进程相比，创建一个线程的开销要小得多。但因为受到操作系统对单个进程内线程数量的限制，该模型仍然无法适用于大并发量的情况。另外，线程数量太多，会给操作系统带来严重的线程切换开销。

模型(3) 预创建型多进程并发模型

这是对模型(1)的优化。为了避免不断地创建和销毁进程，该模型在程序初始化时预先创建好了一批子进程。Apache httpd 使用了这种模型。

模型(4) 预创建型多线程并发模型

这是对模型(2)的优化。为了避免不断地创建和销毁线程，该模型在程序初始化时预先创建好了一批线程。Apache httpd 除了使用模型(3)外，也使用了这种模型。

模型(5) 基于 select 的小规模 IO 多路复用并发模型

IO 多路复用(IO multiplexing)是为了避免进程或线程阻塞于某个特定的 IO 系统调用而产生的技术。IO 多路复用技术使得单个进程或线程可以同时关注多个 socket 的变化。Linux 平台下的相应机制有 select、poll、epoll；Windows 平台下则有 select、Overlapped IO、IOCP 等。该模型使用 select 作为 IO 多路复用的基础。它之所以被冠以“小规模”，是因为 select 调用能同时关注的 socket 的数量受到操作系统的限制：在 Linux 平台下，这个限制一般是 1024；而在 Windows 平台下则更小，一般是 64。该模型由于受到 select 的限制，所以并不实用。

模型(6) 基于 IO 多路复用的单线程并发模型

除模型(5)外，其它模型都不使用 select，而是采用伸缩性更佳的 Linux::EPoll 或 Windows::IOCP。在这种模型中，只有一个线程，既负责 IO，又负责业务逻辑计算，虽然能同时应对很多并发连接，但较难发挥多核威力。所以，这种模型适合 IO 密集型应用，而不太适合 CPU 密集型应用。

模型(7) 基于 IO 多路复用的单线程 IO 加工作线程池并发模型

在这个模型中，一个线程负责 IO，另增加一个固定大小的线程池（根据计算密集程度和 CPU 数量确定线程池的大小），用于业务逻辑计算。IO 线程接收到客户端的任务后，将任务转移至工作线程池，线程池完成任务后再交由 IO 线程完成最后的对客户端的应答。

这种模型很好地弥补了模型(6)的不足。既能利用 IO 多路复用处理并发连接，又能借助工作线程池充分发挥多核优势。在 IO 压力不大时，此模型能很好地满足一般需求。当 IO 压力大时，可考虑采用“多事件循环”方案，见模型(9)和模型(11)。

模型(8) 基于 IO 多路复用的单事件循环多线程并发模型

在 Reactor/Proactor 模式中，程序执行点由多路事件选择器（demultiplexor）开始， 运行至事件处理器，再返回至多路事件选择器，这个循环称为“事件循环（event loop)”。在具体实现中，这个循环中可能有事件发生，也可能没有。当没有事件发生时，实际上是一个什么也没做的空循环。事件循环的运转要由线程（或进程）来驱动。一个事件循环可以由一个线程来驱动，也可以由多个线程来驱动（事件发生时，多路事件选择器只将事件传递给一个线程)。反过来，一个线程无法同时驱动多个事件循环。

在这种模型中，只使用单个事件循环（即只有一个多路事件选择器)，但是有多个线程在驱动。这是典

型的 Windows::IOCP 的做法。在 IOCP 中，一般只创建一个“完成端口句柄”，同时会根据 CPU 核数创建若干个线程。这些线程都参与事件循环（调用 API：GetQueuedCompletionStatus）。这种模型与 Windows::IOCP 非常匹配，由于 Windows 操作系统内置了高效的异步机制，在即使只有一个“完成端口句柄”的情况下仍能高效地工作，而多线程可以充分利用 CPU 资源。

在多数场合下，这种模型能很好地工作。但由于单事件循环的特性（即单个事件循环中管理着所有并发连接），使得应用层很难区分不同连接的优先级；另外，单个连接上的不同请求会被分配到不同的线程，使得请求处理的顺序性很难得到保证。

模型(9) 基于 IO 多路复用的多事件循环多线程并发模型

基于目前操作系统的性能，对于一个网络程序，一个事件循环 [1]就足以应付千兆以太网的网络IO。在一个程序使用多个事件循环更多的是基于其它方面的考虑，比如利用多事件循环来区别对待不同优先级的连接。

这种模型就是这样，它使用了多个事件循环（即多个多路事件选择器），并为每个事件循环只分配一个线程。事件循环数量与线程数量相等，数量大小一般由 CPU 核数确定。

这种模型中，多线程模式保证了对 CPU 资源的充分利用；并发 IO 请求由多个事件循环共同承担，避免突发 IO 导致单个事件循环的负载饱和；另外，一个连接被分配到一个事件循环后，完全由一个线程管理，保证了连接上请求处理的顺序性。与模型(7)相比，由于此模型使用了多线程 IO，所以在处理突发 IO 时更能应对自如，虽然少了工作线程池，但小规模计算仍可以在当前 IO 线程中进行。此外，由于使用了多个事件循环，能够处理连接的优先级，可以将优先级高的连接分配到单独的事件循环中。

综上，这是一种适应性很强的并发模型。同时也是 ISE 的默认 TCP 并发模型。

模型(10) 基于 IO 多路复用的多事件循环多进程并发模型

与模型(9)相比，这个模型将线程换成了进程。Nginx 采用的正是这个模型。ISE 不支持此模型。

模型(11) 基于 IO 多路复用的多事件循环多线程加工作线程池并发模型

在模型(9)在基础上增加工作线程池，就形成了这种并发模型。这种模型适用于既有大规模 IO 并发请求，又存在大量的计算任务的情形。如同模型(7)的做法一样，IO 线程并不参与计算，而是把计算任务转移到工作线程池中。ISE 提供了对这一模型的支持。

2.1.4 ISE的TCP并发模型

在上节提出的 12 种并发模型中，实用的模型有以下四种：

- **模型(6)** 基于 IO 多路复用的单线程并发模型；
- **模型(7)** 基于 IO 多路复用的单线程 IO 加工作线程池并发模型；
- **模型(9)** 基于 IO 多路复用的多事件循环多线程并发模型；
- **模型(11)** 基于 IO 多路复用的多事件循环多线程加工作线程池并发模型。

这四种实用模型在 ISE 中都得到了完善的支持。具体使用方法，详见「并发模型的选择」。

[1] 一个事件循环，在 Linux 平台下意味着只使用一个 EPoll fd；在 Windows 平台下意味着只创建一个 IOCP handle。

2.2 基于ISE开发TCP服务端程序

在了解 ISE 的 TCP 并发模型后，本节将介绍基于 ISE 开发 TCP 服务端程序的技术细节。ISE 作为一个程序框架，为开发者提供便利的同时，也提出了一些合理的“规定”，比如 TCP 连接的管理方式、发送和接收数据的方式等。开发者必须遵循这些规则，才能写出正确的程序。

2.2.1 与TCP相关的几个角色简介

在 ISE 中，与 TCP 相关的角色有：

- TcpConnection
  封装了一个 TCP 连接。开发者的主要工作是和它打交道，在连接上可以进行数据的发送和接收操作。它的对象依靠 boost::shared_ptr 管理（TcpConnectionPtr）。ISE 和开发者共同控制 TCP 连接对象的生命期。
- TcpServer
  TCP 的服务端，由 ISE 内部创建和管理，开发者可以无视它。
- TcpClient
  TCP 的客户端，应用程序主动发起连接时会用到它，由 ISE 内部使用，开发者同样可以无视它。
- TcpConnector
  TCP 连接器，用于以客户端身份主动发起连接。开发者在应用程序中发起连接时可调用 iseApp().tcpConnector().connect(...)。连接成功后，自动创建的 TCP 连接（TcpConnection）会自动转移到事件循环中。
- EventLoop
  事件循环。ISE 内部管理，开发者可以无视它。
- InetAddress
  封装了基于 IPv4 的 IP 和端口号。值语义。程序中会经常用到它，比如开发者可通过 TcpConnection::getPeerAddr() 来取得此连接中对方的地址。

总之，虽然与 TCP 相关的角色有好几个，但开发者只需关注 3 个，那就是：TcpConnection、TcpConnector 和 InetAddress。

2.2.2 发送和接收数据

对于网络程序而言，最重要的事情莫过于发送和接收数据了。基于传统 socket 网络编程，开发者常常要为看上去简单的发送和接收花费不少心思，非阻塞（non-blocking IO）模式下数据读写并不是一件简单的事情。

TCP 是基于字节流的协议，而在字节流协议上做应用层分包常常是网络编程的基本需求。所谓“分包”，指的是在发送一个消息时，通过一定的处理，让接收方能从字节流中识别并截取出一个完整的消息。TCP 协议本身并没有“应用层分包”的概念和机制，这些工作一定是由应用层来完成的。所以，所谓的“TCP 粘包”问题并不能称作一个问题。

前文已提到，在 ISE 中，应用程序首先通过下面的接口获得了一个连接：

```
void IseBusiness:onTcpConnected(const TcpConnectionPtr& connection);
```

现在程序要在连接（connection）上发送或接收数据了，发送数据的函数原型如下：

```
void TcpConnection::send(
    const void *buffer,
    size_t size,
    const Context& context = EMPTY_CONTEXT,
    int timeout = TIMEOUT_INFINITE
    );
```

函数说明：

发送操作是一个异步任务。send() 调用会立即返回，真正的发送操作由 ISE 完成。当数据发送完成时，ISE 通过 IseBusiness::onTcpSendComplete() 通知应用程序。如果发送过程遇到错误（比如网络中断），ISE 会断开连接，调用 IseBusiness::onTcpDisconnected()，并最终销毁连接对象。对应用程序而言，不需要处理“数据只发了一半，还需要继续发送下一半”的情况。在 ISE 中，一个发送任务要么成功（发送完成），要么失败（连接会被断开）。

