Mooon 基础类库编程手册

一见 2011.7.11

1. 什么是 Mooon？ 7
1.1. 基础类库 7
1.2. 公共组件 7
1.3. 应用平台 8
1.4. 改进型 MapReduce 8
2. 代码风格 9
2.1. 类 9
2.2. 接口 9
2.3. 结构体 9
2.4. 变量/类变量 9
2.5. 文件名 9
2.6. 编译工具 9
3. 为何只支持 Linux 9
4. 如何编译基础类库？ 10
5. util 库 10
5.1. 库文件 10
5.2. 名字空间 10
5.3. 依赖库 11
5.4. delete_helper 和 free_helper 11
5.4.1. 主要功能 11
5.4.2. 成员函数 11
5.4.3. 示例代码 1 12
5.4.4. 示例代码 2 12
5.4.5. 示例代码 3 13
5.5. va_list_helper 13
5.5.1. 主要功能 13
5.5.2. 成员函数 13
5.5.3. 示例代码 13
5.6. CountHelper 14
5.6.1. 头文件 14
5.6.2. 主要功能 14
5.6.3. 成员函数 14
5.6.4. 示例代码 14
5.7. CIntegerUtil 15
5.7.1. 头文件 15
5.7.2. 主要功能 15
5.7.3. 成员函数 15
5.8. CStringUtil 类 16

5.8.1. 头文件 16
5.8.2. 主要功能 16
5.8.3. 成员函数 16
5.8.4. 示例代码 18
5.8.5. 示例代码 2 19
5.8.6. 示例代码 3 19
5.8.7. 示例代码 4 19
5.9. CTokenList 21
5.9.1. 头文件 21
5.9.2. 主要功能 21
5.9.3. 成员函数 21
5.9.4. 示例代码 21
5.10. CArrayQueue 22
5.10.1. 头文件 22
5.10.2. 主要功能 22
5.10.3. 成员函数 22
5.10.4. 示例代码 23
5.11. CListable 和 CListQueue 23
5.11.1. 头文件 23
5.11.2. 主要功能 23
5.11.3. 成员函数 24
5.12. CHistogramArray 25
5.12.1. 头文件 25
5.12.2. 主要功能 25
5.12.3. 注意事项 25
5.12.4. 成员函数 25
5.12.5. 示例代码 27
6. sys 库 27
6.1. 库文件 27
6.2. 名字空间 27
6.3. 依赖库 27
6.4. CSyscallException 28
6.4.1. 头文件 28
6.4.2. 主要功能 28
6.4.3. 成员函数 28
6.5. CRefCountable 29
6.5.1. 头文件 29
6.5.2. 主要功能 29
6.5.3. 成员函数 29
6.5.4. 示例代码 30
6.6. close_helper 30
6.6.1. 头文件 30
6.6.2. 主要功能 30
6.6.3. 成员函数 31

6.6.4. 示例代码 1：32
6.6.5. 示例代码 2：32
6.6.6. 示例代码 3：32
6.7. CThread33
6.7.1. 头文件33
6.7.2. 主要功能33
6.7.3. 成员函数33
6.7.4. 示例代码35
6.8. CThreadPool 和和 CPoolThread37
6.8.1. 头文件37
6.8.2. 主要功能37
6.8.3. 池线程和线程37
6.8.4. 成员函数38
6.8.5. 示例代码40
6.9. CLock 和 CRecLock42
6.9.1. 头文件42
6.9.2. 主要功能42
6.9.3. 成员函数42
6.9.4. 示例代码44
6.10. CEvent45
6.10.1. 头文件45
6.10.2. 主要功能45
6.10.3. 成员函数45
6.10.4. 示例代码46
6.11. CReadWriteLock47
6.11.1. 头文件47
6.11.2. 主要功能47
6.11.3. 成员函数47
6.12. CEventQueue50
6.12.1. 头文件50
6.12.2. 主要功能50
6.12.3. 成员函数50
6.12.4. 示例代码51
6.13. CMMap54
6.13.1. 头文件54
6.13.2. 主要功能54
6.13.3. 成员函数54
6.13.4. 示例代码57
6.14. CDynamicLinkingLoader58
6.14.1. 头文件58
6.14.2. 主要功能58
6.14.3. 成员函数58
6.14.4. 示例代码59
6.15. CDatetimeUtil59

6.15.1. 头文件 59
6.15.2. 主要功能 59
6.15.3. 成员函数 59
6.15.4. 示例代码 62
6.16. CFileUtil 63
6.16.1. 头文件 63
6.16.2. 主要功能 63
6.16.3. 成员函数 63
6.17. CFSUtil 64
6.17.1. 头文件 64
6.17.2. 主要功能 64
6.17.3. 成员函数 64
6.18. CFSTable 65
6.18.1. 头文件 65
6.18.2. 主要功能 65
6.18.3. 成员函数 65
6.19. CInfo 67
6.19.1. 头文件 67
6.19.2. 主要功能 67
6.19.3. 成员函数 67
6.20. CUtil 73
6.20.1. 头文件 73
6.20.2. 主要功能 73
6.20.3. 成员函数 73
6.21. CRawObjectPool 和 CThreadObjectPool 75
6.21.1. 头文件 75
6.21.2. 主要功能 75
6.21.3. CPoolObject 76
6.21.4. CRawObjectPool 76
6.21.5. CThreadObjectPool 77
6.22. CRawMemPool 和 CThreadMemPool 78
6.22.1. 头文件 78
6.22.2. 主要功能 78
6.22.3. CRawMemPool 79
6.22.4. CThreadMemPool 80
7. net 库 81
7.1. 库文件 81
7.2. 名字空间 81
7.3. 依赖库 81
7.4. 主要类图 82
7.5. ip_address_t 82
7.6. ipv4_node_t 和 ipv6_node_t 83
7.6.1. ipv4_node_hasher 84
7.6.2. ipv4_node_comparer 84

7.6.3. ipv6_node_hasher....84
7.6.4. ipv6_node_comparer....84
7.7. CEpollable....85
7.7.1. 头文件....85
7.7.2. 主要功能....85
7.7.3. 成员函数....85
7.7.4. 名字空间全局函数....87
7.8. CEpoller....89
7.8.1. 头文件....89
7.8.2. 主要功能....89
7.8.3. 成员函数....89
7.8.4. 示例代码....91
7.9. CListener....93
7.9.1. 头文件....93
7.9.2. 主要功能....93
7.9.3. 成员函数....93
7.9.4. 示例代码....94
7.10. CListenManager....96
7.10.1. 头文件....96
7.10.2. 主要功能....96
7.10.3. 成员函数....97
7.10.4. 示例代码....97
7.11. CTcpClient....98
7.11.1. 头文件....98
7.11.2. 主要功能....98
7.11.3. 成员函数....98
7.11.4. 示例代码 1....102
7.11.5. 示例代码 2....104
7.11.6. 示例代码 3....104
7.12. CTcpWaiter....106
7.12.1. 头文件....106
7.12.2. 主要功能....106
7.12.3. 成员函数....106
7.12.4. 示例代码....109
7.13. CTimeoutable 和 CTimeoutManager....110
7.13.1. 头文件....110
7.13.2. 类图....110
7.13.3. 主要功能....111
7.13.4. 适用场景....111
7.13.5. 成员函数....111
7.13.6. 示例代码....113
7.14. CUtil....114
7.14.1. 头文件....114
7.14.2. 主要功能....114

7.14.3. 成员函数 114
7.14.4. 示例代码 1：字节序转换 117
7.15. CEpollableQueue 118
7.15.1. 头文件 118
7.15.2. 主要功能 118
7.15.3. 应用场景 118
7.15.4. 成员函数 118
7.15.5. 示例代码 120
8. logger 122
8.1. 什么是 mooon logger？ 122
8.2. 类结构 123
8.3. ILogger 接口 123
8.4. 写日志宏 124
8.5. 工作原理 125
8.6. 日志格式 126
8.7. 示例代码 126
9. 数据库连接池 127
9.1. 简述 127
9.2. 相关头文件和库文件 127
9.3. 分层结构 127
9.4. 源码目录结构 128
9.4.1. 头文件 128
9.4.2. CPP 文件 128
9.4.3. 测试代码 129
9.5. DB API 129
9.5.1. 异常类 129
9.5.2. 抽象接口 130
9.5.3. 助手类 135
9.6. 示例 137
9.6.1. 源代码 137
9.6.2. 编译运行 139
10. XML 配置文件读取 140
10.1. 介绍 140
10.2. 主要功能 140
10.3. 相关文件 141
10.4. IConfigReader 接口 141
10.5. IConfigFile 143
10.6. ConfigReaderHelper 143
10.7. 全局配置变量 144
10.8. 示例代码 144

1. 什么是 Mooon?

Mooon 不是 Moon，中文意思非月，含有飞越之意，取其最简单的组合就是飞月。您可以通过 http://code.google.com/p/mooon 或 http://bbs.hadoopor.com/index.php?gid=67 了解更多和最新的内容。

Mooon 是一组基础类库、公共组件和应用平台等的总称，目标是输出高质量、高效率、易于使用、容易维护、可测试可观察的开源软件。

采用分层结构组织，以达到最大程度的软件复用度，以下是 Mooon 的一个分层组件图：

Mooon 提供两类业务平台：一是应用平台，用以支持各类常规业务的开发；另一个是计算平台，用于支撑海量数据的处理，计算平台是一个改进型的 MapReduce，但并不是一个 MapReduce，它不存在数据倾斜问题，以及具备类似于进程调度器一样的调度能力。

1.1. 基础类库

提供各类工具类，由三个主基础库和若干插件库组成，涉及：配置、日志、数据库、字符串处理、日期时间操作、线程/线程池、锁和事件、网络工具类、系统工具类等。

基础类库为最低层库，三个主基础库的依赖关系为：net --> sys --> util，基础插件库是对它们的实现，同时也依赖于这三个主基础库。

1.2. 公共组件

对常用功能模块的组件化，达到高可复用和易于组装的目的，如：TCP 服务端框架、HTTP 协议解析和消息分发等。 公共组件不但是应用平台的组成部分，也可以用于其它系统。

1.3. 应用平台

面向常规应用开发，同时支持 Service、Session 和 Object 三种开发模式，它们的不同点如下表所示：

	Service	Session	Object
是否有状态	√	√	✖
是否线程安全	✖	√	–
是否可动态创建和删除	✖	√	√

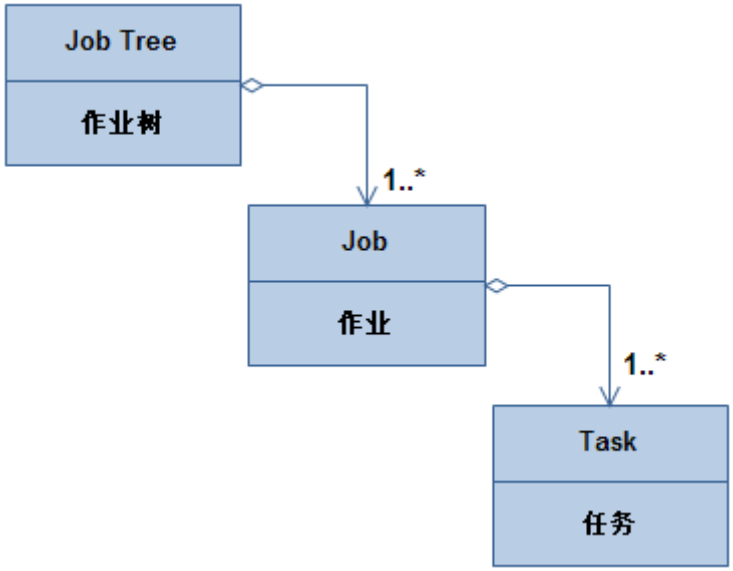
√：是，✖：否，–：未定的

三者间的关系如下图所示：

从上图可以看出，Session 由 Service 创建和管理，而 Object 与 Service/Session 无关。

1.4. 改进型 MapReduce

改进型的 MapReduce 并不是一个 MapReduce，它不存在数据倾斜问题，以及具备类似于进程调度器一样的调度能力。它的处理流程中，除了 Map 和 Reduce 外，还有个 Balance 过程。对同一作业 Job 内的任务 Task 划分，也不止 MapTask 和 ReduceTask，而是更多种类，每一个 Task 可以认为是一个进程调度单位。



改进型 MapReduce 的核心工作是将待执行的工作动态的切分万若干多的 Task，然后调度集群内各节点完成所有的 Task，其中有些 Task 可以并行执行，有些只能顺序执行，不同作业树中的作业总是可以并行执行，同一作业树中的作业有些可以并行，但也有些不可以并行。

2. 代码风格

2.1. 类

类名采用骆驼命名法，并以大写字母 C 打头，如：CStringUtil。

2.2. 接口

类名采用骆驼命名法，并以大写字母 I 打头，如：ILogger，并且回调接口使用宏 CALLBACK_INTERFACE 标识。

2.3. 结构体

采用 posix 风格，以_t 结尾，如：message_t。

2.4. 变量/类变量

非类成员变量采用小写+下划线风格，如：queue_size；类成员变量均以单个下划线打头，其余和非类成员变量相同，如：_queue_size。

2.5. 文件名

文件名采用小写+下划线风格，如：array_queue.h。

2.6. 编译工具

使用 automake 作为编译工具，但每次从 SVN 上下载代码后，都应当执行一次 first_once.sh。

3. 为何只支持 Linux

Mooon 中的各基础类库、公共组件和应用平台等，都只支持 Linux 系统。这样选择的原因是：一 Linux 系统上的应用范围已经足够广泛，有着十分强的生命力和广阔的需求；二是这样可以精简代码和简化结构；三是可以除低难度和减少工作量。总之，专注于 Linux，能够将 Linux 上的做好就已经非常足够了。

4. 如何编译基础类库？

编译测试代码之前，需要先编译好 common 库，common 库包含两大部分：基础类库和插件类库，它们的编译是自动进行的，而且会自动探测 MySQL 的安装目录。