参数说明：

- buffer
  要发送数据的缓存区。
- size
  要发送数据的字节数。
- context
  由于发送操作是一个异步任务，发送完成通知需要等到 IseBusiness::onTcpSendComplete()，为了能在 onTcpSendComplete() 中能知道是哪个任务完成了，可以在 send() 时给这个任务标记一个“上下文”。Context 对象是一个 boost::any，它可以存入任何值语义的数据。如果不需要上下文，要以传入 EMPTY_CONTEXT，这是一个“空上下文对象”。
- timeout
  通过此参数指定该发送任务的超时时间（单位为毫秒）。如果在指定时限内数据还没有发送完成，ISE 会主动断开此连接（IseBusiness::onTcpDisconnect() 会被调用）。此参数缺省为 TIMEOUT_INFINITE，表示无限时。

注意事项：

调用 send() 提交一个发送任务后，在此任务完成之前，如果主动断开了连接，比如调用了 connection->disconnect()，数据并不会完整发送，这时连接会被马上断开，剩余未发数据会被丢弃。

下面是接收数据的函数原型：

```
void TcpConnection::recv(
    const PacketSplitter& packetSplitter = ANY_PACKET_SPLITTER,
    const Context& context = EMPTY_CONTEXT,
    int timeout = TIMEOUT_INFINITE
    );
```

函数说明：

和发送操作一样，接收操作也是一个异步任务。recv() 调用会立即返回，真正的接收操作由 ISE 完成。

当接收完成时，ISE 通过 IseBusiness::onTcpRecvComplete() 通知应用程序。如果接收过程遇到错误（比如网络中断），ISE 会断开连接，调用 IseBusiness::onTcpDisconnected()，并最终销毁连接对象。应用程序在调用 recv() 时，已经通过 packetSplitter 参数告知了期望的分包规则。只有接收到了完整的数据包，才能算是“接收完成”。所以，对于应用程序而言，不需要处理“只接收到数据包的一半，还需要继续接收下一半”的情况。在 ISE 中，一个接收任务要么成功（接收完成），要么失败（连接会被断开）。

参数说明：

- packetSplitter
  在 ISE 中，PacketSplitter 被称为“分包器”。如前所述，应用层分包是网络编程的基本需求。ISE 支持自动分包机制，这使得开发者的在接收数据方面的思维负担大大减轻。关于分包，由于篇幅较大，详见后文。
- context
  由于接收操作是一个异步任务，接收完成通知需要等到 IseBusiness::onTcpRecvComplete()，为了能在 onTcpRecvComplete() 中能知道是哪个任务完成了，可以在 recv() 时给这个任务标记一个“上下文”。Context 对象是一个 boost::any，它可以存入任何值语义的数据。如果不需要上下文，要以传入 EMPTY_CONTEXT，这是一个“空上下文对象”。
- timeout
  通过此参数指定该接收任务的超时时间（单位为毫秒）。如果在指定时限内数据还没有接收完成，ISE 会主动断开此连接（IseBusiness::onTcpDisconnect() 会被调用）。此参数缺省为 TIMEOUT_INFINITE，表示无限时。

注意事项：

在 ISE 内部，无论应用程序有没有提交接收任务，都会尝试接收数据，并将收到数据存入接收缓存中。这种做法可以提高网络收发效率，并且在对方主动断开连接情况下，我方能尽快察觉（read 返回 0）。但是，ISE 并非无休止地接收数据，在当前没有任何接收任务的情况下，ISE 会给每个连接的接收缓存设定最大容量。最大容量值可由 IseBusiness::initIseOptions() 配置，缺省为 1M Bytes.

下面说说分包。分包器（PacketSplitter）的定义如下：

```
typedef boost::function<void (
    const char *data,    // 缓存中可用数据的首字节指针
    int bytes,           // 缓存中可用数据的字节数
    int& retrieveBytes   // 返回分离出来的数据包大小，返回 0 表示现存数据中尚不足以分离出一个完整数据包
)> PacketSplitter;
```

正确的分包需要发送方和接收方配合进行，双方约定一个一致的规则。发送方按照约定的规则对数据进行“打包”，而接收方遵循同样的规则进行“分包”。PacketSplitter 便代表了这样的规则。从定义可以看出，PacketSplitter 其实是一个 boost::function，它可以是一个纯函数、成员函数或仿函数。

在执行接收任务时，ISE 每收到一点数据，都会调用 packetSplitter（recv() 事先已传入），如果 packetSplitter 的 retrieveBytes 参数返回了一个大于 0 的值，说明缓存中已经有了一个完整的数据包，它的大小是 retrieveBytes。ISE 正是根据这样简单的规则来进行分包。

ISE 只对 PacketSplitter 做了定义，具体的分包规则需由应用程序来实现。这么做的原因是很显然的：ISE 并不懂“业务逻辑”。当然，作为程序框架，ISE 内置实现了几个常用的分包器：

- BYTE_PACKET_SPLITTER
  这个分包器认为每个字节都是一个数据包。

- LINE_PACKET_SPLITTER
  这个分包器识别换行符（CR 或 LF 或其组合），认为每一行文本都是一个数据包。
- NULL_TERMINATED_PACKET_SPLITTER
  这个分包器以 ‘\0’ 为分界进行分包。
- ANY_PACKET_SPLITTER
  这个分包器最特别，它只要一收到数据便认为是一个数据包。

应用程序要实现自己的分包器非常简单，只需要实现一个符合 PacketSplitter 定义的函数（或成员函数、或仿函数），再用 boost::bind 传给 TcpConnection::recv() 即可。

作为一个示例，下面列出了 NULL_TERMINATED_PACKET_SPLITTER 的实现：

```
01 void nullTerminatedPacketSplitter(const char *data, int bytes, int& retrieveBytes)
02 {
03    const char DELIMITER = '\0';
04
05    retrieveBytes = 0;
06    for (int i = 0; i < bytes; ++i)
07    {
08       if (data[i] == DELIMITER)
09       {
10          retrieveBytes = i + 1;
11          break;
12       }
13    }
14 }
```

2.2.3 连接的断开与销毁

连接的断开分两种：主动断开和被动断开。主动断开指应用程序主动调用了断开指令（socket API: close/shutdown）；而被动断开指对方先断开了连接，被我方检测到（socket API: read 返回 0）。

在 ISE 中，如果要主动断开连接，可以用下面的方法：

- TcpConnection::disconnect();
  此函数会立即关闭连接的“发送”通道。由于“接收”通道仍未关闭，所以此时程序仍可以接收数据。正常情况下，对方在检测到我方关闭发送后（read 返回 0），应关闭连接。这样我方接着会 read 到 0，从而关闭“接收”通道。这是“优雅关闭”的常见做法，完整流程是：
  我方发完了数据 -> 关闭发送 -> 对方 read 返回 0 -> 对方关闭连接 -> 我方 read 返回 0 -> 我方关闭接收 -> 连接完全关闭然后销毁。
- TcpConnection::shutdown(bool closeSend, bool closeRecv);
  应用程序如果希望自行控制连接的关闭过程，可以调用这个函数。它提供的两个参数 closeSend 和 closeRecv 分别代表是否关闭发送、和是否关闭接收。

在 ISE 中，有两种情况会导致 ISE 主动断开连接并双向关闭：

a) 发送或接收超时（即在调用 send()/recv() 时使用了 timeout 参数）；
b) 程序退出前关闭剩余的连接。

需要注意的是，无论是 disconnect() 还是 shutdown()，都会立即执行相应操作，而不管该连接上有没有未完成的发送任务，未完成的任务会被丢弃。如果希望在发送任务完成后再主动断开连接，可以在

IseBusiness::onTcpSendComplete() 中调用 connection->disconnect()。

另外，对于网络服务程序而言，不能不注意另外一个问题：主动断开 TCP 连接会给服务器留下 TIME_WAIT 问题。这是 TCP 协议栈使然。此问题有不同的解决办法，比如让客户端主动断开连接。

ISE 会检测对方是否断开了连接，当发现对方已断开时，会进入被动断开连接的流程。IseBusiness::onTcpDisconnected() 会被调用，之后连接将被销毁。这种情形属于被动断开，服务器并不会产生 TIME_WAIT。

连接双向关闭后，对应用程序来说已没有意义，ISE 会尝试销毁这个连接对象。由于 ISE 中连接对象依靠 boost::shared_ptr 管理，所以如果这时应用程序仍在持有该对象，对象将不会立即销毁，直到应用程序放开了该对象。由此可见，TcpConnection 对象的生命期由 ISE 和应用程序共同管理，只有当双方都认为该对象可以销毁时，对象才会销毁。

2.2.4 主动发起连接

在某些情况下，服务端程序也需要主动发起 TCP 连接，比如从一台服务器连接到另一台服务器。这里需要用到 TcpConnector。

应用程序可以通过 iseApp().tcpConnector() 来取得此对象。要主动发起连接，可调用 TcpConnector::connect() 方法。下面是 connect() 的原型：

```
void TcpConnector::connect(
    const InetAddress& peerAddr,
    const CompleteCallback& completeCallback,
    const Context& context = EMPTY_CONTEXT
    );
```

参数说明：

- peerAddr
  对方的地址。
- completeCallback
  连接完毕回调。无论连接是否成功都会进行回调，可通过回调函数中的参数 success 判断连接是否成功。回调函数定义如下：

  ```
  typedef boost::function<void (
      bool success,
      TcpConnection *connection,
      const InetAddress& peerAddr,
      const Context& context
  )> CompleteCallback;
  ```

- context
  对 connect() 的调用是一个异步操作，连接结果要在 completeCallback 中才能获知。通过 context 传入的上下文信息，将在回调函数的 context 参数中体现。Context 对象是一个 boost::any，可存放任意具有值语义的信息。如果不需要上下文，可忽略此参数。

连接成功后，ISE 会产生一个新的 TcpConnection 对象，和 TCP 服务器“被动”接受到的连接一样，这个“主动”产生的新连接也会自动交由 ISE 的事件循环（event loop）来管理。所以，和之前一样，IseBusiness::onTcpConnected() 会被调用。从这时开始，一切都和“被动”连接的操作方式相同。

通过一个简单的方法，可以在 onTcpConnected() 中获知这个新连接的来源：

- connection->isFromServer() 返回 true，说明它来自 TCP 服务器。
- connection->isFromClient() 返回 true，说明它是 ISE 主动发起的，即来自 TCP 客户端。