要求 MySQL 安装在/usr/local 或用户主目录下，或者由环境变量 MYSQL_HOME 指定安装路径，而且 MySQL 的目录结构应当如下：

MySQL 安装目录

|---------include

|----------lib

在 include 目录下要求有 mysql.h 头文件，在 lib 目录下要求有 libmysqlclient_r.so 库文件。

当然，如果您不使用 mooon db wrapper，可以不用安装 MySQL，因为编译脚本会自动检测，如果没有安装，不会执行编译。

言归正传，先从 SVN 上取最新的 mooon common-library 代码，上传到 Linux 后，进入 src 目录，然后依次：

1) 执行 **first_once.sh**，该文件默认无执行权限，所以可 sh first_once.sh 方式执行（注：first_once.sh 只有在从 SVN 上取出代码时执行一次，这也是取 first_once 的意思）
2) 接下来，完全和普通的 **automake** 操作步骤一样，就不多说了

编译好 mooon common-library 后，就可以进入 test 目录，编译测试代码了。在测试代码目录中，Makefile 是用来编译测试代码的，所以只需要执行 make 即可，而且 run.sh 是用来运行测试代码的，每一个 CPP 文件都会被编译成一个可执行的文件，所以必须保证目录下所有的 CPP 文件都实现了 main 函数。

常用的编译步骤如下：

1) sh first_once.sh
2) ./configure --prefix=指定一个安装目录
3) make
4) make install

其中，上面的第一步 sh first_once.sh，仅当从 SVN 上更新代码时，才需要执行一次，其它情况下不需要执行。使用 sh first_once.sh，而不是./first_once.sh，是因为从 SVN 上取下的 first_once.sh 默认没有可执行权限。

5. util 库

5.1. 库文件

libutil.so

该库中的所有类和函数都是非线程安全的。

5.2. 名字空间

util，如：using namespace util;

建议：

尽量少使用 using namespace util，特别是在头文件中，即使是 CPP 文件中也应当少使用，因为这会使用名字空间失去本来的作用-防止名字冲突，最多在函数内部使用 using namespace util;。

虽然这样做会增加一些麻烦，但可以增加代码的纯洁性，预防潜在问题和增强代码可读性。也不建议以 using util::CStringUtil 方式使用，原因和 using namespace util 相同。

5.3. 依赖库

不依赖其它库。

5.4. delete_helper 和 free_helper

5.4.1. 主要功能

delete_helper 是一个模板类，用于辅助 new 或 malloc 等后的对象自动删除和内存自动释放。free_helper 模板类的作用和 delete_helper 相同，只不过它是针对 malloc 和 realloc 分配的内存。

5.4.2. 成员函数

5.4.2.1. delete_helper

```
/***
  * delete 帮助类，用来自动释放 new 分配的内存
  */
template <class ObjectType>
class delete_helper
{
public:
    /***
      * 构造一个 delete_helper 对象
      * @obj: 需要自动删除的对象指针
      * @is_array: 是否为 new 出来的数组
      */
    delete_helper(ObjectType*& obj, bool is_array=false);

    /** 析构中，用于自动调用 delete 或 delete []，调用后，指针将被置为 NULL */
    ~delete_helper();
};
```

5.4.2.2. free_helper

```
/***
  * malloc 帮助类，用来自动释放 new 分配的内存
  */
template <typename ObjectType>
class free_helper
{
public:
    /***
      * 构造一个 free_helper 对象
      * @obj: 需要自动删除的对象指针
      */
    free_helper(ObjectType*& obj);

    /** 析构中，用于自动调用 free，调用后，指针将被置为 NULL */
    ~free_helper();
};
```

5.4.3. 示例代码 1

```
自动删除 new 出来的单个对象
{ // 这里作用域开始
    int* p = new int(9); // new 一个 int 对象
    util::delete_helper<int> dh(p); // 使用 delete 帮助类
    。。。。。。
} // 这里作用域结束，在这儿将会自动调用 delete p;操作
```

5.4.4. 示例代码 2

```
自动删除 new 出来的对象数组
{ // 这里作用域开始
    int* p = new int[10]; // new 一个 int 对象数组
    util::delete_helper<int> dh(p, true); // 使用 delete 帮助类
    。。。。。。
} // 这里作用域结束，在这儿将会自动调用 delete []p;操作
```

5.4.5. 示例代码 3

自动释放 malloc 出来的内存

```
{ // 这里作用域开始
    int* p = (int*)malloc(9); // 分配内存
    util::free_helper<int> dh(p); // 使用 free 帮助类
    。。。。。。
} // 这里作用域结束，在这儿将会自动调用 free(p);操作
```

5.5. va_list_helper

5.5.1. 主要功能

用于调用 va_start 后，自动调用 va_end。

5.5.2. 成员函数

```
/***
  * va_list 帮助类，用于自动调用 va_end
  */
class va_list_helper
{
public:
    va_list_helper(va_list& args);

    /** 析构函数，自动调用 va_end */
    ~va_list_helper();
};
```

5.5.3. 示例代码

```
#include <util/util_config.h>

size_t update(const char* format, ...)
{
    va_list args;
    va_start(args, format);
    util::va_list_helper vlh(args); // 在析构时将自动调用 va_end(args)

    size_t affected_rows = _mysql_connection.update(format, args);
    return affected_rows;
```

```
}   // 这里将自动调用 va_end(args)
```

5.6. CountHelper

5.6.1. 头文件

```
#include <util/util_config.h>
```

5.6.2. 主要功能

在构造函数中，对计数器进行增一操作；在析构函数中，对计数器进行减一操作。

5.6.3. 成员函数

```
/***
  * 计数帮助类，用于自动对计数器进行增一和减一操作
  */
template <typename DataType>
class CountHelper
{
public:
    /** 构造函数，对计数器 m 进行增一操作 */
    CountHelper(DataType& m);

    /** 析构函数，对计数器 m 进行减一操作 */
    ~CountHelper();
};
```

5.6.4. 示例代码

```
    // 下列代码来源于 CEventQueue 的实现
    /***
      * 弹出队首元素
      * @elem: 存储被弹出的队首元素
      * @return: 如果成功从队列弹出数据则返回 true，否则返回 false
      * @exception: 可能抛出 CSyscallExceptoin 异常，其它异常和 RawQueue 有关
      */
    bool pop_front(DataType& elem)
    {
        CLockHelper<CLock> lock(_lock);
        while (_raw_queue.is_empty())
```

```
        {
            // 如果不等待，则立即返回
            if (0 == _pop_milliseconds) return false;
            // 使用助手类管理计数，因为 timed_wait 可能抛异常
            // 这句相当于++_pop_waiter_number;
            util::CountHelper<volatile int> ch(_pop_waiter_number);

            // 超时则立即返回
            if (!_event.timed_wait(_lock, _pop_milliseconds)) return false;
        } // ch 在这里被析构，析构中会调用--_pop_waiter_number;

        elem = _raw_queue.pop_front();
        if (_push_waiter_number > 0) _event.signal();
        return true;
    }
```

5.7. CIntegerUtil

5.7.1. 头文件

```
#include <util/integer_util.h>
```

5.7.2. 主要功能

提供整数类型的操作函数：

1) 判断是一个数字是否为质数
2) 判断一个数字是否可为另一数字类型

5.7.3. 成员函数

```
/***
  * 整数数字操作工具类
  */
class CIntegerUtil
{
public:
    template <typename DataType>
    static bool is_prime_number(DataType x);

    /** 判断一个数字是否可为 int16_t 数字 */
    static bool is_int16(int32_t num);
```

```
    /** 判断一个数字是否可为 uint16_t 数字 */
    static bool is_uint16(int32_t num);
    static bool is_uint16(uint32_t num);

    /** 判断一个数字是否可为 int32_t 数字 */
    static bool is_int32(int64_t num);

    /** 判断一个数字是否可为 uint32_t 数字 */
    static bool is_uint32(int64_t num);
    static bool is_uint32(uint64_t num);
};
```

5.8. CStringUtil 类

5.8.1. 头文件

```
#include <util/string_util.h>
```

5.8.2. 主要功能

提供以下功能：

1) 字符串大小转换
2) 字符串去空格
3) 删除字符串尾部指定字符或字符串
4) 字符串转换成各种有符号和无符号整数类型，可指定需要转换的字符个数，以及是否自动跳过打头的 0 字符
5) 各种有符号和无符号整数类型转换成字符串
6) 返回实际复制长度的 snprintf 和 vsnprintf 函数
7) 字符串哈希函数
8) 跳过字符串空格函数

5.8.3. 成员函数

5.8.3.1. remove_last

```
    /** 删除字符串尾部从指定字符开始的内容
     * @source: 需要处理的字符串
     * @c: 分隔字符
     * @example: 如果 str 为“/usr/local/test/bin/”，而 c 为“/”，
     *           则处理后 str 变成“/usr/local/test/bin”
     */
```

```
static void remove_last(std::string& source, char c);

/** 删除字符串尾部从指定字符串开始的内容
  * @source: 需要处理的字符串
  * @sep: 分隔字符串
  * @example: 如果 str 为“/usr/local/test/bin/tt”，而 sep 为“/bin/”，
  *           则处理后 str 变成“/usr/local/test
  */
static void remove_last(std::string& source, const std::string& sep);
```

5.8.3.2. to_upper

```
/** 将字符串中的所有小写字符转换成大写 */
static void to_upper(char* source);
static void to_upper(std::string& source);
```

5.8.3.3. to_lower

```
/** 将字符串中的所有大写字符转换成小写 */
static void to_lower(char* source);
static void to_lower(std::string& source);
```

5.8.3.4. is_space

```
/** 判断指定字符是否为空格或 TAB 符(\t)或回车符(\r)或换行符(\n) */
static bool is_space(char c);
```

5.8.3.5. trim

```
/** 删除字符串首尾空格或 TAB 符(\t)或回车符(\r)或换行符(\n) */
static void trim(char* source);
static void trim(std::string& source);
```

5.8.3.6. trim_left

```
/** 删除字符串首部空格或 TAB 符(\t)或回车符(\r)或换行符(\n) */
static void trim_left(char* source);
static void trim_left(std::string& source);
```

5.8.3.7. trim_right

```
/** 删除字符串尾部空格或 TAB 符(\t)或回车符(\r)或换行符(\n) */
static void trim_right(char* source);
static void trim_right(std::string& source);
```

5.8.3.8. 字符串转整数

仅以 string2int32 为例，其它同类函数的参数和返回值含义相同。

```
/***
  * @source: 待转换为整数的字符串
  * @result: 用来存储转换后的整数
  * @converted_length: 对多少个字符进行转换，如果为 0 表示对整个字符串进行转换
  * @ignored_zero: 是否忽略打头的一个或多个字符串 0
  * @return: 如果转换成功，则返回 true，否则返回 false
  */
static bool string2int32(const char* source
                         , int32_t& result
                         , uint8_t converted_length=0
                         , bool ignored_zero=false);
```

同类函数：

string2int8，string2int16，string2int32，string2int64，
string2uint8，string2uint16，string2uint32，string2uint64

转换失败的几种原因：

1) 在需要转换的长度内，字符串中含有非数字，如果为参数 converted_length 为 0，则表示整个字符串
2) 转换后的整数值超出目标类型的取值范围，如超过了 32 位整数的最大值
3) 当 ignored_zero 参数值为 false 时，字符串以 0 打头
4) 参数 source 为空字符串，或为 NULL 指针

5.8.4. 示例代码

```
int32_t year;
const char* str = "2010";

if (util::CStringUtil::string2int32(str, year))
    printf("%s ==> %d\n", str, year);
else
    printf("Error converting %s to int32_t\n", str);
```

这个转换会**成功**，将输出：2010 ==> 2010。

5.8.5. 示例代码 2

```
int32_t year;
const char* str = "2010year";

if (util::CStringUtil::string2int32(str, year))
    printf("%s ==> %d\n", str, year);
else
    printf("Error converting %s to int32_t\n", str);
```

这个转换会**失败**，将输出：Error converting 2010year to int32_t。

5.8.6. 示例代码 3

```
int32_t year;
const char* str = "2010year";

if (util::CStringUtil::string2int32(str, year, 4))
    printf("%s ==> %d\n", str, year);
else
    printf("Error converting %s to int32_t\n", str);
```

这个转换会**成功**，将输出：2010year ==> 2010，因为这里第 3 个参数 **converted_length** 的值为 4，即只转换前 4 个字符，忽略后面的字符。

5.8.7. 示例代码 4

```
int32_t year;
const char* str = "002010";

if (util::CStringUtil::string2int32(str, year))
    printf("%s ==> %d\n", str, year);
else
    printf("Error converting %s to int32_t\n", str);
```

这个转换会**失败**，将输出：Error converting 02010 to int32_t。

5.8.7.1. 示例代码 5

```
int32_t year;
const char* str = "002010";

if (util::CStringUtil::string2int32(str, year, 0, true))
    printf("%s ==> %d\n", str, year);
else
    printf("Error converting %s to int32_t\n", str);
```

这个转换会**成功**，将输出 2010 ==> 2010，因为这里第 4 个参数 **ignored_zero** 的值为 true，即允许字符串以 0 打头。

5.8.7.2. 示例代码 6

```
char str8[] = " def";
printf("abc%s\n", str8);
CStringUtil::trim_left(str8);
printf("abc%s\n", str8);
```

上面这段代码将输出如下两行内容：

```
abc def
abcdef
```

注意第一行输出中 abc 和 def 间有一个空格。

5.8.7.3. 示例代码 7

```
char str9[] = "abc    ";
printf("%sdef\n", str9);
CStringUtil::trim_right(str9);
printf("%sdef\n", str9);
```

上面这段代码将输出如下两行内容：

```
abc    def
abcdef
```

注意第一行输出中 abc 和 def 间有两个空格。

5.8.7.4. 示例代码 8

```
char str10[] = " 456 ";
printf("123%s789\n", str10);
CStringUtil::trim(str10);
printf("123%s789\n", str10);
```

上面这段代码将输出如下两行内容：

123 456 789
123456789
注意第一行输出中 123 和 456 间有一个空格，而且 456 和 789 间也有一个空格。

5.9. CTokenList

5.9.1. 头文件

#include "util/token_list.h"