另外，上文提到主动产生的连接也会自动交给事件循环来管理，需要注意的是，用于接管主动连接的事件循环与 TCP 服务器的事件循环是相互独立的，两者并不会串用。应用程序可以在 IseBusiness::initIseOptions() 中通过 options.setTcpClientEventLoopCount(int count) 来设置用于主动连接的事件循环的数量，缺省数量为 1。

示例程序 ise/examples/chargen_client 演示了主动发起 TCP 连接的用法。

2.2.5 并发模型的选择

前文「ISE 的 TCP 并发模型」已提到，ISE 支持四种实用并发模型。这一节介绍如何在 ISE 中选择这些并发模型。ISE 是一个高度可配置化的程序框架。通过不同的配置参数可实现应用程序常规设置、不同环境下的性能调优、并发模型的选择等等。

ISE 配置的入口在：

```
virtual void IseBusiness::initIseOptions(IseOptions& options);
```

应用程序覆写这个虚函数，修改 options 参数的值，即可实现对各种配置参数的修改。下面是通过设置不同的参数来选择不同的并发模型的方法。

模型(6) “基于 IO 多路复用的单线程并发模型”的实现：

```
options.setServerType(ST_TCP);
options.setTcpServerEventLoopCount(1);  // 1 个事件循环，即单线程。
```

模型(7) “基于 IO 多路复用的单线程 IO 加工作线程池并发模型”的实现：

配置参数同模型(6)，应用程序中创建 ThreadPool 对象，实现工作线程池。ThreadPool 是 ISE 中的一个类（class），它基于生产者消费者队列（blocking queue）机制，实现多线程并发工作。

模型(9) “基于 IO 多路复用的多事件循环多线程并发模型”的实现：

```
options.setServerType(ST_TCP);
options.setTcpServerEventLoopCount(3);  // 此处的“3”为事件循环数量，可配置。
```

模型(11) “基于 IO 多路复用的多事件循环多线程加工作线程池并发模型”的实现：

配置参数同模型(9)，应用程序中创建 ThreadPool 对象，实现工作线程池。

2.2.6 多个TCP服务器的实现

有些情况下，需要在一个网络服务程序中同时开启多个 TCP 服务（即监听多个端口），比如一个 TCP 服务用于客户端业务，一个 TCP 服务用于运维（状态监视）。

为了实现多个 TCP 服务，可以采用下面的配置：

```
options.setServerType(ST_TCP);
options.setTcpServerCount(2);          // 2 个 TCP 服务
options.setTcpServerPort(0, 10001);    // 监听 10001 端口
```

```
options.setTcpServerPort(1, 10002);   // 监听 10002 端口
```

在 IseBusiness::onTcpConnected() 中，如何知道 connection 是来自哪个 TCP 服务呢？

```
01 void AppBusiness::onTcpConnected(const TcpConnectionPtr& connection)
02 {
03     int port = connection->getServerPort();
04     switch (port)
05     {
06     case 10001:
07         ...
08         break;
09     case 10002:
10         ...
11         break;
12     }
13 }
```

以上方法虽然可行，但并不完美。更好的方法是使用 ISE 的“服务模块（Server Module）”，每个服务模块对应一个 TCP 服务。关于服务模块，详见「服务模块（Server Modules）」。

2.3 实现一个简单的echo服务

在大致了解了用 ISE 开发 TCP 服务程序的技术细节后，这一节我们来利用 ISE 开发一个简单的 echo 服务程序。我们要开发的这个 echo 服务端的“业务逻辑”如下：

- 服务端监听 TCP 端口：10000；
- 客户端连上来之后，服务端立即输出一条欢迎信息，并等待客户端的输入；
- 服务端收到客户端输入的信息和回车后，将收到的数据原封不动地发回给客户端，循环往复；
- 如果服务端 5 秒内还没有收到完整的客户端输入，则认为超时而断开连接；
- 服务端收到客户端输入“quit”和回车后，主动断开连接，服务终止。

首先编写头文件：

echo_server.h

```
01 #ifndef _ECHO_SERVER_H_
02 #define _ECHO_SERVER_H_
03
04 #include "ise/main/ise.h"
05
06 using namespace ise;
07
08 class AppBusiness : public IseBusiness
09 {
10 public:
11     AppBusiness() {}
12     virtual ~AppBusiness() {}
13
14     virtual void initialize();
15     virtual void finalize();
16
17     virtual void afterInit();
18     virtual void onInitFailed(Exception& e);
```

```
19
20      virtual void initIseOptions(IseOptions& options);
21
22      virtual void onTcpConnected(const TcpConnectionPtr& connection);
23      virtual void onTcpDisconnected(const TcpConnectionPtr& connection);
24      virtual void onTcpRecvComplete(const TcpConnectionPtr& connection,
25          void *packetBuffer, int packetSize, const Context& context);
26      virtual void onTcpSendComplete(const TcpConnectionPtr& connection,
27          const Context& context);
28  };
29
30  #endif // _ECHO_SERVER_H_
```

例行公事地，首先从 IseBusiness 类继承，实现自己的 AppBusiness 类，并覆写感兴趣的虚函数。关于 IseBusiness 各个接口的说明，详见「IseBusiness 接口」，此处不再赘述。

一般而言，initIseOptions() 总是要被覆写的。echo_server 是一个 TCP 服务程序，所以与 TCP 相关的四个虚函数也是要被覆写的。

现在来看看实现部分。

echo_server.cpp

```
01  #include "echo_server.h"
02
03  IseBusiness* createIseBusinessObject()
04  {
05      return new AppBusiness();
06  }
07
08  // class AppBusiness
09
10  const int RECV_TIMEOUT = 1000*5;  // ms
11
12  void AppBusiness::initialize()
13  {
14      // nothing
15  }
16
17  void AppBusiness::finalize()
18  {
19      string msg = formatString("%s stoped.", getAppExeName(false).c_str());
20      std::cout << msg << std::endl;
21      logger().writeStr(msg);
22  }
23
24  void AppBusiness::afterInit()
25  {
26      string msg = formatString("%s started.", getAppExeName(false).c_str());
27      std::cout << std::endl << msg << std::endl;
28      logger().writeStr(msg);
29  }
30
31  void AppBusiness::onInitFailed(Exception& e)
32  {
33      string msg = formatString("fail to start %s.", getAppExeName(false).c_str());
34      std::cout << std::endl << msg << std::endl;
35      logger().writeStr(msg);
```

```
36 }
37
38 void AppBusiness::initIseOptions(IseOptions& options)
39 {
40     options.setServerType(ST_TCP);
41     options.setTcpServerCount(1);
42     options.setTcpServerPort(10000);
43     options.setTcpServerEventLoopCount(1);
44 }
45
46 void AppBusiness::onTcpConnected(const TcpConnectionPtr& connection)
47 {
48     logger().writeFmt("onTcpConnected (%s) (ConnCount: %d)",
49         connection->getPeerAddr().getDisplayStr().c_str(),
50         connection->getServerConnCount());
51
52     string msg = "Welcome to the simple echo server, type 'quit' to exit.\r\n";
53     connection->send(msg.c_str(), msg.length());
54 }
55
56 void AppBusiness::onTcpDisconnected(const TcpConnectionPtr& connection)
57 {
58     logger().writeFmt("onTcpDisconnected (%s)", connection->getConnectionName().c_str());
59 }
60
61 void AppBusiness::onTcpRecvComplete(const TcpConnectionPtr& connection,
62     void *packetBuffer, int packetSize, const Context& context)
63 {
64     logger().writeStr("onTcpRecvComplete");
65
66     string msg((char*)packetBuffer, packetSize);
67     msg = trimString(msg);
68     if (msg == "quit")
69         connection->disconnect();
70     else
71         connection->send((char*)packetBuffer, packetSize);
72
73     logger().writeFmt("Received message: %s", msg.c_str());
74 }
75
76 void AppBusiness::onTcpSendComplete(const TcpConnectionPtr& connection,
77     const Context& context)
78 {
79     logger().writeStr("onTcpSendComplete");
80     connection->recv(LINE_PACKET_SPLITTER, EMPTY_CONTEXT, RECV_TIMEOUT);
81 }
```

在第 *3* 行，例行公事地实现了一个名为 createIseBusinessObject 的函数，该函数创建了一个 AppBusiness 对象并返回。这是 ISE 对程序的要求。

在第 *17*、*24*、*31* 行，程序分别在 finalize、afterInit、onInitFailed 中输出了一些信息。

第 *38* 行，在 initIseOptions() 中进行了必要的设置。

第 *46-54* 行，在 onTcpConnected() 中接收到客户端的连接，所以调用 connection->send() 发送一行欢迎文本。

按照业务逻辑中的说明，欢迎文本发送完毕后，需要接收客户端的输入。所以第 *76-81* 行，在

onTcpSendComplete() 中调用了 connection->recv()。接收客户端的输入以回车为分界符进行分包，所以程序直接采用了 ISE 预定义的 LINE_PACKET_SPLITTER 分包器。另外，recv() 的第 3 个参数传入了常量 RECV_TIMEOUT，这个常量在第 *10* 行被定义为 5 秒，这是服务端接收输入的超时时间。

在收到客户端的完整数据包（一行文本）后，在 onTcpRecvComplete() 中第 *71* 行，调用 send() 把收到的数据原封不动地发回给客户端。而如果收到的文本是“quit”，在第 *69* 行，则调用 disconnect() 主动断开连接。

在调用 connection->send() 把收到数据原封不动地发回客户端后，当数据发送完成时，程序再次执行了 onTcpSendComplete()。而在 onTcpSendComplete() 中程序再次调用了 connection->recv()，接收客户端的输入，程序逻辑进入循环。

在 ise/examples/echo_server 目录下可以查看这个示例程序的源码。

三. 使用ISE开发UDP服务端程序

与 TCP 服务相比，ISE 中 UDP 服务的实现机制要简单一些。ISE 只支持在一个应用程序中创建单个 UDP 服务器（即只监听一个端口），并允许开启多个监听线程接收 UDP 数据包，采用生产者消费者队列（Blocking Queue）方式将 UDP 数据包分发给工作线程池。由于 UDP 协议中数据包存在确定边界的特性，应用层并不需要考虑像 TCP 那样分包机制。

对于简单的 UDP 服务端程序而言，开发者一般只需关注 UDP 数据包接收事件，即：IseBusiness::onRecvedUdpPacket。对性能要求较高的 UDP 服务端程序，需要考虑按照业务重要程度对 UDP 请求包进行分组，即 IseBusiness::classifyUdpPacket。在程序中发送 UDP 数据包也很简单，只需调用 iseApp().udpServer().sendBuffer() 函数。

3.1 UDP服务端工作原理

ISE 采用了相对简单的 UDP 并发机制。总的来说就是：监听线程池 + 生产者消费者队列 + 动态工作者线程池。

3.1.1 监听线程池

为了处理高并发 UDP 请求，ISE 在 UDP 监听端采用了线程池模式。线程池中线程数量固定，数量可通过 IseBusiness::initIseOptions() 配置。如下：

```
options.setUdpListenerThreadCount(n);
```

缺省情况下，ISE 只为线程池创建 1 个线程。实际应用中如果存在高并发，可根据情况为线程池配置多个线程。

3.1.2 生产者消费者队列

在监听线程（listener thread）与工作者线程（worker thread）之间，ISE 采用生产者消费者队列（Blocking Queue）作为联结两者的桥梁。生产者消费者队列是一个数据结构，它由生产者（即监听线程）放入资源（收到的 UDP 请求包），而由消费者（即工作者线程）取出资源。生产者和消费者都可能是多个线程，当队列中没有数据包时，工作者线程进入等待，直至监听线程再次放入了新的数据包。工作者线程每次只取出一个数据包，既而通过 IseBusiness::onRecvedUdpPacket() 通知应用程序。在下文中，生产者消费者队列通常被称为“请求队列”，而队列中的数据包称为“请求包”。

ISE 提供了一些与队列相关的配置。

- 队列的最大容量（即可容纳多少个数据包）。

```
void IseOptions::setUdpRequestQueueCapacity(
    int groupIndex,         // 请求组别序号 (0-based)
```

```
    int capacity            // 队列最大容量
    );
```

- UDP 数据包在队列中的最长等待时间（秒），超时则不予处理。

```
void IseOptions::setUdpRequestMaxWaitTime(int seconds);
```

- 队列中数据包数量警戒线。线程池根据此警戒线来判断是否需要动态递增工作者线程数量。

```
void IseOptions::setUdpRequestQueueAlertLine(int count);
```

3.1.3 动态工作者线程池

ISE 的 UDP 服务使用动态工作者线程池。动态工作者线程池的责任是：负载自适应 和 线程僵死与超时检测。

- **负载自适应**

在实际应用中，网络请求在不同的时段内其密集程度可能不尽相同，甚至差距甚大。当密集程度大时，服务器的负载自然随之变大，这时如果线程池难以应付不断到来的请求，则应该创建更多的工作者线程，以便能及时处理这些请求；当密集程度小时，服务器的负载随之变小，这时为了节省服务器的系统资源，应适量减少工作者线程的数量。这种调节动作在服务器程序的运行过程中持续地进行着，我们称之为负载自适应。

工作者线程数量的递增和递减并不是无数量限制的。在 ISE 配置中，可以设定工作者线程数量的上下限：

```
void IseOptions::setUdpWorkerThreadCount(int groupIndex, int minThreads, int maxThreads);
```

一般来说，线程数的上限不宜太高，太多的线程将会导致系统开销增大，操作系统不得不花大量的时间用于线程切换。ISE 的缺省设置中，下限为 1，上限为 8。

请求队列中现存的请求数量，通常指示着当前服务的处理能力。当现存的请求数量偏大，则说明服务器的负载在增大，此时应考虑递增工作者线程数量，从而提高服务质量。对于每个请求队列，都有有一个“警戒线”。线程池根据此警戒线来判断是否需要递增工作者线程数量。此警戒线的值可通过“ISE 配置”来设置。

ISE 尽量以较少的线程来满足服务的需要。当服务刚启动时，线程数为最低下限。只有当系统检测到负载增大，才递增线程数量。

- **线程僵死与超时的检测**

线程开始工作后，由于多种原因，可能会造成僵死或超时。在 ISE 中，僵死和超时是两个不同的概念：

  - 僵死线程：已被通知退出但长久不退出的线程。
  - 超时线程：因某一请求进入工作状态但长久未完成的线程。

线程僵死一般发生在系统递减工作者线程数时，被通知退出的线程由于多种原因而长久不退出。对于这类僵死的线程，它们既不工作，又占用了线程数量，严重妨碍负载自适应工作。线程池会强行杀死它们。

而线程超时则稍有不同，当工作者线程收到一个请求后，马上进入工作状态。一般而言，工作者线程为单个请求持续工作的时间不宜太长，若太长则会导致服务器空闲工作者线程短缺，使得应付并发请求的能力下降。尤其对于 UDP 服务来说情况更是如此。通常情况下，线程工作超时，很少是因为程序的流程和逻辑，而是由于外部原因，比如数据库繁忙、资源死锁、网络拥堵等等。当线程工作超时后，线程池会通

知其退出，若被通知退出后若干时间内仍未退出，则该线程的身份变为僵死线程，从而被看成僵死线程来对待。

在 ISE 中的缺省配置中，UDP 工作者线程全部启用了超时检测，超时值为 60 秒。应用程序可通过下面的方法统一设置每个线程的超时值：

```
void IseOptions::setUdpWorkerThreadTimeout(int seconds);
```

须注意的是，在某些场合下，由于业务需要，工作者线程的确要为单个请求正常持续工作很长时间。对于这类工作者线程，应单独禁用它的超时检测。单独设置线程超时值的方法如下：

```
void AppBusiness::onRecvedUdpPacket(UdpWorkerThread& workerThread, int groupIndex,
    UdpPacket& packet)
{
    // 禁用超时检测。
    workerThread.getTimeoutChecker().setTimeoutSecs(0);
    // 启用超时检测，超时值为 20 秒。
    workerThread.getTimeoutChecker().getTimeoutSecs(20);
}
```

3.2 高并发UDP服务程序设计

本节介绍了 ISE 中高并发 UDP 服务程序设计的要点：请求分组和参数调优。

3.2.1 对UDP请求包进行分组（Request Grouping）

一般来说，在高密集度的 UDP 请求中，由于业务需要，总会存在一些“比较重要”的请求。以即时通讯服务为例，简化起见，假设只存在保活（keep alive）请求和登录（login）请求。按请求的重要程度来比较，login 远比 keep alive 重要；但按请求的密集程度来比较，keep alive 显然远比 login 要密集（一个用户只 login 一次，但须持续地定时 keep alive）。在这种情况下，如果 login 和 keep alive 请求包以 1:50 的数量比率到达服务器，login 请求难免会被 keep alive 请求淹没。这里所谓的淹没，是指服务器在定量的负载能力下，由于服务器同等看待每个请求包，导致“数量较少的那一类”的请求更容易被延迟或疏忽。

为了解决这个问题，ISE 用“UDP 请求分组”来分隔不同重要程度的请求。在每个分组中，有独立的请求队列和工作者线程池。这样一来，用户可以根据业务需要，将 UDP 请求按重要程度或者逻辑模块划分成若干个组别。不同组别的 UDP 请求到达服务器后，有独立的“迎候”和“处理”机制。只要组别划分得当，便可将淹没现象减到最少。

如何进行 UDP 请求分组呢？可分成两个步骤：

首先，用户可以在“服务器配置”中指定“UDP 请求的组别总数”：

```
void IseOptions::setUdpRequestGroupCount(int count);
```

其次，覆写 IseBusiness::classifyUdpPacket() 函数，此函数用于对 UDP 数据包进行分类。函数原型如下：

```
virtual void classifyUdpPacket(void *packetBuffer, int packetSize, int& groupIndex);
```

在 ISE 中，组别号皆从 0 算起，如果组别总数为 N，那么组别号 I ∈ [0, N-1]。

一般来说，用户定义的数据包中，都会有一个域用于表达此包的用途（比如：是 login 还是 keep alive），我们习惯上称此域为“动作代码（action code）”。用户可在 classifyUdpPacket 函数中依据数据包的动作代码来决定它的组别号。须注意的是，计算组别号的算法应尽量简单，UDP 数据包的分类工作是由 UDP 监听线程来负责的，太过复杂的算法必然影响 UDP 监听线程的循环周期，从而降低 UDP 服务的并发性能。

另外，对于不合法的数据包，可以将其组别号设为 –1，这样，ISE 会自动丢弃该数据包，不会放入任何请求队列中。所以，classifyUdpPacket 函数还兼有 UDP 数据包过滤的作用。

下面这段示例代码演示了如何在 classifyUdpPacket() 中对数据包进行分类：

```
01 void AppBusiness::classifyUdpPacket(void *packetBuffer, int packetSize,
02    int& groupIndex)
03 {
04    groupIndex = -1;
05
06    PacketHeader* header = (PacketHeader*)packetBuffer;
07    if (packetSize < sizeof(PacketHeader)) return;
08
09    switch (header->actionCode)
10    {
11    case CS_REGISTER:
12       groupIndex = 0;
13       break;
14
15    case CS_LOGIN:
16    case CS_LOGOUT:
17       groupIndex = 1;
18       break;
19
20    case CS_KEEP_ALIVE:
21       groupIndex = 2;
22       break;
23    }
24 }
```

3.2.2 设置相关参数

ISE 中有诸多配置用于优化 UDP 服务的性能，下面分别列出这些配置，并说明这些配置的用途及设置方法。这些配置皆可通过重写 IseBusiness::initIseOptions() 函数来设置（详见「ISE 参数配置」）。