5.9.2. 主要功能

以指定的字符串作为分隔符，将整个字符中各 Token 解析到一个 Token 链表中。

5.9.3. 成员函数

CTokenList 类只有一个静态函数 parse。

```
/***
   * 以指定的字符串作为分隔符，将整个字符中各 Token 解析到一个 Token 链表中
   * @token_list: 存储 Token 的链表
   * @source: 被解析的字符串
   * @sep: Token 分隔符
   */
static void parse(TTokenList& token_list, const std::string& source, const std::string& sep);
```

5.9.4. 示例代码

```
std::string str = "ABC||DEFG||HIJK";
util::CTokenList::TTokenList token_list;

util::CTokenList::parse(token_list, str, "||");
util::CTokenList::TTokenList::iterator iter = token_list.begin();
for (;iter != token_list.end(); ++iter)
{
    printf("%s\n", iter->c_str());
}
```

这个例子将输出 3 行如下内容：
ABC
DEFG

HIJK

5.10. CArrayQueue

5.10.1. 头文件

#include "util/array_queue.h"

5.10.2. 主要功能

以数组方式提供容量固定的队列功能，结构简单，而且在多线程环境进行锁优化。

5.10.3. 成员函数

```
/** 队列中的元素数据类型 */
typedef DataType _DataType;

/***
  * 构造一个数组队列
  * @queue_max: 需要构造的队列大小
  */
CArrayQueue(uint32_t queue_max)

/** 判断队列是否已满 */
bool is_full() const;

/** 判断队列是否为空 */
bool is_empty() const;

/** 返回队首元素 */
DataType front() const;

/***
  * 弹出队首元素
  * 注意: 调用 pop 之前应当先使用 is_empty 判断一下
  * @return: 返回队首元素
  */
DataType pop_front();

/***
  * 往队尾插入一个元素
```

```
     * 注意: 调用 pop 之前应当先使用 is_full 判断一下
     */
    void push_back(DataType elem);

    /** 得到队列中存储的元素个数 */
    uint32_t size() const;
```

5.10.4. 示例代码

```
#include <util/array_queue.h> // 包含头文件

// 构造一个 int 类型的数组队列，总共可容纳 10 个元素
util::CArrayQueue<int> int_queue(10);

int_queue.push_back(1);
int_queue.push_back(3);
int_queue.push_back(5);
int_queue.push_back(7);

while (!int_queue.is_empty())
{
    int x = int_queue.pop_front();
    printf("%d\n", x);
}
```

运行上面的代码，将输出如下内容：

```
1
3
5
7
```

5.11. CListable 和 CListQueue

5.11.1. 头文件

```
#include <util/listable.h>
#include <util/list_queue.h>
```

5.11.2. 主要功能

CListable：可链表对象的基类，用来将某类对象串起成一个链表。

CListQueue：存储可链表对象的模板基类，队列中的每一个对象类型都应当为 CListable 子类型，特点是可随机删除队列中的某个元素，并且 0 查找，因此删除操作高效率。

在 Mooon 中，这两个类经常用于超时管理功能中，依靠它们可以避免对整个超时队列的遍历，而仅仅只需要遍历确实已经超时的对象。

5.11.2.1. 示例代码

请参见 net 库中的 CTimeoutManager 一节点。

5.11.3. 成员函数

5.11.3.1. CListable

```
/***
  * 可链表对象的基类
  */
class CListable
{
public:
    /** 得到下一个可链表对象 */
    CListable* get_next() const;

    /** 得到前一个可链表对象 */
    CListable* get_prev() const;

    /** 关联下一个可链表对象 */
    void set_next(CListable* next);

    /** 关联前一个可链表对象 */
    void set_prev(CListable* prev);
};
```

5.11.3.2. CListQueue

```
/***
  * 可链表对象模板队列
  */
template <class ListableClass>
class CListQueue
{
```

```
public:
    /** 构造一个可链表对象模板队列 */
    CListQueue();

    /** 得到指向队首对象的指针 */
    ListableClass* front() const;

    /** 在队尾添加一个可链表对象 */
    void push(ListableClass* listable);

    /**
     * 将一个可链表对象从队列中删除
     * 删除操作是高效的，因为 0 查找，只需要解除链接关系即可
     */
    void remove(ListableClass* listable);
};
```

5.12. CHistogramArray

5.12.1. 头文件

```
#include <util/histogram_array>
```

5.12.2. 主要功能

提供直方图形状的数组，即一维个数确定，但其它维的个数不确定。

5.12.3. 注意事项

CHistogramArray 的性能较低，不适合对数据有频繁的增减操作。

5.12.4. 成员函数

```
/***
 * 直方图数组
 */
template <typename DataType>
class CHistogramArray
{
public:
```

```
    /** 数组存储的元素类型 */
    typedef DataType _DataType;

    /***
      * 构造一个直方图数组
      * @array_size: 直方图数组大小
      */
    CHistogramArray(uint32_t array_size);

    /** 析构直方图数组 */
    ~CHistogramArray();

    /***
      * 插入元素到指定的直方图中
      * @position: 直方图在数组中的位置
      * @elem: 需要插入的元素
      * @unique: 是否做去重检查，如果是，则不重复插入相同的元素
      * @return: 插入成功返回true，否则返回false
      */
    bool insert(uint32_t position, DataType elem, bool unique=true);

    /***
      * 从直方图中删除一个元素
      * @position: 直方图在数组中的位置
      * @elem: 需要删除的元素
      * @return: 如果元素在直方图中，则返回true，否则返回false
      */
    bool remove(uint32_t position, DataType elem);

    /** 得到可容纳的直方图个数 */
    uint32_t get_capacity() const;

    /** 检测一个直方图是否存在 */
    bool histogram_exist(uint32_t position) const;

    /** 得到直方图大小 */
    uint32_t get_histogram_size(uint32_t position) const;

    /** 得到直方图 */
    DataType* get_histogram(uint32_t position) const;
};
```

5.12.5. 示例代码

```
int main()
{
    CHistogramArray<int> hist(10);
    hist.insert(1,1);
    hist.insert(1,3);
    hist.insert(5,5);
    hist.insert(1,5);

    for (uint32_t i=0; i<hist.get_size(); ++i)
    {
        int* histogram = hist.get_histogram(i);
        if (NULL == histogram)
        {
            printf("histogram %d is null\n", i);
        }
        else
        {
            uint32_t histogram_size = hist.get_histogram_size(i);
            for (uint32_t j=0; j<histogram_size; ++j)
                printf("[%d][%d] --> %d\n", i, j, histogram[j]);
        }
    }
    return 0;
}
```

6. sys 库

6.1. 库文件

libsys.so

6.2. 名字空间

sys，如：using namespace sys;

6.3. 依赖库

依赖于 libutil.so 库。

6.4. CSyscallException

6.4.1. 头文件

#include <sys/syscall_exception.h>

6.4.2. 主要功能

系统调用出错，不采用传统的返回值和 errno 结合方式，而是以异常形式返回给调用者，这样做的目的是为了写出更为简洁的代码。

6.4.3. 成员函数

```
/** 系统调用出错异常，多数系统调用出错时，均以此异常型反馈给调用者 */
class CSyscallException
{
public:
    /** 构造系统调用异常
     * @errcode: 系统调用出错代码
     * @filename: 出错所发生的文件名
     * @linenumber: 出错发生的行号
     * @tips: 额外的增强信息，用以进一步区分是什么错误
     */
    CSyscallException(int errcode, const char* filename, int linenumber, const char* tips=NULL);

    /** 得到调用出错发生的文件名 */
    const char* get_filename() const;

    /** 得到调用出错时发生的文件行号 */
    int get_linenumber() const;

    /** 得到调用出错时的系统出错码，在 Linux 上为 errno 值 */
    int get_errcode() const;

    /** 得到调用出错时的提示信息，提示信息用于辅助明确问题，为可选内容
     * 如果返回非 NULL，则表示有提示信息，否则表示无提示信息
     */
    const char* get_tips() const;
};
```

6.5. CRefCountable

6.5.1. 头文件

#include <sys/ref_countable.h>

6.5.2. 主要功能

作为所有需要引用计数功能的基类，提供增减引用计数和自删除功能。

6.5.3. 成员函数

6.5.3.1. CRefCountable

```
/***
  * 线程安全的引用计数基类
  * 不应当直接使用此类，而应当总是从它继承
  */
class CRefCountable
{
public:
    /** 虚拟析构函数是必须的，否则无法删除子类对象 */
    virtual ~CRefCountable();

    /** 得到引用计数值 */
    int get_refcount() const;

    /** 对引用计数值增一 */
    void inc_refcount();

    /***
      * 对引用计数值减一
      * 如果引用计数值减一后，引用计数值变成 0，则对象自删除
      * @return: 如果对象自删除，则返回 true，否则返回 false
      */
    bool dec_refcount();
};
```

6.5.3.2. CRefCountHelper

```
/***
  * 引用计数帮助类，用于自动减引用计数
  */
template <class RefCountClass>
class CRefCountHelper
{
public:
    /***
      * 对引用计数增一
      * 析构时会将 ref_countable 指向 NULL
      */
    CRefCountHelper(CRefCountable*& ref_countable);

    /** 析构函数，自动减引用计数 */
    ~CRefCountHelper();
};
```

6.5.4. 示例代码

```
#include "sys/ref_countable.h"

// 使用引用计数帮助类，自动进行引用计数增一和减一
sys::CRefCountHelper<sys::IDBConnection> rch(general_connection);
```

6.6. close_helper

6.6.1. 头文件

```
#include <sys/close_helper.h>
```

6.6.2. 主要功能

用于帮助自动调用 close 函数，用于自动关闭一个对象、或文件句柄 fd 或文件描述符指针 FILE*。

6.6.3. 成员函数

6.6.3.1. close_helper

```
/***
  * 类类型 close 助手函数，要求该类有公有的 close 方法
  */
template <class ClassType>
class close_helper
{
public:
    /***
      * 构造一个 close_helper 对象
      * @obj: 需要 close_helper 自动调用其公有 close 方法的对象(非指针)
      */
    close_helper(ClassType& obj);

    /** 析构函数，用于自动调用对象的 close 方法 */
    ~close_helper();
};
```

6.6.3.2. close_helper<int>

```
/***
  * 针对整数类型文件句柄
  */
template <>
class close_helper<int>
{
public:
    close_helper<int>(int& fd);
    /** 析构函数，自动调用::close */
    ~close_helper<int>();
};
```

6.6.3.3. close_helper<FILE*>

```
/***
  * 针对标准 I/O
  */
template <>
```

```
class close_helper<FILE*>
{
public:
    close_helper<FILE*>(FILE*& fp);
    /** 析构函数，自动调用 fclose */
    ~close_helper<FILE*>();
};
```

6.6.4. 示例代码 1：

```
#include <sys/close_helper.h> // 包含头文件

class X
{
public:
    void close() {} // X 类有一个公有的 close 方法
};

{ // 作用域开始
    X x;
    sys::close_helper<X> ch(x); // 使用 close 帮助类
    。。。。。。
} // 作用域结束，在这里 x 的 close 方法会被自动调用
```

6.6.5. 示例代码 2：

```
{   // 作用域开始
    int fd = open("x.log", O_RDONLY); // 打开文件 x.log
    sys::close_helper<int> ch(fd); // 使用 close 帮助类
    。。。。。。
} // 作用域结束，在这里会自动调用 close(fd);
```

6.6.6. 示例代码 3：

```
{   // 作用域开始
    FILE* fp = fopen("x.log", "r"); // 打开文件 x.log
    sys::close_helper<FILE*> ch(fp); // 使用 close 帮助类
    。。。。。。
} // 作用域结束，在这里会自动调用 fclose(fp);
```

6.7. CThread

6.7.1. 头文件

#include <sys/thread.h>

6.7.2. 主要功能

提供对 Linux 线程的包装，使得使用线程更为简单。

6.7.3. 成员函数

```
/**
 * 线程抽象基类，支持引用计数
 */
class CThread: public CRefCountable
{
private:
    /***
     * 线程执行体函数，子类必须实现该函数，
     * 此函数内的代码半在一个独立的线程中执行
     */
    virtual void run() = 0;

    /** 在启动线程之前被调用的回调函数，如果返回 false，则会导致 start 调用也返回 false。
     * 可以重写该函数，将线程启动之前的逻辑放在这里。
     */
    virtual bool before_start() { return true; }

public:
    /** 得到当前线程号 */
    static uint32_t get_current_thread_id();

public:
    CThread();
    virtual ~CThread();

    /** 将_stop 成员设置为 true，线程可以根据_stop 状态来决定是否退出线程
     * @wait_stop: 是否等待线程结束，只有当线程是可 Join 时才有效
```

```
    * @exception: 当 wait_stop 为 true 时，抛出和 join 相同的异常，否则不抛异常
    */
virtual void stop(bool wait_stop=true);

/** 启动线程
    * @detach: 是否以可分离模式启动线程
    * @exception: 如果失败，则抛出 CSyscallException 异常
    *                  如果是因为 before_start 返回 false，则出错码为 0
    */
void start(bool detach=false);

/** 设置线程栈大小。应当在 start 之前调用，否则设置无效，如放在 before_start 当中。
    * @stack_size: 栈大小字节数
    * @exception: 不抛出异常
    */
void set_stack_size(size_t stack_size) { _stack_size = stack_size; }

/** 得到线程栈大小字节数
    * @exception: 如果失败，则抛出 CSyscallException 异常
    */
size_t get_stack_size() const;

/** 得到本线程号 */
uint32_t get_thread_id() const { return _thread; }

/** 等待线程返回
    * @exception: 如果失败，则抛出 CSyscallException 异常
    */
void join();

/** 将线程设置为可分离的，
    * 请注意一旦将线程设置为可分离的，则不能再转换为可 join。
    * @exception: 如果失败，则抛出 CSyscallException 异常
    */
void detach();

/** 返回线程是否可 join
    * @return: 如果线程可 join 则返回 true，否则返回 false
    * @exception: 如果失败，则抛出 CSyscallException 异常
    */
bool can_join() const;

/***
    * 如果线程正处于等待状态，则唤醒
```

```
        */
    void wakeup();

protected: // 仅供子类使用
    /***
      * 判断线程是否应当退出，默认返回_stop 的值
      */
    virtual bool is_stop() const;

    /***
      * 毫秒级 sleep，线程可以调用它进入睡眠状态，并且可以通过调用 do_wakeup 唤醒
      * 请注意只本线程可以调用此函数，其它线程调用无效
      */
    void do_millisleep(int milliseconds);
};
```