- `void setUdpListenerThreadCount(int count);`

说明：设置 UDP 监听线程的数量。

- `void setUdpRequestGroupCount(int count);`

说明：设置 UDP 请求的组别总数。

- `void setUdpRequestQueueCapacity(int groupIndex, int capacity);`

说明：设置 UDP 请求队列的最大容量（即可容纳多少个数据包）。若队列容量已满，则丢弃队列中最先到来的那个请求，再将刚到来的请求追加到队列尾部。对于高密集度的请求组别，可以将队列容量设得

大些。队列实际所占内存只是按需分配，并非一开始就占用最大容量。

- `void setUdpRequestMaxWaitTime(int seconds);`

说明：设置 UDP 请求在队列中的最长等待时间（秒），超时则不予处理。一般来说，客户端发出请求后会在一定时间内等候服务器端响应，若在指定时间内没有任何响应，则认为请求失败。但若服务端并未丢弃该请求，只因积压过久，造成客户端等待超时，而服务端在客户端已经等待超时之后再处理请求并应答，此应答其实已无任何用处，反倒浪费了服务器资源。所以此处的“最长等待时间”用于避免服务器响应已经过期的请求，此值一般等于客户端等待超时时限。

- `void setUdpRequestQueueAlertLine(int count);`

说明：设置 UDP 请求队列中数据包数量警戒线。系统在进行负载自适应时，会检测请求队列中当前请求的个数，若超过警戒线则尝试增加线程。

- `void setUdpWorkerThreadCount(int groupIndex, int minThreads, int maxThreads);`

说明：设置 UDP 工作者线程个数的上下限。对于高密集度的请求组别，可以将线程数量上限设得大些，但也应注意不宜太大，以免线程过多造成系统开销太大。

- `void setUdpWorkerThreadTimeout(int seconds);`

说明：设置 UDP 工作者线程的工作超时时间（秒），若为 0 表示不进行超时检测。关于工作者线程超时，「动态工作者线程池」节已详细讨论，此处不再赘述。

- `void setUdpAdjustThreadInterval(int seconds);`

说明：设置后台调整工作者线程数量的时间间隔（秒）。此参数用于设置负载自适应的检测周期。

3.3 实现一个简单的UDP服务

本节以一个示例程序来简述 UDP 服务端程序的开发过程。我们要开发的这个 UDP 服务端程序的“业务逻辑”如下：

- 服务端监听 UDP 端口：8000；
- 客户端可以向服务端发送“hello”数据包；
- 服务端在收到“hello”数据包后，向客户端发送应答包；

业务逻辑非常简单。首先定义数据包格式：

ise/examples/udp_server/packet_defs.h

```
01 #ifndef _PACKET_DEFS_H_
02 #define _PACKET_DEFS_H_
03
04 #include "ise/main/ise.h"
05
06 using namespace ise;
07
08 // Action Codes:
09
```

```
10 const UINT AC_HELLO = 100;
11 const UINT AC_ACK   = 200;
12
13 // UDP Packet Header
14
15 #pragma pack(1)
16
17 class UdpPacketHeader
18 {
19 public:
20    UINT actionCode;      // action code
21    UINT seqNumber;       // auto increment
22 public:
23    void init(UINT actionCode, UINT seqNumber = 0)
24    {
25       static SeqNumberAlloc seqNumAlloc;
26       this->actionCode = actionCode;
27       this->seqNumber = (seqNumber == 0 ? seqNumAlloc.allocId() : seqNumber);
28    }
29 };
30
31 #pragma pack()
32
33 // class BaseUdpPacket
34
35 class BaseUdpPacket : public Packet
36 {
37 public:
38    UdpPacketHeader header;
39 protected:
40    virtual void doPack() { writeBuffer(&header, sizeof(header)); }
41    virtual void doUnpack() { readBuffer(&header, sizeof(header)); }
42 };
43
44 // HelloPacket
45
46 class HelloPacket : public BaseUdpPacket
47 {
48 public:
49    string message;
50 protected:
51    virtual void doPack()
52    {
53       BaseUdpPacket::doPack();
54       writeString(message);
55    }
56
57    virtual void doUnpack()
58    {
59       BaseUdpPacket::doUnpack();
60       readString(message);
61    }
62
63 public:
64    void initPacket(const string& message)
65    {
66       header.init(AC_HELLO);
```

```
67          this->message = message;
68          pack();
69      }
70  };
71
72  // AckPacket
73
74  class AckPacket : public BaseUdpPacket
75  {
76  public:
77      INT32 ackCode;
78  protected:
79      virtual void doPack()
80      {
81          BaseUdpPacket::doPack();
82          writeINT32(ackCode);
83      }
84
85      virtual void doUnpack()
86      {
87          BaseUdpPacket::doUnpack();
88          ackCode = readINT32();
89      }
90
91  public:
92      void initPacket(UINT seqNumber, INT32 ackCode)
93      {
94          header.init(AC_ACK, seqNumber);
95          this->ackCode = ackCode;
96          pack();
97      }
98  };
99
100 #endif // _PACKET_DEFS_H_
```

数据包格式采用“固定首部”+“变长数据体”方式。第 *17* 行定义了数据包的首部，首部中有一个关键字段“actionCode”，即动作代码，它代表了数据包的“业务类别”。在本例中，数据包有两种：“hello 数据包”和“应答包”。在第 *10*、*11* 行定义了这两种动作代码。

第 *46* 行定义了 HelloPacket，它的数据体部分由一个字符串构成。第 *74* 行定义了 AckPacket，它的数据体部分是一个整数应答码。

接下来编写头文件：

ise/examples/udp_server/udp_server.h

```
01  #ifndef _UDP_SERVER_H_
02  #define _UDP_SERVER_H_
03
04  #include "ise/main/ise.h"
05
06  using namespace ise;
07
08  // class AppBusiness
09
10  class AppBusiness : public IseBusiness
11  {
12  public:
```

```
13      AppBusiness() {}
14      virtual ~AppBusiness() {}
15
16      virtual void initialize();
17      virtual void finalize();
18
19      virtual void afterInit();
20      virtual void onInitFailed(Exception& e);
21
22      virtual void initIseOptions(IseOptions& options);
23
24      virtual void onRecvedUdpPacket(UdpWorkerThread& workerThread, int groupIndex,
25          UdpPacket& packet);
26  };
27
28  #endif // _UDP_SERVER_H_
```

例行公事地，首先从 IseBusiness 类继承，实现自己的 AppBusiness 类，并覆写感兴趣的虚函数。关于 IseBusiness 各个接口的说明，详见「IseBusiness 接口」，此处不再赘述。

一般而言，initIseOptions() 总是要被覆写的。udp_server 是一个 UDP 服务程序，所以还覆写了 onRecvedudpPacket()。

现在来看看实现部分。

ise/examples/udp_server/udp_server.cpp

```
01  #include "udp_server.h"
02  #include "packet_defs.h"
03
04  IseBusiness* createIseBusinessObject()
05  {
06      return new AppBusiness();
07  }
08
09  // class AppBusiness
10
11  void AppBusiness::initialize()
12  {
13      // nothing
14  }
15
16  void AppBusiness::finalize()
17  {
18      string msg = formatString("%s stoped.", getAppExeName(false).c_str());
19      std::cout << msg << std::endl;
20      logger().writeStr(msg);
21  }
22
23  void AppBusiness::afterInit()
24  {
25      string msg = formatString("%s started.", getAppExeName(false).c_str());
26      std::cout << std::endl << msg << std::endl;
27      logger().writeStr(msg);
28  }
29
30  void AppBusiness::onInitFailed(Exception& e)
31  {
```

```
32      string msg = formatString("fail to start %s.", getAppExeName(false).c_str());
33      std::cout << std::endl << msg << std::endl;
34      logger().writeStr(msg);
35  }
36
37  void AppBusiness::initIseOptions(IseOptions& options)
38  {
39      options.setServerType(ST_UDP);
40      options.setUdpServerPort(8000);
41  }
42
43  void AppBusiness::onRecvedUdpPacket(UdpWorkerThread& workerThread, int groupIndex,
44      UdpPacket& packet)
45  {
46      void* packetBuffer = packet.getPacketBuffer();
47      int packetSize = packet.getPacketSize();
48
49      if (packetSize < (int)sizeof(UdpPacketHeader))
50      {
51          logger().writeStr("Invalid packet.");
52          return;
53      }
54
55      UINT actionCode = ((UdpPacketHeader*)packetBuffer)->actionCode;
56
57      switch (actionCode)
58      {
59      case AC_HELLO:
60          {
61              HelloPacket reqPacket;
62              if (reqPacket.unpack(packetBuffer, packetSize))
63              {
64                  logger().writeFmt("Received packet from <%s>: %s.",
65                      packet.getPeerAddr().getDisplayStr().c_str(),
66                      reqPacket.message.c_str());
67
68                  AckPacket ackPacket;
69                  ackPacket.initPacket(reqPacket.header.seqNumber, 12345);
70                  iseApp().udpServer().sendBuffer(ackPacket.getBuffer(),
71                      ackPacket.getSize(), packet.getPeerAddr());
72              }
73              break;
74          }
75
76      default:
77          {
78              logger().writeStr("Unknown packet.");
79              break;
80          }
81      }
82  }
```

在第 *4* 行，例行公事地实现了一个名为 createIseBusinessObject 的函数，该函数创建了一个 AppBusiness 对象并返回。这是 ISE 对程序的要求。

在第 *16*、*23*、*30* 行，程序分别在 finalize、afterInit、onInitFailed 中输出了一些信息。

第 *37* 行，在 initIseOptions() 中进行了必要的设置。

第 *43* 行，程序开始处理收到的 UDP 请求包。

第 *55* 行从数据包中解析出相应的动作代码，如果是 AC_HELLO，则在第 *62* 行交给 HelloPacket 进行数据解码（unpack），如果解码成功，说明收到的数据包合法。

第 *68-69* 行，构造了一个应答包。接下来调用 iseApp().udpServer().sendBuffer() 将应答包发送给对方。

在 ise/examples/udp_server 目录下可以查看这个示例程序的源码。

四. 更进一步了解ISE

前文介绍了使用 ISE 开发网络服务程序的基础知识，了解这些基础后，可以容易地开发 TCP 服务或 UDP 服务程序。然而 ISE 的内容并不仅限于此，开发者要充分利用 ISE，还需进一步了解 ISE 的其它重要特性。比如服务模块，利用它可以使服务端程序更好地模块化；辅助线程帮助开发者以最简单统一的方式创建和管理线程；通过 ISE 扩展可以大大丰富 ISE 的功能；而数据库接口（DBI）可以帮助开发者在不同平台以统一的方式访问数据库，等等。

4.1 服务模块（Server Modules）

在实际应用中，网络服务端程序一般由若干个较为独立的功能模块组成，比如即时通讯服务端，可能由客户端服务、页面调用服务、状态监视服务等几个部分组成，这些模块往往彼此相对独立，耦合性不高。ISE 针对这种现象，在框架级别提供了“服务模块”机制。

4.1.1 服务模块工作原理

在 ISE 中，服务模块机制是一个独立的“子系统”。为了实现服务模块机制，ISE 接管了 IseBusiness 接口，派生为 IseSvrModBusiness，并将原本在 IseBusiness 中的很多事件接口（如 TCP/UDP 事件）转移到了 IseServerModule 中。IseServerModule 是服务模块的基类，应用程序中的每个服务模块都必须从此类继承。

程序初始化时，ISE 通过 IseSvrModBusiness::createServerModules() 创建所有服务模块的实例。当事件发生时，遍历全部服务模块并调用它们的事件接口。

4.1.2 基于服务模块的编程方式

基于服务模块的编程方式与之前介绍的编程步骤稍有不同，主要应注意以下几点：

- AppBusiness 不再从 IseBusiness 类继承，而是从 IseSvrModBusiness 继承；
- 每个服务模块从 IseServerModule 继承；
- 部分“ISE 配置”不再在 IseBusiness::initIseOptions() 中设置，而是通过覆写 IseServerModule 类的某些虚函数来设置；
- UDP 与 TCP 的事件回调从 IseBusiness 转移到了 IseServerModule 类中；
- 覆写 IseSvrModBusiness::createServerModules()，在其中创建全部服务模块。

4.1.3 服务模块编程示例

示例程序 ise/examples/server_modules 展示了详细的使用方法和细节。它在一个程序中集成了 echo、daytime、discard 和 chargen 四个不同的 TCP 服务，每个服务对应一个服务模块。

示例程序 ise/examples/server_module_msgs 展示了在不同服务模块间发送消息的方法。

由于服务模块的编程方式比较简单，此处不详细讲述。

4.2 辅助线程（Assistor Threads）

在实际应用中，服务程序除了提供网络服务外，还需要在后台周期性地执行某些与网络服务无关的事务，比如内存数据的定期保存、垃圾数据清理、数据维护等。这些事务有时可能执行时间比较长，不便于放在网络服务线程中执行。这时，开发者往往需要创建单独的线程专门用于处理这类事务。

为了满足这类需求，ISE 提供了“辅助线程”机制。所谓辅助线程，就是指服务程序中，除了 TCP 事件循环线程、UDP 工作者线程之外的第三类线程，它的创建和销毁由 ISE 负责，而它的具体功能由应用程序自由定义。缺省情况下，ISE 不会创建辅助线程。应用程序可在初始化时通过 ISE 配置指定需要创建的辅助线程的数量：

```
void IseOptions::setAssistorThreadCount(int count);
```

辅助线程的执行入口点在 IseBusiness 提供的接口中：

```
virtual void IseBusiness::assistorThreadExecute(
    AssistorThread& assistorThread,
    int assistorIndex
    );
```

应用程序可通过覆写 IseBusiness::assistorThreadExecute() 接管辅助线程的执行。每个辅助线程都有其自身的索引号（assistor index），索引号从 0 算起。

需注意的是，应用程序应在线程执行函数中判断 assistorThread.isTerminated()，当它返回 true 时应及时退出线程，以便程序退出时 ISE 能及时终止线程。

在服务模块中使用辅助线程的方式稍有不同。在服务模块方式下，不需要在 ISE 配置中指定辅助线程数量，也不可以覆写 IseBusiness::assistorThreadExecute()，取而代之的是 IseServerModule 中的两个虚函数：

```
// 返回此模块所需辅助服务线程的数量
virtual int getAssistorThreadCount() { return 0; }
// 辅助服务线程执行(assistorIndex: 0-based)
virtual void assistorThreadExecute(AssistorThread& assistorThread, int assistorIndex) {}
```

4.3 服务状态监视（Server Inspector）

在运维工作中，我们常常借助第三方监控工具来监视服务的状态。需要监视的状态有很多，比如 CPU 负载、内存占用、网络带宽、硬盘空间等等。可是这些工具却很难监视程序内部的运行状态，比如每秒处理请求的数量、有多少请求超时或被拒绝等。如果能让服务端程序本身通过一种易于操作的方式（比如 HTTP 方式）向外输出这些数据，那么服务程序状态的监视将变得更加简单。

ISE 支持这种方式。它内建了一个简单的 HTTP Server，并允许应用程序从 IseServerInspector 继承，实现自己的监视服务模块。简单地说，服务状态监视功能的开发步骤如下：

- 以“服务模块”方式设计程序，见「服务模块（Server Modules）」；
- 创建一个服务模块，从 IseServerInspector 继承，在构造函数中指定 HTTP 服务端口号；

- 调用 IseServerInspector::add(...)，添加监视项目。

添加监视项目只需调用 IseServerInspector::add()，它的定义如下：

```
void IseServerInspector::add(
    const string& category,
    const string& command,
    const OutputCallback& outputCallback,
    const string& help
    );
```

参数说明：

- category
  监视项目可能会有很多，为了显示得更有条理，在此可以指定一个分类名。
- command
  监视项目的命令名称。此名称可根据监视项目的意义自由命名。比如 status、cpu、file_count 等。
- outputCallback
  输出监视项目的实际内容。当用在浏览器中打开状态监视页面时，ISE 会调用这个回调，输出相应信息。
- help
  为这个监视项目提供一段简单的帮助文本，以便用户知道它的意义。

这就是服务状态监视开发的全部。最终，用户可以在浏览器中输入 http://192.168.0.100:8080，查看当前服务状态。IP 和端口请换成相应的服务器真实 IP 和监视服务模块的端口。

示例程序 ise/examples/server_inspector 展示了开发监视服务的细节。

4.4 多线程环境编程基础设施

ISE 是一个多线程框架。在多线程环境中编程需要面对很多问题，比如线程的创建、管理和退出、同步原语的正确使用、发现和解决各种竞态条件（race condition），锁争用（lock contention）的处理和死锁的避免等等。为了降低多线程环境下的编程复杂度，ISE 提供了一系列跨平台的用于多线程环境编程的基础设施。

- Thread
  线程的封装类。它有两种使用方式：程序可以继承它，实现一个新的线程类；或者调用 Thread::create() 并传入一个 boost::function。ISE 为开发者提供了完善的线程管理机制，所以应用程序一般不需要直接使用 Thread 类。
- Mutex
  互斥锁。它基于 Linux::pthread_mutex_t 和 Windows::CriticalSection 来实现，都属于多线程环境下的轻量级互斥锁（采用 futex 机制）。应用程序可能会经常用到它。互斥锁的使用通常需要结合 RAII 手法（见下文中的 AutoLocker）。
- Condition
  条件变量。在 Linux 平台下它基于 pthread_cond_t 来实现；而在 Windows 平台下，由于 Vista 之前的系统并未直接支持，所以只能用“更重量级的内核对象 Mutex”+“信号量”+

“SignalObjectAndWait”来模拟。条件变量一般用于 BlockingQueue 之类的生产者消费者队列。

- Semaphore
  信号量。这是一种较为古老的同步原语，为了解决生产者-消费者问题而设计。信号量自身维护了一套计数器，用于表示可用资源的数量，这往往与应用程序中数据结构蕴含的计数重复，造成信息冗余。在实际应用中，信号量并不是必备的同步原语，因为条件变量完全可以替代它。
- AtomicInt
  原子整数。在需要对整数进行原子地增减时，可以使用它。它比“用互斥锁保护整数的操作”要高效得多。同时提供的还有 AtomicInt64。
- AutoLocker
  基于 RAII 的互斥锁调用。众所周知，在 C++中基于 RAII 手法对资源进行保护是一种通行做法。程序可以在 scope 中定义“AutoLocker locker(mutex);”便可以实现对此 scope 的加锁和解锁。
- ThreadPool
  基于任务的线程池。程序可以向线程池放入任务，而线程池不断地完成这些任务。每个任务都是一个 boost::function。ThreadPool 一般用于与并发模型中的 IO 线程搭配，完成某些 CPU 密集型的计算任务。
- BlockingQueue
  阻塞式的生产者消费者队列。生产者向队列中放入资源，而消费者从队列中取出资源。当队列中没有资源可取时，消费者进入等待，直到生产者再次放入了新的资源。BlockingQueue 是一个模板类，允许程序通过模板参数指定资源类型。

4.5 ISE扩展（ISE Extensions）

ISE 由两大部分构成：ISE 主程序 和 ISE 扩展。

ISE 主程序是 ISE 的主体，实现了完整的网络服务框架（ise/ise/main）。而 ISE 扩展则是服务端开发常见需求的功能集合（ise/ise/ext），目前它包含了数据库接口、常用加解密和散列算法集和 XML。ISE 扩展是独立的（源码独立、静态库独立），虽然它包含了许多杂项功能，但并不影响 ISE 主程序的纯洁性。

下面简单介绍 ISE 扩展目前提供的功能。

- **数据库接口**
  源码目录：ise/ise/ext/dbi
  ISE 主程序中定义了数据库接口标准（ise_database.*），而具体的接口实现则由 ISE 扩展来完成。目前 ISE 实现了 MySQL 的接口（ise/ise/exe/dbi/mysql）。
  详细介绍见「对数据库的支持」。
- **常用加解密和散列算法集**
  源码目录：ise/ise/ext/utils/cipher
  此算法集提供的算法有：base64、crc32、MD4/MD5、SHA/SHA1、Blowfish、IDEA、DES、Gost。
  包含头文件：ise/ext/utils/cipher/ise_cipher.h
  静态库文件：libise_utils_cipher.a （Windows 下的扩展名为 .lib）