6.7.4. 示例代码

```
#include <sys/thread.h>
#include <sys/sys_util.h>

// 所有非线程池线程都应当是 CThread 的子类
// Exam 线程 CDemoThread 运行后，每隔 1 秒往标准输出打印一行“continue...”
class CExamThread: public sys::CThread
{
private:
    virtual void run();
    virtual bool before_start();
};

bool CExamThread::before_start()
{
    // start 的返回值和 before_start 的相同，一般用于线程启动前的初始化
    return true;
}

// 线程每秒钟往标准输出打印一次 continue...
void CExamThread::run()
{
    // stop 将使得 is_stop 返回 false
    while (!is_stop())
    {
        // 睡眠 1 秒钟
        // do_millisleep 是由 CThread 提供给子类睡眠使用的，
```

```
            // 可通过调用 wakeup 将睡眠中断
            do_millisleep(1000);
            printf("continue ...\n");
        }
    }

    int main()
    {
        // 创建并启动线程
        CExamThread* thread = new CExamThread;
        thread->inc_refcount(); .// new 之后，调用 inc_refcount 是安全的使用方法
        try
        {
            if (!thread->start())
            {
                // 只有当 before_start 返回 false，thread->start 才会返回 false，否则总是返回 true
                thread->dec_refcount();
                exit(1);
            }
        }
        catch (sys::CSyscallException& ex)
        {
            printf("Start thread error: %s\n"
                    , sys::CSysUtil::get_error_message(ex.get_errcode()).c_str());
            thread->dec_refcount();
            exit(1);
        }

        // 主线程睡眠 10 秒钟
        sys::CSysUtil::millisleep(10000);

        thread->stop(); // 停止并待线程退出
        thread->dec_refcount(); // 记得增加了引用计数，就需要在使用完后，相应的减引用计数

        return 0;
    }
```

运行上面的代码，在输出如下 10 行内容后，进程退出：

```
continue ...
continue ...
continue ...
continue ...
continue ...
continue ...
continue ...
```

continue ...
continue ...
continue ...

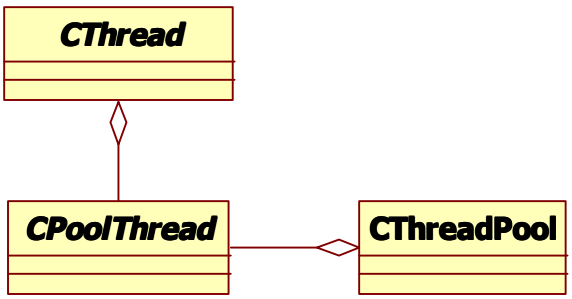
6.8. CThreadPool 和和 CPoolThread

6.8.1. 头文件

线程池头文件：#include <sys/thread_pool.h>
池线程头文件：#include <sys/pool_thread.h>

6.8.2. 主要功能

提供通用的线程池功能。请注意 CPoolThread 并不是 CThread 的子类，它们的关系如下图所示：



也即 CThread 只是 CPoolThread 的一个子对象，而非父类，CThreadPool 创建的是 CPoolThread，而非 CThread。线程池中的线程个数是在创建线程池时确定的，不能动态修改。

6.8.3. 池线程和线程

池线程和线程有何区别？主要以下几点：
1.池线程是自动循环的，如：

```
// 线程写法
void CMyThread::run()
{
    for (;;)
    {
        do_something();
    }
}
```

```
// 池线程写法
void CMyThread::run()
{
    do_something();
}
```

2.池线程的生命周期是由线程池 CThreadPool 管理的
3.线程池 create 成功后，必须对池中每个池线程调用它的 wakeup，池线程才会开始执行
4.池线程不支持 detach，它总是由线程池回收，所以必须为可 join 的

6.8.4. 成员函数

6.8.4.1. CPoolThread

```
// 用于线程池的线程抽象基类
class CPoolThread: public CRefCountable
{
protected: // protected 类型析构和构造，目的是禁止直接创建 CPoolThread 的实例
    CPoolThread();
    virtual ~CPoolThread();
    /***
      * 毫秒级 sleep，线程可以调用它进入睡眠状态，并且可以通过调用 wakeup 唤醒，
      * 请注意只本线程可以调用此函数，其它线程调用无效
      */
    void do_millisleep(int milliseconds);

public:
    /***
      * 唤醒池线程，池线程启动后，都会进入睡眠状态，
      * 直接调用 wakeup 将它唤醒
      */
    void wakeup();

    /***
      * 得到池线程在线程池中的序号，序号从 0 开始，
      * 且连续，但总是小于线程个数值。
      */
    uint16_t get_index() const;

    /** 设置线程栈大小，应当在 before_start 中设置。
      * @stack_size: 栈大小字节数
      * @exception: 不抛出异常
      */
    void set_stack_size(size_t stack_size);
```

```
    /** 得到线程栈大小字节数
      * @exception: 如果失败，则抛出 CSyscallException 异常
      */
    size_t get_stack_size() const;

    /** 得到本线程号 */
    uint32_t get_thread_id() const;
};
```

6.8.4.2. CThreadPool

```
/***
  * 线程池模板类，模板参数为线程类
  */
template <class ThreadClass>
class CThreadPool
{
public:
    /** 构造一个线程池 */
    CThreadPool();

    /** 创建线程池，并启动线程池中的所有线程，
      * 池线程创建成功后，并不会立即进行运行状态，而是处于等待状态，
      * 所以需要唤醒它们，用法请参见后面的示例
      * @thread_count: 线程池中的线程个数
      * @exception: 可抛出 CSyscallException 异常
*                  如果是因为 CPoolThread::before_start 返回 false，则出错码为 0
      */
    bool create(uint16_t thread_count);

    /** 销毁线程池，这里会等待所有线程退出，然后删除线程 */
    void destroy();

    /***
      * 激活线程池，将池中的所有线程唤醒，
      * 也可以单独调用各池线程的 wakeup 将它们唤醒
      */
    void activate ()

    /** 得到线程池中的线程个数 */
    uint16_t get_thread_count() const;
```

```
    /** 得到线程池中的线程数组 */
    ThreadClass** get_thread_array() const;

    /***
      * 得到指向下个线程的指针，从第一个开始循环遍历，无终结点
      *，即达到最后一个时，又指向第一个，
      * 主要应用于采用轮询方式将一堆任务分配均衡分配给池线程。
      */
    ThreadClass* get_next_thread();
};
```

6.8.5. 示例代码

```
#include <assert.h>
#include <sys/sys_util.h>
#include <sys/pool_thread.h>
#include <sys/thread_pool.h>

// 测试线程，请注意是继承池线程 CPoolThread，而不是线程 CThread，
// 另外 CPoolThread 并不是 CThread 的子类
class CTestThread: public sys::CPoolThread
{
public:
    CTestThread()
        :_number_printed(0)
    {
    }

private:
    virtual void run()
    {
        // 在这个函数里实现需要池线程干的事
        if (_number_printed++ < 3)
        {
            // 只打印三次，以便观察效率
            printf("thread %u say hello.\n", get_thread_id());
        }

        // do_millisleep 是由 CPoolThread 提供给子类睡眠用的，
        // 可以通过调用 wakeup 中断睡眠
        do_millisleep(1000);
    }

    virtual bool before_start()
```

```
    {
        // 在这个函数里完成线程启动前的准备工作
        return true;
    }

private:
    int _number_printed; // 打印次数
};

int main()
{
    sys::CThreadPool<CTestThread> thread_pool; // 这里定义线程池实例

    try
    {
        // create 可能抛出异常，所以需要捕获
        if (!thread_pool.create(10)) // 创建 10 个线程的线程池
        {
            // 在本例中，永远也不会走到这里，因为 before_start 总是返回 true，
            // 只有当 before_start 返回 false，才会走到这里
            assert(false);
            exit(0);
        }

        // 池线程创建成功后，并不会立即进行运行状态，而是处于等待状态，
        // 所以需要唤醒它们，方法如下：
        uint16_t thread_count = thread_pool.get_thread_count();
        CTestThread** test_thread_array = thread_pool.get_thread_array();
        for (uint16_t i=0; i<thread_count; ++i)
        {
            // 唤醒池线程，如果不调用 wakeup，则 CTestThread 一直会等待唤醒它
            test_thread_array[i]->wakeup();
        }

        // 让 CTestThread 有足够的时间完成任务
        sys::CSysUtil::millisleep(5000);
        // 等待所有线程退出，然后销毁线程池
        thread_pool.destroy();
    }
    catch (sys::CSyscallException& ex)
    {
        // 将异常信息打印出来，方便定位原因
        printf("Create thread pool exception: %s.\n"
            ,sys::CSysUtil::get_error_message(ex.get_errcode()).c_str());
```

```
    }

    return 0;
}
```

6.9. CLock 和 CRecLock

6.9.1. 头文件

```
#include <sys/lock.h>
```

6.9.2. 主要功能

用于线程间互斥，支持尝试性和超时方式获取独占锁。

6.9.3. 成员函数

6.9.3.1. CLock

```
/***
  * 互斥锁类
  * 对于非递归锁，同一线程在未释放上一次加的锁之前，
  * 不能连续两次去加同一把锁，但递归锁允许同一线程连续对同一把锁加多次
  */
class CLock
{
public:
    /***
      * 构造一个互斥锁
      * @recursive: 是否构造为递归锁
      * @exception: 出错抛出 CSyscallException 异常
      */
    CLock(bool recursive = false);
    ~CLock();

    /***
      * 加锁操作，如果不能获取到锁，则一直等待到获取到锁为止
      * @exception: 出错抛出 CSyscallException 异常
      */
    void lock();
```

```
    /***
      * 解锁操作
      * 请注意必须已经调用了 lock 加锁，才能调用 unlock 解锁
      * @exception: 出错抛出 CSyscallException 异常
      */
    void unlock();

    /***
      * 尝试性的去获取锁，如果得不到锁，则立即返回
      * @return: 如果获取到了锁，则返回 true，否则返回 false
      * @exception: 出错抛出 CSyscallException 异常
      */
    bool try_lock();

    /***
      * 以超时方式去获取锁，如果指定的毫秒时间内不能获取到锁，则一直等待直到超时
      * @return: 如果在指定的毫秒时间内获取到了锁，则返回 true，否则如果超时则返回
false
      * @exception: 出错抛出 CSyscallException 异常
      */
    bool timed_lock(uint32_t millisecond);
};
```

6.9.3.2. CRecLock

```
/***
  * 递归锁类
  * 对于非递归锁，同一线程在未释放上一次加的锁之前，
  * 不能连续两次去加同一把锁，但递归锁允许同一线程连续对同一把锁加多次
  */
class CRecLock: public CLock
{
public:
    /***
      * 构造一个递归锁
      * @exception: 出错抛出 CSyscallException 异常
      */
    CRecLock();
};
```

6.9.3.3. CLockHelper

```
/***
```

```
 * 锁帮助类，用于自动解锁
 */
template <class LockClass>
class CLockHelper
{
public:
    /***
     * 构造锁帮助类对象
     * @exception: 出错抛出 CSyscallException 异常
     */
    CLockHelper(LockClass& lock);

    /***
     * 析构函数，会自动调用 unlock 解锁
     * @exception: 析构函数不允许抛出任何异常
     */
    ~CLockHelper();
};
```

6.9.4. 示例代码

```
#include <sys/lock.h>

class X
{
public:
    void hello(); // 须线程安全的方法

private:
    sys::CLock _lock; // 锁成员
};

void X::hello()
{
    // 建议尽量使用帮助类 CLockHelper，而不是直接调用_lock.lock();
    sys::CLockHelper<sys::CLock> lock(_lock); // 加锁
    。。。
} // 这里会自动解锁
```

6.10. CEvent

6.10.1. 头文件

#include <sys/event.h>

6.10.2. 主要功能

1) 永久阻塞调用线程，直到被另一线程唤醒
2) 超时方式阻塞调用线程，直到被另一线程唤醒或超时
3) 唤醒单个因调用 wait 或 timed_wait 而被阻塞的线程
4) 广播唤醒所有因调用 wait 或 timed_wait 而被阻塞的线程

6.10.3. 成员函数

```
/***
  * 通知事件类
  * 常用于队列的实现，队列为空时则等待队列有数据，如果往队列 push 了数据
  * ，则唤醒等待线程去队列取数据
  */
class CEvent
{
public:
    /***
      * 构造通知事件实例
      * @exception: 出错抛出 CSyscallException 异常，通常可不捕获此异常
      */
    CEvent();
    ~CEvent();

    /***
      * 让调用者进入等待状态，直接到被唤醒
      * @exception: 出错抛出 CSyscallException 异常，通常可不捕获此异常
      */
    void wait(CLock& lock);

    /***
      * 让调用者进入等待状态，直接到被唤醒，或等待的时长超过指定的毫秒数
      * @return: 如果在指定的毫秒数之前被唤醒，则返回 true，否则返回 false
      * @exception: 出错抛出 CSyscallException 异常，通常可不捕获此异常
      */
```

```
    bool timed_wait(CLock& mutex, uint32_t millisecond);

    /***
      * 唤醒一个进入等待状态的线程，如果没有线程正处于等待状态，则唤醒动作忽略
      * 只有当 signal 调用发生在 wait 调用之后才有效
      * @exception: 出错抛出 CSyscallException 异常，通常可不捕获此异常
      */
    void signal();

    /***
      * 广播唤醒信号，将所有进入等待状态的线程全部唤醒，
      * 如果没有线程正处于等待状态，则唤醒动作忽略
      * 只有当 broadcast 调用发生在 wait 调用之后才有效
      * @exception: 出错抛出 CSyscallException 异常，通常可不捕获此异常
      */
    void broadcast();
};
```