具体使用方法请参考头文件 ise_cipher.h。

- **XML**
  源码目录：ise/ise/ext/utils/xml
  此扩展出于跨平台的目的，对常用 XML 操作进行了简单封装。
  包含头文件：ise/ext/utils/cipher/ise_xml.h
  静态库文件：libise_utils_xml.a （Windows 下的扩展名为 .lib）
  具体使用方法请参考头文件 ise_xml.h。

4.6 对数据库的支持

实际应用中，很多服务端程序需要访问数据库，为此 ISE 提供了一套通用的数据库访问接口（DBI），并允许开发者基于该接口标准实现各种具体的数据库访问接口。ISE 扩展中实现了 MySQL 接口。

4.6.1 ISE数据库接口（DBI）

ISE 的数据库接口简称 DBI（Database Interface）。它的架构借鉴了 BDE 和 JDBC。下面列出 DBI 的全部类和相关说明。

- DbException
  数据库异常类。所有的数据库操作异常都使用该类或它的派生类。
- DbConnParams
  数据库连接参数类。连接参数包括：数据库主机地址、主机端口号、用户名、用户口令、数据库名。
- DbOptions
  数据库配置参数类。包括：连接池最大连接数、数据库刚建立连接时要执行的命令、数据库刚建立连接时要设置的字符集。
- DbConnection
  数据库连接基类。此类封装了一个数据库连接，数据库连接由连接池管理。
- DbConnectionPool
  数据库连接池基类。连接池中维护一个连接列表，管理当前的所有连接（空闲连接、忙连接），初始为空。连接池负责连接的分配与归还。
- DbFieldDef
  字段定义类。此类用于描述一个字段的定义信息，比如字段名、字段长度等。
- DbFieldDefList
  字段定义列表类。此类用于管理一个表中所有的字段定义对象。
- DbField
  字段数据类。此类封装了一个数据行中某个字段的数据。并提供诸如 asString()、asInteger() 等一系列存取方法。
- DbFieldList
  字段数据列表类。此类用于管理一个数据行中的所有字段数据对象。

- DbParam
  SQL 参数类。此类用于设置一个指定的 SQL 参数。
- DbParamList
  SQL 参数列表类。此类用于管理一条 SQL 语句中所有的参数对象。
- DbDataSet
  数据集类。此类封装了一个数据集。ISE 只提供了数据集的单向遍历功能。
- DbQuery
  数据查询器类。此类用于执行一个 SQL 查询，查询可以返回结果（如 select），也可以不返回结果（如 delete, update…）。若返回结果，则自动创建一个 DbDataSet 对象并返回。
- Database
  数据库工厂类。此类管理“参与访问一个物理数据库”的所有类的对象。同时它起着类工厂的作用。

虽然数据库接口涉及到的类比较多，但对开发者而言，最常用的莫过于三个类：DbQuery、DbDataSet 和 DbField。下面列出了这三个类的主要成员函数。

- **DbQuery**：
  - `void setSql(const string& sql);`
    说明：在执行查询之前，先设置 SQL 语句。
  - `DbParam* paramByName(const string& name);`
    说明：根据名称取得参数对象。
  - `DbParam* paramByNumber(int number);`
    说明：根据序号 (1-based) 取得参数对象。
  - `void execute();`
    说明：执行 SQL （无返回结果，若失败则抛出异常）。
  - `DbDataSet* query();`
    说明：执行 SQL （返回数据集，若失败则抛出异常）。
  - `string escapeString(const string& str);`
    说明：转换字符串使之在 SQL 中合法 （str 中可含 '\0' 字符）。
  - `UINT getAffectedRowCount();`
    说明：取得执行 SQL 后受影响的行数。
  - `UINT64 getLastInsertId();`
    说明：取得最后一条插入语句的自增 ID 的值。

- **DbDataSet**：
  - `bool rewind();`
    说明：将游标指向起始位置（第一条记录之前）。
  - `bool next();`
    说明：将游标指向下一条记录。须注意的是，数据集最初状态为未指向任何记录，须调用 Next 才可指向第一条记录。

- `UINT64 getRecordCount();`
  说明：取得当前数据集中的记录总数。此操作是否耗时，取决于具体实现。
- `bool isEmpty();`
  说明：返回数据集是否为空。
- `int getFieldCount();`
  说明：取得当前记录中的字段总数。
- `DbFieldDef* getFieldDefs(int index);`
  说明：取得当前记录中某个字段的定义（index: 0-based）。
- `DbField* getFields(int index);`
  说明：根据字段下标号取得当前记录中某个字段的数据（index: 0-based）。
- `DbField* getFields(const string& name);`
  说明：根据字段名取得当前记录中某个字段的数据。

- **DbField**：
  - `bool isNull();`
    说明：返回此字段值是否为 NULL。
  - `int asInteger(int defaultVal = 0);`
    说明：以 32 位整型返回字段值（若转换失败则返回缺省值）。
  - `INT64 asInt64(INT64 defaultVal = 0);`
    说明：以 64 位整型返回字段值（若转换失败则返回缺省值）。
  - `double asFloat(double defaultVal = 0);`
    说明：以浮点型返回字段值（若转换失败则返回缺省值）。
  - `bool asBoolean(bool defaultVal = false);`
    说明：以布尔型返回字段值（若转换失败则返回缺省值）。
  - `string asString();`
    说明：以字符串型返回字段值。

4.6.2 MySQL接口

ISE 中提供的数据库接口（DBI）只是一份接口标准，它本身并不能用于访问任何数据库。针对不同类型数据库的具体实现工作由 ISE 扩展完成。目前 ISE 扩展中实现了针对 MySQL 的接口。

MySQL 接口的使用方法：

- 包含头文件：#include "ise/ext/dbi/mysql/ise_dbi_mysql.h"
- 链接静态库文件：libise_dbi_mysql.a （Windows 下扩展名为 .lib）
- 链接静态库文件：libmysqlclient.a （Windows 下为 mysqlclient.lib）

下面的示例展示了访问 MySQL 数据库的方法：

```
01 #include "ise/main/ise.h"
02 #include "ise/ext/dbi/mysql/ise_dbi_mysql.h"
03
04 using namespace ise;
```

```
05
06 ////////////////////////////////////////////////////////////////////////
07 // 数据库连接参数:
08
09 #define DB_HOST_NAME    "127.0.0.1"
10 #define DB_DB_NAME          "mydb"
11 #define DB_USER_NAME    "root"
12 #define DB_PASSWORD         ""
13
14 // 数据库配置参数:
15 const int MAX_DB_POOL_CONNS = 1000;    // 最大连接数
16
17 ////////////////////////////////////////////////////////////////////////
18 // class MyDbOperations
19
20 class MyDbOperations
21 {
22 public:
23     MyDbOperations();
24     bool test();
25 private:
26     MySqlDatabase database_;
27 };
28
29 ////////////////////////////////////////////////////////////////////////
30 // MyDbOperations 的实现部分:
31
32 MyDbOperations::MyDbOperations()
33 {
34     // 初始化数据库连接参数
35     database_.getDbConnParams()->setHostName(DB_HOST_NAME);
36     database_.getDbConnParams()->setUserName(DB_USER_NAME);
37     database_.getDbConnParams()->setPassword(DB_PASSWORD);
38     database_.getDbConnParams()->setDbName(DB_DB_NAME);
39
40     // 初始化数据库配置参数
41     database_.getDbOptions()->setMaxDbConnections(MAX_DB_POOL_CONNS);
42 }
43
44 //------------------------------------------------------------------------
45 // 描述: 测试一个简单的查询
46 //------------------------------------------------------------------------
47 void MyDbOperations::test()
48 {
49     try
50     {
51         // 创建一个查询器对象，并进行对象包装
52         DbQueryWrapper query(database_.createDbQuery());
53         // 创建一个数据集包装器
54         DbDataSetWrapper dataSet;
55
56         // 设置好待执行的 SQL 语句
57         query->setSql("SELECT (1+2) AS result");
58         // 执行 SQL 查询，并将结果数据集返回到 DataSet 包装器中
59         dataSet = query->query();
60         // 定位到首条记录
61         dataSet->next();
```

```
62
63          // 从数据集的当前行中中取出字段值，并检验是否正确
64          if (dataSet->getFields(0)->asInteger() == 3)
65              logger().writeStr("Test OK!");
66      }
67      catch(Exception& e)
68      {
69          logException(e);
70      }
71  }
```

4.6.3 开发更多的数据库接口

ISE 中的 DBI 是一个可扩展的接口。如果 ISE 扩展中提供的数据库接口不能满足需求，开发者可以自己现实新的数据库接口。一般来说，要现实新的数据库接口，应至少继承四个类：DbConnection、DbDataSet、DbQuery 和 Database。这些类中各有若干虚函数，用户可自行查看 ise_database.* 及源码注释，并结合 ISE 扩展中的预置数据库接口实现的源码，即可实现针对自己需要的数据库接口。

五. ISE编程示例

5.1 四个简单的TCP协议

本节介绍四个简单的 TCP 网络服务程序，它们是 echo（RFC 862）、discard（RFC 863）、daytime（RFC 867）和 chargen（RFC 864）。每五个示例程序将这四个服务集中到一个程序中。

echo

代码：ise/examples/echo_server

这是一个回显服务，服务端把客户端发过来的每行数据原封不动地发送回去。它需要关注 onTcpConnected、onTcpRecvComplete 和 onTcpSendComplete 三个事件，并需要用到 ISE 的分包器 LINE_PACKET_SPLITTER。