6.10.4. 示例代码

```
// CEvent 常用于队列的实现中
#include <sys/lock.h>
#include <sys/event.h>

class CMyQueue
{
public:
    CMyQueue()
        :_waiters_number(0)
    {
    }

    void push(char* msg)
    {
        sys::CLockHelper<sys::CLock> lock_helper(_lock);
        // 往队尾插入消息
        。。。。。。
        if (_waiters_number > 0)
            _event.signal(); // 唤醒一个等待者
    }

    char* pop()
    {
        sys::CLockHelper<sys::CLock> lock_helper(_lock);
```

```
        // 如果队列为空，则等待队列中有数据
        while (is_empty())
        {
            ++_waiters_number;
            // 进入 wait 后，锁会被解除，但在 wait 返回后，锁又会被加上
            _event.wait(_lock);
            --_waiters_number;
        }

        。。。。。。
        // 从队列取出数据
    }

private:
    volatile int _waiters_number; // 等待者个数
    sys::CLock _lock;
    sys::CEvent _event;
};
```

6.11. CReadWriteLock

6.11.1. 头文件

```
#include <sys/read_write_lock.h>
```

6.11.2. 主要功能

提供读写锁的功能，支持尝试性和超时方式获取读写锁。可以同时加多个读锁，但同一时刻只允许加一把写锁。

6.11.3. 成员函数

6.11.3.1. CReadWriteLock

```
/***
  * 读写锁类
  * 请注意：谁加锁谁解锁原则，即一个线程加的锁，不能由另一线程来解锁
  * 而且加锁和解锁必须成对调用，否则会造成死锁
  */
class CReadWriteLock
{
```

```
public:
    CReadWriteLock();
    ~CReadWriteLock();

    /***
      * 释放读或写锁
      * @exception: 出错抛出 CSyscallException 异常
      */
    void unlock();

    /***
      * 获取读锁，如果写锁正被持有，则一直等待，直接可获取到读锁
      * @exception: 出错抛出 CSyscallException 异常
      */
    void lock_read();

    /***
      * 获取写锁，如果写锁正被持有，则一直等待，直接可获取到写锁
      * @exception: 出错抛出 CSyscallException 异常
      */
    void lock_write();

    /***
      * 尝试去获取读锁，如果写锁正被持有，则立即返回
      * @return: 如果成功获取了读锁，则返回 true，否则返回 false
      * @exception: 出错抛出 CSyscallException 异常
      */
    bool try_lock_read();

    /***
      * 尝试去获取写锁，如果写锁正被持有，则立即返回
      * @return: 如果成功获取了写锁，则返回 true，否则返回 false
      * @exception: 出错抛出 CSyscallException 异常
      */
    bool try_lock_write();

    /***
      * 以超时方式获取读锁，如果写锁正被持有，则等待指定的毫秒数，
      * 如果在指定的毫秒时间内，仍不能得到读锁，则立即返回
      * @millisecond: 等待获取读锁的毫秒数
      * @return: 如果在指定的毫秒时间内获取到了读锁，则返回 true，否则如果超时则返回 false
      * @exception: 出错抛出 CSyscallException 异常
      */
```

```
    bool timed_lock_read(uint32_t millisecond);

    /***
      * 以超时方式获取写锁，如果写锁正被持有，则等待指定的毫秒数，
      * 如果在指定的毫秒时间内，仍不能得到写锁，则立即返回
      * @millisecond: 等待获取写锁的毫秒数
      * @return: 如果在指定的毫秒时间内获取到了写锁，则返回 true，否则如果超时则返回 false
      * @exception: 出错抛出 CSyscallException 异常
      */
    bool timed_lock_write(uint32_t millisecond);
};
```

6.11.3.2. CReadLockHelper

```
/***
  * 读锁帮助类，用于自动释放读锁
  */
class CReadLockHelper
{
public:
    CReadLockHelper(CReadWriteLock& lock);

    /** 析构函数，会自动调用 unlock 解锁 */
    ~CReadLockHelper();
};
```

6.11.3.3. CWriteLockHelper

```
/***
  * 读锁帮助类，用于自动释放写锁
  */
class CWriteLockHelper
{
public:
    CWriteLockHelper(CReadWriteLock& lock);

    /** 析构函数，会自动调用 unlock 解锁 */
    ~CWriteLockHelper();
};
```

6.12. CEventQueue

6.12.1. 头文件

#include <sys/event_queue.h>

6.12.2. 主要功能

利用等待队列特性，常用于线程间的消息通信。如多个线程往队列里存消息，利用通知手段告诉另一个或多个线程去队列取消息。

6.12.3. 成员函数

```
/** 事件队列，总是线程安全
  * 特性 1: 如果队列为空，则可等待队列有数据时
  * 特性 2: 如果队列已满，则可等待队列为非满时
  * RawQueueClass 为原始队列类名，如 util::CArrayQueue
  */
template <class RawQueueClass>
class CEventQueue
{
public:
    /** 队列中的元素数据类型 */
    typedef typename RawQueueClass::_DataType DataType;

    /***
      * 构造一个事件队列
      * @pop_milliseconds: pop_front 时等待队列为非空时的毫秒数，如果为 0 则表示不等
      *                    待，这种情况下如果队列为空，则 pop_front 立即返回 false
      * @push_milliseconds: push_back 时等待队列为非满时的毫秒数，如果为 0 则表示不
      *                     等待，这种情况下如果队列已满，则 push_back 立即返回 false
      * @queue_max: 需要构造的队列大小
      */
    CEventQueue(uint32_t queue_max, uint32_t pop_milliseconds, uint32_t push_milliseconds);

    /** 判断队列是否已满 */
    bool is_full() const;

    /** 判断队列是否为空 */
    bool is_empty() const;
```

```
    /***
      * 返回队首元素
      * @elem: 存储队首元素
      * @return: 如果队列不为空则返回 true，否则返回 false
      */
    bool front(DataType& elem) const;

    /***
      * 弹出队首元素，如果队列为空，则会等待在构造函数中指定的毫秒数
      * @elem: 存储被弹出的队首元素
      * @return: 如果成功从队列弹出数据则返回 true，否则（队列为空或超时）返回 false
      * @exception: 可能抛出 CSyscallExceptoin 异常，其它异常和 RawQueue 有关
      */
    bool pop_front(DataType& elem);

    /***
      * 往队尾插入一个元素，如果队列已满，则会等待在构造函数中指定的毫秒数
      * @elem: 需要插入队尾的数据
      * @return: 如果成功往对尾插入了数据，则返回 true，否则（队列满或超时）返回 false
      * @exception: 可能抛出 CSyscallExceptoin 异常，其它异常和 RawQueue 有关
      */
    bool push_back(DataType elem);

    /** 得到队列中存储的元素个数 */
    uint32_t size() const;
};
```

6.12.4. 示例代码

```
#include <sys/thread.h>
#include <sys/sys_util.h>
#include <sys/event_queue.h>
#include <util/array_queue.h>
#include <sys/datetime_util.h>

// 定义处理消息线程，由主线程向它发消息
class CMyThread: public sys::CThread
{
public:
    // 设置队列大小为 100
    // pop 等待时长为 3000 毫秒
    // push 等待时长为 0 毫秒
    CMyThread()
        :_queue(100, 3000, 0)
```

```
    {
    }

    void push_message(int m)
    {
        if (_queue.push_back(m))
            printf("push %d SUCCESS by thread %u\n", m,
                    sys::CThread::get_current_thread_id());
        else
            printf("push %d FAILURE by thread %u\n", m
                    , sys::CThread::get_current_thread_id());
    }

private:
    virtual void run()
    {
        for (;;)
        {
            // 加判断，以保证所有消息都处理完才退出
            if (_stop && _queue.is_empty()) break;

            int m;
            printf("before pop ==> %s\n", sys::CDatetimeUtil::get_current_time().c_str());
            if (_queue.pop_front(m)) // 如果队列没有消息，这里会等待 3000 毫秒
                printf("pop %d ==> %s\n", m,
                        sys::CDatetimeUtil::get_current_time().c_str());
            else
                printf("pop NONE ==> %s\n",
                        sys::CDatetimeUtil::get_current_time().c_str());
        }
    }

private:
    // 使用整数类型的数组队列
    sys::CEventQueue<util::CArrayQueue<int> > _queue;
};

int main()
{
    CMyThread* thread = new CMyThread;
    // 使用引用计数帮助类，以协助自动销毁 thread
    sys::CRefCountHelper<CMyThread> ch(thread);

    try
```

```
    {
        (void)thread->start(); // 启动线程

        // 给线程发消息
        for (int i=0; i<10; ++i)
        {
            sys::CSysUtil::millisleep(2000);
            thread->push_message(i);
        }

        // 让 CMyThread 超时
        sys::CSysUtil::millisleep(3000);

        // 停止线程
        thread->stop();
    }
    catch (sys::CSyscallException& ex)
    {
        printf("Main exception: %s at %s:%d\n"
            , sys::CSysUtil::get_error_message(ex.get_errcode()).c_str()
            , ex.get_filename()
            , ex.get_linenumber());
    }

    return 0;
}
```

运行上面的代码，输出的内容大致（线程号和时间会不同，另外 pop 和 push 的顺序可能会有些不同，因为两线程为并行的，但输出的最后一行内容都应当为 **pop NONE**）如下：

```
pop 0 ==> 21:06:59
before pop ==> 21:06:59
push 0 SUCCESS by thread 782374640
push 1 SUCCESS by thread 782374640
pop 1 ==> 21:07:01
before pop ==> 21:07:01
push 2 SUCCESS by thread 782374640
pop 2 ==> 21:07:03
before pop ==> 21:07:03
push 3 SUCCESS by thread 782374640
pop 3 ==> 21:07:05
before pop ==> 21:07:05
push 4 SUCCESS by thread 782374640
pop 4 ==> 21:07:07
before pop ==> 21:07:07
```

```
push 5 SUCCESS by thread 782374640
pop 5 ==> 21:07:09
before pop ==> 21:07:09
push 6 SUCCESS by thread 782374640
pop 6 ==> 21:07:11
before pop ==> 21:07:11
push 7 SUCCESS by thread 782374640
pop 7 ==> 21:07:13
before pop ==> 21:07:13
push 8 SUCCESS by thread 782374640
pop 8 ==> 21:07:15
before pop ==> 21:07:15
push 9 SUCCESS by thread 782374640
pop 9 ==> 21:07:17
before pop ==> 21:07:17
pop NONE ==> 21:07:20
```

6.13. CMMap

6.13.1. 头文件

```
#include <sys/mmap.h>
```

6.13.2. 主要功能

提供内存映射文件功能，包括：

1) 以只读方式将文件映射到内存
2) 以只写方式将文件映射到内存
3) 以读和写方式将文件映射到内存
4) 以文件名的方式将文件映射到内存
5) 以文件句柄的方式将文件映射到内存
6) 从指定偏移位置开始映射
7) 限定可映射的最大字节数
8) 限定映射到内存的字节数

6.13.3. 成员函数

```
typedef struct
{
    int fd;         /** 如果是通过 fd 映射的，则为文件句柄的负值，否则为 mmap 操作时打开的文件句柄 */
```

```
    void* addr;   /** 文件映射到内存的地址，如果文件未被映射到内存则为 NULL */
    size_t len;   /** 计划映射到内存的大小 */
}mmap_t;

/** 内存文件映射操作类，所以成员均为静态类成员 */
class CMMap
{
public:
    /** 以只读方式将文件映射到内存
     * @fd: 文件句柄，调用者需要负责关闭此句柄，mmap_t 结构的 fd 成员为它的负值
     * @size: 需要映射到内存的大小，如果为 0 则映射整个文件
     * @offset: 映射的偏移位置
     * @size_max: 最大可映射字节数，超过此大小的文件将不会被映射到内存，mmap_t 结构的 addr 成员将为 NULL
     * @return: 返回指向 mmap_t 结构的指针，返回值总是不会为 NULL
     * @exception: 出错抛出 CSyscallException 异常
     */
    static mmap_t* map_read(int fd, size_t size=0, size_t offset=0, size_t size_max=0);

    /** 以只读方式将文件映射到内存
     * @filename: 文件名，mmap_t 结构的 fd 成员为打开此文件的句柄
     * @size_max: 最大可映射字节数，超过此大小的文件将不会被映射到内存，mmap_t 结构的 addr 成员将为 NULL
     * @return: 返回指向 mmap_t 结构的指针，返回值总是不会为 NULL
     * @exception: 出错抛出 CSyscallException 异常
     */
    static mmap_t* map_read(const char* filename, size_t size_max=0);

    /** 以只写方式将文件映射到内存
     * @fd: 文件句柄，调用者需要负责关闭此句柄，mmap_t 结构的 fd 成员为它的负值
     * @size: 需要映射到内存的大小，如果为 0 则映射整个文件
     * @offset: 映射的偏移位置
     * @size_max: 最大可映射字节数，超过此大小的文件将不会被映射到内存，mmap_t 结构的 addr 成员将为 NULL
     * @return: 返回指向 mmap_t 结构的指针，返回值总是不会为 NULL
     * @exception: 出错抛出 CSyscallException 异常
     */
    static mmap_t* map_write(int fd, size_t size=0, size_t offset=0, size_t size_max=0);

    /** 以只写方式将文件映射到内存
     * @filename: 文件名，mmap_t 结构的 fd 成员为打开此文件的句柄
     * @size_max: 最大可映射字节数，超过此大小的文件将不会被映射到内存，mmap_t 结构的 addr 成员将为 NULL
     * @return: 返回指向 mmap_t 结构的指针，返回值总是不会为 NULL
```

```
     * @exception: 出错抛出 CSyscallException 异常
     */
    static mmap_t* map_write(const char* filename, size_t size_max=0);

    /** 以读和写方式将文件映射到内存
     * @fd: 文件句柄，调用者需要负责关闭此句柄，mmap_t 结构的 fd 成员为它的负值
     * @size: 需要映射到内存的大小，如果为 0 则映射整个文件
     * @offset: 映射的偏移位置
     * @size_max: 最大可映射字节数，超过此大小的文件将不会被映射到内存，mmap_t 结构的 addr 成员将为 NULL
     * @return: 返回指向 mmap_t 结构的指针，返回值总是不会为 NULL
     * @exception: 出错抛出 CSyscallException 异常
     */
    static mmap_t* map_both(int fd, size_t size=0, size_t offset=0, size_t size_max=0);

    /** 以读和写方式将文件映射到内存
     * @filename: 文件名，mmap_t 结构的 fd 成员为打开此文件的句柄
     * @size_max: 最大可映射字节数，超过此大小的文件将不会被映射到内存，mmap_t 结构的 addr 成员将为 NULL
     * @return: 返回指向 mmap_t 结构的指针，返回值总是不会为 NULL
     * @exception: 出错抛出 CSyscallException 异常
     */
    static mmap_t* map_both(const char* filename, size_t size_max=0);

    /***
     * 释放已创建的内存映射，如果是通过指定文件名映射的，则关闭在 mmap 中打开的句柄
     * @ptr: 已创建的内存映射
     * @exception: 出错抛出 CSyscallException 异常
     */
    static void unmap(mmap_t* ptr);

    /***
     * 同步地将内存刷新到磁盘
     * @ptr: 指向 map_read、map_both 或 map_write 得到的 mmap_t 指针
     * @offset: 需要刷新的偏移位置
     * @length: 需要刷新的大小，如果为 0，则表示从偏移处到最尾，
     *           如果超出边界，则只刷新到边界
     * @invalid: 是否标识内存为无效
     * @exception: 出错抛出 CSyscallException 异常
     */
    static void sync_flush(mmap_t* ptr, size_t offset=0, size_t length=0, bool invalid=false);

    /***
```

```
        * 异步地将内存刷新到磁盘
        * @ptr: 指向 map_read、map_both 或 map_write 得到的 mmap_t 指针
        * @offset: 需要刷新的偏移位置
        * @length: 需要刷新的大小，如果为 0，则表示从偏移处到最尾，
        *           如果超出边界，则只刷新到边界
        * @invalid: 是否标识内存为无效
        * @exception: 出错抛出 CSyscallException 异常
        */
    static void async_flush(mmap_t* ptr, size_t offset=0, size_t length=0, bool invalid=false);
};

/***
  * CMMap 帮助类，用于自动卸载已经映射的文件
  */
class CMMapHelper
{
public:
    /** 构造一个 CMmap 帮助类 */
    CMMapHelper(mmap_t*& ptr);

    /** 析构函数，用来自动卸载已经映射的文件 */
    ~CMMapHelper();
};
```

6.13.4. 示例代码

```
#include <sys/mmap.h>

try
{
    sys::mmap_t* ptr = sys::CMMap::map_read("/tmp/map.txt");
    sys::CMMapHelper mh(ptr); // 使用帮助类

    printf("%s\n", (char*)(ptr->addr)); // 在屏幕上输出文件内容
} // 由于使用了帮助类，所以在这里会自动调用 sys::CMMap::unmap(ptr);
catch (sys::CSyscallException& ex)
{
    // 异常处理
}
```

6.14. CDynamicLinkingLoader

6.14.1. 头文件

#include <sys/dynamic_linking_loader.h>

6.14.2. 主要功能

1) 加载共享库
2) 通过符号名获取符号的地址，符号通常为函数

6.14.3. 成员函数

```
/***
  * 共享库加载工具类
  * 非线程安全类，不要跨线程使用
  */
class CDynamicLinkingLoader
{
public:
    /** 构造一个共享库加载器 */
    CDynamicLinkingLoader();
    ~CDynamicLinkingLoader();

    /**
      * 根据文件名加载共享库
      * @filename: 共享库文件名
      * @flag: 加载标志，可取值 RTLD_NOW 或 RTLD_LAZY
      * @return: 如果加载成功返回 true，否则返回 false，如果返回 false，
      *          可调用 get_error_message 获取失败原因
      */
    bool load(const char *filename, int flag = RTLD_NOW);

    /** 卸载已经加载的共享库 */
    void unload();

    /**
      * 得到出错信息，load 或 get_symbol 失败时，
      * 应当调用它来得到出错信息
      */
    const std::string& get_error_message() const { return _error_message; }
```

```
    /***
     * 根据符号名得到符号的地址
     * @symbol_name: 符号名，通常为函数名
     * @return: 返回符号对应的地址，通常为函数地址，如果失败则返回 NULL
     */
    void* get_symbol(const char *symbol_name);
};
```

6.14.4. 示例代码

```
// 加载共享库 test.so，并名字为 test 函数的址，然后调用 test 函数
#include <sys/dynamic_linking_loader.h>

typedef void (*TEST)();

sys::CDynamicLinkingLoader dll;
dll.load("test.so"); // 加载 test.so 共享库
TEST test = (TEST)dll.get_symbol("test"); // 通过函数得到函数地址
if (test != NULL)
    (*test)();
。。。
dll.unload(); // 不用了，卸载掉
```

6.15. CDatetimeUtil

6.15.1. 头文件

```
#include <sys/datetime_util.h>
```

6.15.2. 主要功能

1) 得到“YYYY-MM-DD HH:SS:MM”格式的当前日期时间
2) 判断指定年份是否为闰年
3) 将一个“YYYY-MM-DD HH:MM:SS”格式的字符串转换为日期时间结构
4) 单独获取当前时间中的年、月、日、小时、分钟和秒等

6.15.3. 成员函数

```
/** 日期时间工具类
  */
```

```
class CDatetimeUtil
{
public:
    /** 判断指定年份是否为闰年 */
    static bool is_leap_year(int year);

    /** 得到当前日期和时间，返回格式为: YYYY-MM-DD HH:SS:MM
     * @datetime_buffer: 用来存储当前日期和时间的缓冲区
     * @datetime_buffer_size: datetime_buffer 的字节大小，不应当于小“YYYY-MM-DD HH:SS:MM”
     */
    static void get_current_datetime(char* datetime_buffer, size_t datetime_buffer_size);
    static std::string get_current_datetime();

    /** 得到当前日期，返回格式为: YYYY-MM-DD
     * @date_buffer: 用来存储当前日期的缓冲区
     * @date_buffer_size: date_buffer 的字节大小，不应当于小“YYYY-MM-DD”
     */
    static void get_current_date(char* date_buffer, size_t date_buffer_size);
    static std::string get_current_date();

    /** 得到当前时间，返回格式为: HH:SS:MM
     * @time_buffer: 用来存储当前时间的缓冲区
     * @time_buffer_size: time_buffer 的字节大小，不应当于小“HH:SS:MM”
     */
    static void get_current_time(char* time_buffer, size_t time_buffer_size);
    static std::string get_current_time();

    /** 得到当前日期和时间结构
     * @current_datetime_struct: 指向当前日期和时间结构的指针
     */
    static void get_current_datetime_struct(struct tm* current_datetime_struct);

    /** 日期和时间 */
    static void to_current_datetime(struct tm* current_datetime_struct, char* datetime_buffer, size_t datetime_buffer_size);
    static std::string to_current_datetime(struct tm* current_datetime_struct);

    /** 仅日期 */
    static void to_current_date(struct tm* current_datetime_struct, char* date_buffer, size_t date_buffer_size);
    static std::string to_current_date(struct tm* current_datetime_struct);

    /** 仅时间 */
```

```
    static void to_current_time(struct tm* current_datetime_struct, char* time_buffer, size_t time_buffer_size);
    static std::string to_current_time(struct tm* current_datetime_struct);

    /** 仅年份 */
    static void to_current_year(struct tm* current_datetime_struct, char* year_buffer, size_t year_buffer_size);
    static std::string to_current_year(struct tm* current_datetime_struct);

    /** 仅月份 */
    static void to_current_month(struct tm* current_datetime_struct, char* month_buffer, size_t month_buffer_size);
    static std::string to_current_month(struct tm* current_datetime_struct);

    /** 仅天 */
    static void to_current_day(struct tm* current_datetime_struct, char* day_buffer, size_t day_buffer_size);
    static std::string to_current_day(struct tm* current_datetime_struct);

    /** 仅小时 */
    static void to_current_hour(struct tm* current_datetime_struct, char* hour_buffer, size_t hour_buffer_size);
    static std::string to_current_hour(struct tm* current_datetime_struct);

    /** 仅分钟 */
    static void to_current_minite(struct tm* current_datetime_struct, char* minite_buffer, size_t minite_buffer_size);
    static std::string to_current_minite(struct tm* current_datetime_struct);

    /** 仅秒钟 */
    static void to_current_second(struct tm* current_datetime_struct, char* second_buffer, size_t second_buffer_size);
    static std::string to_current_second(struct tm* current_datetime_struct);

    /***
     * 将一个字符串转换成日期时间格式
     * @str: 符合“YYYY-MM-DD HH:MM:SS”格式的日期时间
     * @datetime_struct: 存储转换后的日期时间
     * @return: 转换成功返回 true，否则返回 false
     */
    static bool datetime_struct_from_string(const char* str, struct tm* datetime_struct);
    static bool datetime_struct_from_string(const char* str, time_t* datetime);
};
```

6.15.4. 示例代码

下面的代码演示如何将一个字符串转换成一个日期时间结构。

```
#include "sys/datetime_util.h" // 引用头文件

int main()
{
    struct tm datetime_struct;
// 需要转换的字符串，格式必须为：YYYY-MM-DD HH:MM:SS
    const char* str = "2010-09-18 17:28:30";

    // 调用 datetime_struct_from_string 进行转换
    if (!sys::CDatetimeUtil::datetime_struct_from_string(str, &datetime_struct))
        printf("ERROR datetime_struct_from_string: %s\n", str); // 转换失败
    else
        printf("%s ==> \n"
               "\tyear=%04d\n"
               "\tmonth=%02d\n"
               "\tday=%02d\n"
               "\thour=%02d\n"
               "\tminitue=%02d\n"
               "\tsecond=%02d\n"
              ,str
              ,datetime_struct.tm_year+1900
              ,datetime_struct.tm_mon+1
              ,datetime_struct.tm_mday
              ,datetime_struct.tm_hour
              ,datetime_struct.tm_min
              ,datetime_struct.tm_sec);

    // 下面开始验证是否转换正确
    time_t dt = mktime(&datetime_struct);
    if (-1 == dt)
        printf("Error mktime\n");
    else
    {
        struct tm result;
        localtime_r(&dt, &result);

        // 如果前面转换成功，则下面的输出应当是：2010-09-18 17:28:30
        printf("%ld ==> \n"
               "\tyear=%04d\n"
               "\tmonth=%02d\n"
               "\tday=%02d\n"
```

```
                "\thour=%02d\n"
                "\tminitue=%02d\n"
                "\tsecond=%02d\n"
               ,dt
               ,result.tm_year+1900
               ,result.tm_mon+1
               ,result.tm_mday
               ,result.tm_hour
               ,result.tm_min
               ,result.tm_sec);
    }

    return 0;
}
```

6.16. CFileUtil

6.16.1. 头文件

```
#include <sys/file_util.h>
```

6.16.2. 主要功能

1) 文件复制
2) 取文件大小

6.16.3. 成员函数

```
/**
  * 文件相关的工具类
  */
class CFileUtil
{
public:
    /** 文件复制函数
      * @src_fd: 打开的源文件句柄
      * @dst_fd: 打开的目的文件句柄
      * @return: 返回文件大小
      * @exception: 出错抛出 CSyscallException 异常
      */
    static size_t file_copy(int src_fd, int dst_fd);
    static size_t file_copy(int src_fd, const char* dst_filename);
```

```
    static size_t file_copy(const char* src_filename, int dst_fd);
    static size_t file_copy(const char* src_filename, const char* dst_filename);

    /** 得到文件字节数
      * @fd: 文件句柄
      * @return: 返回文件字节数
      * @exception: 出错抛出 CSyscallException 异常
      */
    static off_t get_file_size(int fd);
    static off_t get_file_size(const char* filename);
};
```

6.17. CFSUtil

6.17.1. 头文件

和 CFSTable 在同一个文件中：#include <sys/fs_util.h>

6.17.2. 主要功能

1) 得到磁盘（分区）总的大小
2) 得到分区已经使用的大小
3) 得到分区未使用的大小
4) 得到分区块大小
5) 得到分区所支持的最大文件名长度
6) 得到分区支持的最多 inode 节点个数
7) 得到分区未使用的 inode 节点个数

6.17.3. 成员函数

```
/***
  * 操作文件系统的工具类
  * 通过与 CFSTable 类的结合，可以得到各分区的总大小和剩余大小等数据
  */
class CFSUtil
{
public:
    typedef struct
    {
        unsigned long block_bytes;          /** 每个块的字节数大小 */
```

```
        unsigned long total_block_nubmer;       /** 总的块数 */
        unsigned long free_block_nubmer;        /** 没用使用的块数 */
        unsigned long avail_block_nubmer;       /** 非 root 用户可用的块数 */
        unsigned long total_file_node_nubmer; /** 总的文件结点数 */
        unsigned long free_file_node_nubmer;   /** 没有使用的文件结点数 */
        unsigned long avail_file_node_nubmer; /** 非 root 用户可用的文件结点数 */
        unsigned long file_name_length_max;    /** 支持的最大文件名长度 */
    }fs_stat_t;

public:
    /***
      * 统计指定 fd 所指向的文件所在的文件系统，得到该文件系统的数据信息
      * @fd: 文件句柄
      * @stat_buf: 存储统计信息
      * @exception: 如果发生错误，则抛出 CSyscallException 异常
      */
    void stat_fs(int fd, fs_stat_t& stat_buf);

    /***
      * 统计指定路径所指向的文件所在的文件系统，得到该文件系统的数据信息
      * @path: 路径，为所在分区任意存在的路径即可
      * @stat_buf: 存储统计信息
      * @exception: 如果发生错误，则抛出 CSyscallException 异常
      */
    void stat_fs(const char* path, fs_stat_t& stat_buf);
};
```