前文「实现一个简单的 echo 服务」已经详细解说过这个示例，此处不再重复。

discard

代码：ise/examples/discard_server

这是一个长连接协议，服务端在建立连接后丢弃所有收到的数据。它需要关注 onTcpConnected 和 onTcpRecvComplete 事件。此例用到了 ANY_PACKET_SPLITTER 分包器。关于分包器详见「发送和接收数据」节。

```
01 void AppBusiness::onTcpConnected(const TcpConnectionPtr& connection)
02 {
03     connection->recv(ANY_PACKET_SPLITTER);
04 }
05
06 void AppBusiness::onTcpRecvComplete(const TcpConnectionPtr& connection,
07     void *packetBuffer, int packetSize, const Context& context)
08 {
09     logger().writeFmt("[%s] Discarded %u bytes.",
10         connection->getConnectionName().c_str(), packetSize);
11
12     connection->recv(ANY_PACKET_SPLITTER);
13 }
```

daytime

代码：ise/examples/daytime_server

这是一个短连接协议，服务端在接受连接后以字符串形式发送当前系统时间，然后主动断开连接。它需要关注 onTcpConnected 和 onTcpSendComplete 事件。

```
01 void AppBusiness::onTcpConnected(const TcpConnectionPtr& connection)
02 {
```

```
03     string msg = DateTime::currentDateTime().dateTimeString() + "\n";
04     connection->send(msg.c_str(), msg.length());
05 }
06
07 void AppBusiness::onTcpSendComplete(const TcpConnectionPtr& connection,
08     const Context& context)
09 {
10     connection->disconnect();
11 }
```

用 telnet 作为客户端，运行结果如下：

```
# telnet 127.0.0.1 10002
2013-05-29 15:23:32
```

chargen

代码：ise/examples/chargen_server

这是一个长连接协议。服务端在接受连接后，不停地发送测试数据。它只发送数据，不接收数据或忽略收到的数据。它关注 onTcpConnected、onTcpRecvComplete 和 onTcpSendComplete 事件。

```
01 void AppBusiness::onTcpConnected(const TcpConnectionPtr& connection)
02 {
03     connection->setNoDelay(true);
04     connection->recv();
05     connection->send(message_.c_str(), message_.length());
06 }
07
08 void AppBusiness::onTcpRecvComplete(const TcpConnectionPtr& connection,
09     void *packetBuffer, int packetSize, const Context& context)
10 {
11     logger().writeFmt("[%s] Discarded %u bytes.",
12         connection->getConnectionName().c_str(), packetSize);
13     connection->recv();
14 }
15
16 void AppBusiness::onTcpSendComplete(const TcpConnectionPtr& connection,
17     const Context& context)
18 {
19     transferredBytes_ += message_.length();
20     connection->send(message_.c_str(), message_.length());
21 }
```

可结合 chargen_client 示例程序（ise/examples/chargen_client）来测试 chargen_server。

4in1

代码：ise/examples/server_modules

前面四个示例各自独立，互不影响。现在我们把这四个程序合并到一个程序中，这里需要用到“服务模块”（见「服务模块（Server Modules）」）。

这个新的程序（4in1）仍然只用到一个事件循环（即一个线程），但却监听了 4 个 TCP 端口，并同时提供了 4 种不同的网络服务。

限于篇幅，此处不列出源码。

5.2 简单的HTTP服务

代码：ise/examples/http_server

这是一个非常简单的 HTTP 服务的例子，它展示了编写 HTTP 服务程序的基本方法。客户端（浏览器）连上它后，它输出一行文本：“this is a simple http server.”。

编写 HTTP 服务程序的步骤是：

- 创建 HttpServer 对象；
- 向 HttpServer 对象分派 TCP 的四个事件；
- 处理 onHttpSession 事件。

下面列出示例程序的关键代码：

```
01 void AppBusiness::initIseOptions(IseOptions& options)
02 {
03     options.setServerType(ST_TCP);
04     options.setTcpServerCount(1);
05     options.setTcpServerPort(8080);
06     options.setTcpServerEventLoopCount(1);
07 }
08
09 void AppBusiness::onHttpSession(const HttpRequest& request, HttpResponse& response)
10 {
11     string content = "this is a simple http server.";
12
13     response.setStatusCode(200);
14     response.getContentStream()->write(content.c_str(), content.length());
15 }
```

5.3 服务状态监视器

代码：ise/examples/server_inspector

开发服务状态监视程序的方法详见「服务状态监视（Server Inspector）」。

在浏览器中输入 http://192.168.0.100:8080，可查看示例程序的当前状态。IP 地址请换成相应的服务器真实 IP。

5.4 工作者线程池

代码：ise/examples/thread_pool

此例展示了 ThreadPool 的使用方法。ThreadPool 是 ISE 提供的一种基于任务的线程池。程序可以向线程池放入任务，而线程池不断地完成这些任务。每个任务都是一个 boost::function。ThreadPool 一般用于与并发模型中的 IO 线程搭配，完成某些 CPU 密集型的计算任务。详见「多线程环境编程基础设施」。

本例在初始化时向线程池添加了一个任务，并设为重复任务（即线程池执行完这个任务后并不删除任务，而是重复执行）。线程池创建了 10 个工作线程。

```
01 void AppBusiness::initialize()
02 {
03     threadPool_.addTask(boost::bind(&AppBusiness::threadPoolTask, this, _1));
04     threadPool_.setTaskRepeat(true);
05     threadPool_.start(10);
06 }
07
08 void AppBusiness::finalize()
09 {
10     threadPool_.stop();
11 }
12
13 void AppBusiness::initIseOptions(IseOptions& options)
14 {
15     options.setIsDaemon(false);
16 }
17
18 void AppBusiness::threadPoolTask(Thread& thread)
19 {
20     static AtomicInt counter;
21
22     counter.increment();
23     string s = formatString("executing task: %u. (thread: %u)",
24         counter.get(), thread.getThreadId());
25     logger().writeStr(s);
26     std::cout << s << std::endl;
27
28     thread.sleep(1);
29 }
```

5.5 简单的UDP服务端

代码：ise/examples/udp_server

前文「实现一个简单的 UDP 服务」节讲解过本例，此处不再重复。

六. 附录

6.1 ISE参数配置

下面列出了全部 ISE 配置项和它的说明，以及缺省值。

- `void setLogFileName(const string& value, bool logNewFileDaily = false);`
  说明：设置日志文件名。logNewFileDaily 为 true 时每天新建一个日志文件。
  缺省：{app}/log/app_name.log
- `void setIsDaemon(bool value);`
  说明：设置是否后台守护程序（仅针对 Linux 平台）。
  缺省：true
- `void setAllowMultiInstance(bool value)`
  说明：设置是否允许多个程序实体同时运行。
  缺省：false
- `void setServerType(UINT serverType);`
  说明：设置服务器类型（ST_UDP 或 ST_TCP，或两者组合）。
  缺省：两者皆不选。
- `void setAssistorThreadCount(int count);`
  说明：设置辅助线程的数量。
  缺省：0

- `void setUdpServerPort(int port);`
  说明：设置 UDP 服务端口。
  缺省：8000
- `void setUdpListenerThreadCount(int count);`
  说明：设置 UDP 监听线程的数量。
  缺省：1
- `void setUdpRequestGroupCount(int count);`
  说明：设置 UDP 请求的组别总数。
  缺省：1
- `void setUdpRequestQueueCapacity(int groupIndex, int capacity);`
  说明：设置 UDP 请求队列的最大容量（即可容纳多少个数据包）。若队列容量已满，则丢弃队列中最先到来的那个请求，再将刚到来的请求追加到队列尾部。
  缺省：5000
- `void setUdpRequestMaxWaitTime(int seconds);`

说明：设置 UDP 请求在队列中的最长等待时间（秒），超时则不予处理。
缺省：10（秒）

- `void setUdpRequestQueueAlertLine(int count);`
  说明：设置 UDP 请求队列中数据包数量警戒线。系统检测到请求队列中当前请求数量超过警戒线则尝试增加线程。
  缺省：500
- `void setUdpWorkerThreadCount(int groupIndex, int minThreads, int maxThreads);`
  说明：设置 UDP 工作者线程个数的上下限。
  缺省：1 ~ 8
- `void setUdpWorkerThreadTimeout(int seconds);`
  说明：设置 UDP 工作者线程的工作超时时间（秒），若为 0 表示不进行超时检测。
  缺省：60（秒）
- `void setUdpAdjustThreadInterval(int seconds);`
  说明：设置后台调整 UDP 工作者线程数量的时间间隔（秒）。
  缺省：5（秒）

- `void setTcpServerCount(int count);`
  说明：设置 TCP 服务器的数量。
  缺省：1
- `void setTcpServerPort(int serverIndex, int port);`
  说明：设置 TCP 服务端口号。（serverIndex: 0-based）
  缺省：8000
- `void setTcpServerEventLoopCount(int serverIndex, int eventLoopCount);`
  说明：设置每个 TCP 服务器中事件循环的个数。
  缺省：1
- `void setTcpClientEventLoopCount(int eventLoopCount);`
  说明：设置用于全部 TCP 客户端的事件循环的个数。
  缺省：1
- `void setTcpMaxRecvBufferSize(int bytes);`
  说明：设置 TCP 接收缓存在无接收任务时的最大字节数。
  缺省：1M bytes

6.2 参考资料

- 《TCP/IP 详解 卷 1: 协议》（人民邮电出版社）
- 《UNIX 网络编程 第 1 卷：套接口 API （第 3 版）》（杨继张译. 清华大学出版社）
- 《Effective C++ 中文版（第 3 版）》（侯捷译. 电子工业出版社）
- 《Linux 多线程服务端编程 - 使用 muduo C++ 网络库》（陈硕. 电子工业出版社）
- 《Windows 网络编程技术》（Anthony Jones, Jim Ohlund. 机械工业出版社）
- 《代码整洁之道》（Robert C. Martin 人民邮电出版社）
- *Effective TCP/IP Programming*
  Jon C. Snader. Addison-Wesley
- *Effective STL*.
  Scott Meyers. Addison-Wesley, 2001
- *The Art of Multiprocessor Programming*
  Maurice Herlihy, Nir Shavit.
- *The Practice of Programming*
  Brian W. Kernihgan and Rob Pike.
- *C++ and the Perils of Double-Checked Locking*
  Scott Meyers and Andrei Alexandrescu, September 2004.
  http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- *Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events*
  Irfan Pyarali, Tim Harrison, and Douglas C. Schmidt Thomas D. Jordan
  http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf
- *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*
  Douglas C. Schmidt
  http://www.cs.wustl.edu/~schmidt/PDF/Reactor.pdf
- Borland VCL Source Code