6.18. CFSTable

6.18.1. 头文件

和 CFSUtil 在同一个文件中：#include <sys/fs_util.h>

6.18.2. 主要功能

1) 得到系统中所有的分区
2) 得到系统中当前已经加载（mounted）的分区

6.18.3. 成员函数

```
/***
```

```
 * 文件系统表
 * 示例:
 * CFSTable fst;
 * if (fst) {
 *       fs_entry_t ent;
 *       while (get_entry(ent)) {
 *             printf("fs_name=%s\n", ent.fs_name);
 *             printf("dir_path=%s\n", ent.dir_path);
 *             printf("type_name=%s\n", ent.type_name);
 *       }
 * }
 */
class CFSTable
{
public:
    typedef struct
    {
        std::string fs_name;     /** 文件系统名 */
        std::string dir_path;    /** 文件系统所加载的目录路径 */
        std::string type_name; /** 文件系统类型名，如 ext3 等 */
    }fs_entry_t;

public:
    /***
      * 构造对象，并打开文件系统表。对象生成后，应当先判断对象是否可用，如：
      * CFSTable mt; if (mt) { }，这测试用 if 语句不能少，只有为真时才可以调用 get_entry
      * @mounted: 是否只针对已经加载的文件系统
      * @fsname_prefix: 所关心的文件系统名前缀，只有匹配的才关心，如果为 NULL，则表示所有的
      */
    CFSTable(bool mounted=true, const char* fsname_prefix="/");
    ~CFSTable();

    /** 复位，可重新调用 get_entry 获取文件系统列表 */
    void reset();

    /***
      * 从文件系统表中取一条记录
      * @return: 如果取到记录，则返回指针 entry 的指针，否则返回 NULL，表示已经遍历完所有的
      */
    fs_entry_t* get_entry(fs_entry_t& entry);

    /***
```

```
        * 判断在构造函数中是否成功打开了文件系统表
        * @return: 如果已经打开，则返回 true，否则返回 false
        */
    operator bool() const;
};
```

6.19. CInfo

6.19.1. 头文件

```
#include <sys/info.h>
```

6.19.2. 主要功能

1) 得到当前进程实时的 CPU 和内存数据
2) 得到系统实时 CPU 和内存数据
3) 得到当前网卡流量
4) 得到内核版本号

6.19.3. 成员函数

```
/***
  * 用来获取系统、内核和进程的各类实时信息，如CPU和内存数据
  */
class CInfo
{
public:
    /***
      * 系统当前实时信息
      */
    typedef struct
    {
        long uptime_second;             /* Seconds since boot */
        unsigned long average_load[3];  /* 1, 5, and 15 minute load averages */
        unsigned long ram_total;        /* Total usable main memory size */
        unsigned long ram_free;         /* Available memory size */
        unsigned long ram_shared;       /* Amount of shared memory */
        unsigned long ram_buffer;       /* Memory used by buffers */
        unsigned long swap_total;       /* Total swap space size */
        unsigned long swap_free;        /* swap space still available */
```

```
        unsigned short process_number;  /* Number of current processes */
    }sys_info_t;

    /***
      * 当前进程时间信息
      */
    typedef struct
    {
        long user_time;             /* user time */
        long system_time;          /* system time */
        long user_time_children;    /* user time of children */
        long system_time_children; /* system time of children */
    }process_time_t;

    /***
      * 当前系统CPU信息
      */
    typedef struct
    {
        // 单位: jiffies, 1jiffies=0.01秒
uint32_t total;
        uint32_t user;    /** 从系统启动开始累计到当前时刻，处于用户态的运行时间，不包含 nice值为负进程 */
        uint32_t nice;    /** 从系统启动开始累计到当前时刻，nice值为负的进程所占用的CPU时间 */
        uint32_t system;  /** 从系统启动开始累计到当前时刻，处于核心态的运行时间 */
        uint32_t idle;    /** 从系统启动开始累计到当前时刻，除IO等待时间以外的其它等待时间 */
        uint32_t iowait;  /** 从系统启动开始累计到当前时刻，IO等待时间(2.5.41) */
        uint32_t irq;     /** 从系统启动开始累计到当前时刻，硬中断时间(2.6.0) */
        uint32_t softirq; /** 从系统启动开始累计到当前时刻，软中断时间(2.6.0) */
        //uint32_t stealstolen; /** which is the time spent in other operating systems when running in a virtualized environment(2.6.11) */
        //uint32_t guest;       /** which is the time spent running a virtual  CPU for  guest operating systems under the control of the Linux kernel(2.6.24) */
    }cpu_info_t;

    /***
      * 当前系统内存信息
      */
    typedef struct
    {
        uint32_t mem_total;
```

```
        uint32_t mem_free;
        uint32_t buffers;
        uint32_t cached;
        uint32_t swap_cached;
        uint32_t swap_total;
        uint32_t swap_free;
    }mem_info_t;

    /***
      * 内核版本号
      */
    typedef struct
    {
        int16_t major;    /** 主版本号 */
        int16_t minor;    /** 次版本号(如果次版本号是偶数，那么内核是稳定版；若是奇数则是开发版) */
        int16_t revision; /** 修订版本号 */
    }kernel_version_t;

    /***
      * 当时进程状态信息
      *
      * 进程的状态值:
        D    Uninterruptible sleep (usually IO)
        R    Running or runnable (on run queue)
        S    Interruptible sleep (waiting for an event to full)
        T    Stopped, either by a job control signal or because it is being traced.
        W    paging (not valid since the 2.6.xx kernel)
        X    dead (should never be seen)
        Z    Defunct ("zombie") process, terminated but not reaped by its parent.
      */
    typedef struct
    {
        /** 01 */ pid_t pid;                    /** 进程号，其允许的最大值，请查看/proc/sys/kernel/pid_max */
        /** 02 */ char comm[FILENAME_MAX];      /** 进程的名字，不包括路径 */
        /** 03 */ char state;                   /** 进程的状态 */
        /** 04 */ pid_t ppid;                   /** 父进程号 */
        /** 05 */ pid_t pgrp;                   /** 进程组号 */
        /** 06 */ pid_t session;                /** 进程会话号 */
        /** 07 */ int tty_nr;                   /** The tty the process uses */
        /** 08 */ pid_t tpgid;                  /** The tty the process uses */
        /** 09 */ unsigned int flags;           /** The kernel flags word of the process (%lu before Linux 2.6.22) */
```

```
        /** 10 */ unsigned long minflt;          /** The number of minor faults the process has made which have not required loading a memory page from disk */
        /** 11 */ unsigned long cminflt;         /** The number of minor faults that the process's waited-for children have made */
        /** 12 */ unsigned long majflt;          /** The number of major faults the process has made which have required loading a memory page from disk */
        /** 13 */ unsigned long cmajflt;         /** The number of major faults that the process's waited-for children have made */
        /** 14 */ unsigned long utime;           /** The number of jiffies that this process has been scheduled in user mode */
        /** 15 */ unsigned long stime;           /** The number of jiffies that this process has been scheduled in kernel mode */
        /** 16 */ long cutime;                   /** The number of jiffies that this process's waited-for children have been scheduled in user mode */
        /** 17 */ long cstime;                   /** The number of jiffies that this process's waited-for children have been scheduled in kernel mode */
        /** 18 */ long priority;                 /** The standard nice value, plus fifteen.  The value is never negative in the kernel */
        /** 19 */ long nice;                     /** The nice value ranges from 19 (nicest) to -19 (not nice to others) */
        /** 20 */ long num_threads;              /** Number of threads in this process (since Linux 2.6).  Before kernel 2.6, this field was hard coded to 0 as a placeholder */
        /** 21 */ long itrealvalue;              /** The time in jiffies before the next SIGALRM is sent to the process due to an interval timer.2.6.17, this field is no longer maintained, and is hard coded as 0 */
        /** 22 */ long long starttime;           /** The time in jiffies the process started after system boot */
        /** 23 */ unsigned long vsize;           /** Virtual memory size in bytes */
        /** 24 */ long rss;          /** Resident Set Size: number of pages the process has in real memory, minus 3 for administrative purposes */
        /** 25 */ unsigned long rlim;    /** Current limit in bytes on the rss of the process (usually 4294967295 on i386) */
        /** 26 */ unsigned long startcode;  /** The address above which program text can run */
        /** 27 */ unsigned long endcode;    /** The address below which program text can run */
        /** 28 */ unsigned long startstack;  /** The address of the start of the stack */
        /** 29 */ unsigned long kstkesp;  /** The current value of esp (stack pointer), as found in the kernel stack page for the process */
        /** 30 */ unsigned long kstkeip;  /** The current EIP (instruction pointer) */
```

```
        /** 31 */ unsigned long signal;       /** The bitmap of pending signals */
        /** 32 */ unsigned long blocked;      /** The bitmap of blocked signals */
        /** 33 */ unsigned long sigignore;    /** The bitmap of ignored signals */
        /** 34 */ unsigned long sigcatch;     /** The bitmap of caught signals */
        /** 35 */ unsigned long nswap; /** Number of pages swapped (not maintained).
*/
        /** 36 */ unsigned long cnswap;     /** Cumulative nswap for child processes
(not maintained) */
        /** 37 */ int exit_signal;  /** Signal to be sent to parent when we die (since
Linux 2.1.22) */
        /** 38 */ int processor; /** CPU number last executed on (since Linux 2.2.8)
*/
    }process_info_t;

    /***
      * 网卡流量数据结构
      */
    typedef struct
    {
        /** 01 */ char interface_name[INTERFACE_NAME_MAX]; /** 网卡名，如eth0 */

                  /** 接收数据 */
        /** 02 */ unsigned long receive_bytes;       /** 此网卡接收到的字节数 */
        /** 03 */ unsigned long receive_packets;
        /** 04 */ unsigned long receive_errors;
        /** 05 */ unsigned long receive_dropped;
        /** 06 */ unsigned long receive_fifo_errors;
        /** 07 */ unsigned long receive_frame;
        /** 08 */ unsigned long receive_compressed;
        /** 09 */ unsigned long receive_multicast;

                  /** 发送数据 */
        /** 10 */ unsigned long transmit_bytes;     /** 此网卡已发送的字节数 */
        /** 11 */ unsigned long transmit_packets;
        /** 12 */ unsigned long transmit_errors;
        /** 13 */ unsigned long transmit_dropped;
        /** 14 */ unsigned long transmit_fifo_errors;
        /** 15 */ unsigned long transmit_collisions;
        /** 16 */ unsigned long transmit_carrier;
        /** 17 */ unsigned long transmit_compressed;
    }net_info_t;

    /***
      * 进程页信息结构
```

```
     */
    typedef struct
    {
        long size;     /** 程序大小 */
        long resident; /** 常驻内存空间大小 */
        long share;    /** 共享内存页数 */
        long text;     /** 代码段占用内存页数 */
        long lib;      /** 数据/堆栈段占用内存页数 */
        long data;     /** 引用库占用内存页数 */
    }process_page_info_t;

public:
    /** 获取系统信息，具体请参考sys_info_t的描述 */
    static bool get_sys_info(sys_info_t& sys_info);

    /** 获取内存信息，具体请参考mem_info_t的描述 */
    static bool get_mem_info(mem_info_t& mem_info);

    /** 获取总CPU信息，具体请参考cpu_info_t的描述 */
    static bool get_cpu_info(cpu_info_t& cpu_info);

    /** 获取所有CPU信息，具体请参考cpu_info_t的描述 */
    static int get_cpu_info_array(std::vector<cpu_info_t>& cpu_info_array);

    /** 得到内核版本号 */
    static bool get_kernel_version(kernel_version_t& kernel_version);

    /** 获取进程信息，具体请参考process_info_t的描述 */
    static bool get_process_info(process_info_t& process_info);

    /** 获取进程页信息，具体请参考process_page_info_t的描述 */
    static bool get_process_page_info(process_page_info_t& process_page_info);

    /** 获取进程运行时间数据，具体请参考process_time_t的描述 */
    static bool get_process_times(process_time_t& process_time);

    /***
     * 获取网卡流量等信息
     * 流量 = (当前获取的值 - 上一时间获取的值) / 两次间隔的时长
     * @interface_name: 网卡名，如eth0等
     * @net_info: 存储网卡流量等数据
     */
    static bool get_net_info(const char* interface_name, net_info_t& net_info);
    static bool get_net_info_array(std::vector<net_info_t>& net_info_array);
```

};

6.20. CUtil

6.20.1. 头文件

#include <sys/util.h>

6.20.2. 主要功能

1) 线程安全的毫秒级 sleep
2) 得到当前进程所在文件的全路径
3) 根据文件 fd 得到文件名
4) 得到当前调用栈，要求编译时加上-rdynamic 开关
5) 递归创建目录
6) 得到 CPU 个数
7) 获取指定目录的大小
8) 开启和关闭程序 coredump
9) 判断 fd 或 path 是否为文件、或软链接、或目录

6.20.3. 成员函数

```
/***
  * 与系统调用有关的工具类函数实现
  */
class CUtil
{
public:
    /** 线程安全的毫秒级 sleep 函数
      * @millisecond: 需要 sleep 的毫秒数
      */
    static void millisleep(uint32_t millisecond);

    /** 得到指定系统调用错误码的字符串错误信息
      * @errcode: 系统调用错误码
      * @return: 系统调用错误信息
      */
    static std::string get_error_message(int errcode);

    /** 得到当前进程所属可执行文件所在的绝对路径，结尾符不含反斜杠 */
    static std::string get_program_path();
```

```
    /** 得到与指定 fd 相对应的文件名，包括路径部分
      * @fd: 文件描述符
      * @return: 文件名，包括路径部分，如果失败则返回空字符串
      */
    static std::string get_filename(int fd);

    /** 得到 CPU 核个数
      * @return: 如果成功，返回大于 0 的 CPU 核个数，否则返回 0
      */
    static uint16_t get_cpu_number();

    /** 得到当前调用栈
      * 注意事项: 编译源代码时带上-rdynamic 和-g 选项，否则可能看到的是函数地址，而不是函数符号名称
      * @call_stack: 存储调用栈
      * @return: 成功返回 true，否则返回 false
      */
    static bool get_backtrace(std::string& call_stack);

    /** 得到指定目录字节数大小，非线程安全函数，同一时刻只能被一个线程调用
      * @dirpath: 目录路径
      * @return: 目录字节数大小
      */
    static off_t du(const char* dirpath);

    /** 得到内存页大小 */
    static int get_page_size();

    /** 得到一个进程可持有的最多文件(包括套接字等)句柄数 */
    static int get_fd_max();

    /***
      * 递归的创建目录
      * @dirpath: 需要创建的目录
      * @permissions: 目录权限，取值可以为下列的任意组合:
      *                    S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR
      *                    S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP
      *                    S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH
      * @exception: 出错则抛出 CSyscallException
      */
    static void create_directory(
                    const char* dirpath, int permissions=DIRECTORY_DEFAULT_PERM);

    /***
```

```
        * 递归的创建目录
        * @dirpath: 需要创建的目录
        * @permissions: 目录权限
        * @exception: 出错则抛出 CSyscallException
        */
    static void create_directory_recursive(
                    const char* dirpath, int permissions=DIRECTORY_DEFAULT_PERM);

    /** 下列 is_xxx 函数如果发生错误，则抛出 CSyscallException 异常 */
    static bool is_file(int fd);                    /** 判断指定 fd 对应的是否为文件 */
    static bool is_file(const char* path);          /** 判断指定 Path 是否为一个文件 */
    static bool is_link(int fd);                    /** 判断指定 fd 对应的是否为软链接 */
    static bool is_link(const char* path);          /** 判断指定 Path 是否为一个软链接 */
    static bool is_directory(int fd);               /** 判断指定 fd 对应的是否为目录 */
    static bool is_directory(const char* path);     /** 判断指定 Path 是否为一个目录 */

    /***
        * 是否允许当前进程生成 coredump 文件
        * @enable: 如果为 true，则允许当前进程生成 coredump 文件，否则禁止
        * @core_file_size: 允许生成的 coredump 文件大小，如果取值小于 0，则表示不限制文件大小
        * @exception: 如果调用出错，则抛出 CSyscallException 异常
        */
    static void enable_core_dump(bool enabled=true, int core_file_size=-1);

    /** 得到当前进程名，包括路径部分 */
    static const char* get_program_name();

    /** 得到当前进程的短名字，即纯文件名 */
    static const char* get_program_short_name();
};
```

6.21. CRawObjectPool 和 CThreadObjectPool

6.21.1. 头文件

```
#include <sys/object_pool.h>
```

6.21.2. 主要功能

1) 提供性能高，但非线程安全的对象池；
2) 提供线程安全的对象池；

3) 支持对象池中对象不够时，自动从堆中创建新对象。

6.21.3. CPoolObject

```
/***
  * 池对象基类
  * 所以需要对象池的类都应当从它继承而来
  */
class CPoolObject
{
public:
    /** 构造一个池对象 */
    CPoolObject();

    /** 设置池对象的在池对象数组中的序号，0表示不是池中的对象 */
    void set_index(uint32_t index);

    /** 得到池对象的在池对象数组中的序号 */
    uint32_t get_index() const;

    /** 设置对象是否在池中的状态 */
    void set_in_pool(bool in_pool);

    /** 判断池对象是否在池中 */
    bool is_in_pool() const;
};
```

6.21.4. CRawObjectPool

```
/***
  * 裸对象池实现，性能高但非线程安全
  * 要求ObjectClass类必须是CPoolObject的子类
  */
template <class ObjectClass>
class CRawObjectPool
{
public:
    /***
      * 构造一个非线程安全的裸对象池
      * @use_heap: 当对象池中无对象时，是否从堆中创建对象
      */
    CRawObjectPool(bool use_heap);
```

```
    /** 析构裸对象池 */
    ~CRawObjectPool();

    /***
      * 创建对象池
      * @object_number: 需要创建的对象个数
      */
    void create(uint32_t object_number);

    /** 销毁对象池 */
    void destroy();

    /***
      * 从对象池中借用一个对象，并将对象是否在池中的状态设置为false
      * @return: 如果对象池为空，但允许从堆中创建对象，则从堆上创建一个新对象，并返回它，
      *          如果对象池为空，且不允许从堆中创建对象，则返回NULL，
      *          如果对象池不为空，则从池中取一个对象，并返回指向这个对象的指针
      */
    ObjectClass* borrow();

    /***
      * 将一个对象归还给对象池
      * @object: 指向待归还给对象池的对象指针，如果对象并不是对象池中的对象，则delete它，
      *          否则将它放回对象池，并将是否在对象池中的状态设置为true
      */
    void pay_back(ObjectClass* object);

    /** 得到总的对象个数，包括已经借出的和未借出的 */
    uint32_t get_pool_size() const;

    /** 得到对象池中还未借出的对象个数 */
    volatile uint32_t get_avaliable_number() const;
};
```

6.21.5. CThreadObjectPool

```
/***
  * 线程安全的对象池，性能较CRawObjectPool低
  * 要求ObjectClass类必须是CPoolObject的子类
  */
template <class ObjectClass>
class CThreadObjectPool
```

```
{
public:
    /***
      * 构造一个线程安全的裸对象池
      * @use_heap: 当对象池中无对象时，是否从堆中创建对象
      */
    CThreadObjectPool(bool use_heap);

    /***
      * 创建对象池
      * @object_number: 需要创建的对象个数
      */
    void create(uint32_t object_number);

    /** 销毁对象池 */
    void destroy();

    /** 向对象池借用一个对象 */
    ObjectClass* borrow();

    /** 将一个对象归还给对象池 */
    void pay_back(ObjectClass* object);

    /** 得到总的对象个数，包括已经借出的和未借出的 */
    uint32_t get_pool_size() const;

    /** 得到对象池中还未借出的对象个数 */
    volatile uint32_t get_avaliable_number() const;
};
```

6.22. CRawMemPool 和 CThreadMemPool

6.22.1. 头文件

#include <sys/mem_pool.h>

6.22.2. 主要功能

1) 提供非线程安全的内存池；
2) 提供线程安全的内存池；
3) 可设置在内存池不够时，从堆上分配内存；
4) 可设置内存警戒边界，方便定位内存越界问题。

6.22.3. CRawMemPool

```
/***
  * 裸内存池实现，性能高但非线程安全
  */
class CRawMemPool
{
public:
    CRawMemPool();
    ~CRawMemPool();

    /** 销毁由create创建的内存池 */
    void destroy();

    /***
      * 创建内存池
      * @bucket_size: 内存大小
      * @bucket_number: 内存个数
      * @use_heap: 内存池不够时，是否从堆上分配
      * @guard_size: 警戒大小
      * @guard_flag: 警戒标识
      */
    void create(uint16_t bucket_size, uint32_t bucket_number, bool use_heap=true,
uint8_t guard_size=1, char guard_flag='m');

    /***
      * 分配内存内存
      *@return: 如果内存池不够，且设置了从堆上分配内存，则返回从堆上分配的内存，
      *          否则如果内存池不够时返回NULL，否则返回从内存池中分配的内存
      */
    void* allocate();

    /***
      * 回收内存内存
      * @bucket: 需要被回收到内存池中的内存，如果不是内存池中的内存，
      *          但create时允许从堆分配，则直接释放该内存，否则如果在池内存范围
内，
      *          则检查是否为正确的池内存，如果是则回收并返回true，其它情况返回
false
      * @return: 如果被回收或删除返回true，否则返回false
      */
    bool reclaim(void* bucket);
```

```
    /** 返回当内存池不够用时，是否从堆上分配内存 */
    bool use_heap() const;

    /** 得到警戒值大小 */
    uint8_t get_guard_size() const;

    /** 得到池大小，也就是池中可分配的内存个数 */
    uint32_t get_pool_size() const;

    /** 得到内存池可分配的内存大小 */
    uint16_t get_bucket_size() const;

    /** 得到内存池中，当前还可以分配的内存个数 */
    uint32_t get_available_number() const;
};
```

6.22.4. CThreadMemPool

```
/***
  * 线程安全的内存池，性能较CRawMemPool要低
  */
class CThreadMemPool
{
public:
    /** 销毁由create创建的内存池 */
    void destroy();

    /***
      * 创建内存池
      * @bucket_size: 内存大小
      * @bucket_number: 内存个数
      * @use_heap: 内存池不够时，是否从堆上分配
      * @guard_size: 警戒大小
      * @guard_flag: 警戒标识
      */
    void create(uint16_t bucket_size, uint32_t bucket_number, bool use_heap=true,
uint8_t guard_size=1, char guard_flag='m');

    /***
      * 分配内存内存
      *@return: 如果内存池不够，且设置了从堆上分配内存，则返回从堆上分配的内存，
      *          否则如果内存池不够时返回NULL，否则返回从内存池中分配的内存
      */
    void* allocate();
```

```
    /***
     * 回收内存内存
     * @bucket: 需要被回收到内存池中的内存，如果不是内存池中的内存，
     *          但create时允许从堆分配，则直接释放该内存，否则如果在池内存范围内，
     *          则检查是否为正确的池内存，如果是则回收并返回true，其它情况返回false
     * @return: 如果被回收或删除返回true，否则返回false
     */
    bool reclaim(void* bucket);

    /** 返回当内存池不够用时，是否从堆上分配内存 */
    bool use_heap() const;

    /** 得到警戒值大小 */
    uint8_t get_guard_size() const;

    /** 得到池大小，也就是池中可分配的内存个数 */
    uint32_t get_pool_size() const;

    /** 得到内存池可分配的内存大小 */
    uint16_t get_bucket_size() const;

    /** 得到内存池中，当前还可以分配的内存个数 */
    uint32_t get_available_number() const;
};
```

7. net 库

7.1. 库文件

libnet.so

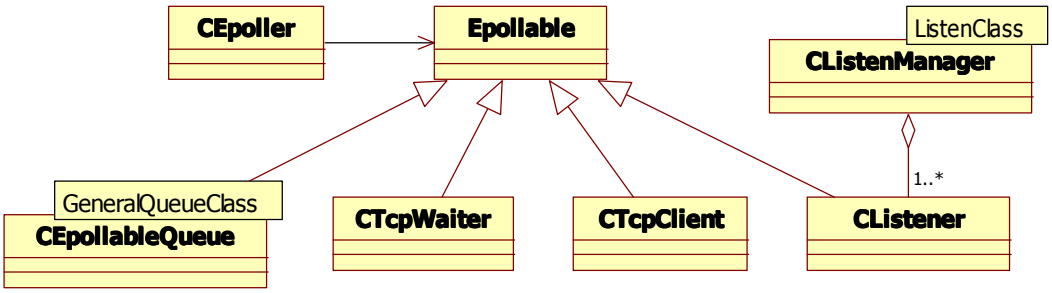
7.2. 名字空间

net，如：using namespace net;

7.3. 依赖库

libnet.so 依赖于 libutil.so 和 libsys.so 两个库。

7.4. 主要类图



7.5. ip_address_t

```
/** IP 地址，兼容 IPV4 和 IPV6 */
class ip_address_t
{
public:
    /** 构造一个 127.0.0.1 地址 */
    ip_address_t();

    /** 构造一个 IPV4 地址 */
    ip_address_t(uint32_t ipv4);

    /***
      * 构造一个 IPV6 地址
      */
    ip_address_t(const uint32_t* ipv6);

    /***
      * 构造一个可能是 IPV6 也可能是 IPV4 的地址
      * @ip: 字符串 IP 地址，如果为 NULL，则构造一个 0.0.0.0 的 IPV4 地址
      * @exception: 如果为无效 IP 地址，则抛出 CSyscallException 异常
      */
    ip_address_t(const char* ip);

    /** 拷贝构造 */
    ip_address_t(const ip_address_t& ip);

    /** 判断是否为 IPV6 地址 */
    bool is_ipv6() const;

    /** 转换成字符串 */
```

```
        std::string to_string() const;

        /** 得到地址数据的有效字节数 */
        size_t get_address_data_length() const;

        /** 得到地址数据 */
        const uint32_t* get_address_data() const;

        /** 判断是否为 0.0.0.0 地址 */
        bool is_zero_address() const;

        /** 判断是否为广播地址 */
        bool is_broadcast_address() const;

public: // 赋值和比较操作
        ip_address_t& operator =(uint32_t ipv4);
        ip_address_t& operator =(const uint32_t* ipv6);
        ip_address_t& operator =(const char* ip);
        ip_address_t& operator =(const ip_address_t& other);
        bool operator <(const ip_address_t& other) const;
        bool operator >(const ip_address_t& other) const;
        bool operator ==(const ip_address_t& other) const;
};
```

7.6. ipv4_node_t 和 ipv6_node_t

```
/***
  * IPV4 结点类型
  */
typedef struct
{
        uint16_t port;   /** 端口号 */
        uint32_t ip;     /** IPV4 地址 */

        /* 构造、赋值和比较函数 */
        ipv4_node_t();
        ipv4_node_t(uint16_t new_port, uint32_t new_ip);
        ipv4_node_t(const ipv4_node_t& other);
        ipv4_node_t& operator =(const ipv4_node_t& other);
        bool operator ==(const ipv4_node_t& other) const;
}ipv4_node_t;

/***
  * IPV6 结点类型
```

```
  */
typedef struct
{
    uint16_t port;   /** 端口号 */
    uint32_t ip[4]; /** IPV6 地址 */

    /* 构造、赋值和比较函数 */
    ipv6_node_t();
    ipv6_node_t(uint16_t new_port, const uint32_t* new_ip);
    ipv6_node_t(const ipv6_node_t& other);
    ipv6_node_t& operator =(const ipv6_node_t& other);
    bool operator ==(const ipv6_node_t& other) const;
}ipv6_node_t;
```

7.6.1. ipv4_node_hasher

求 ipv4_node_t 类型 hash 值仿函数类，用法示例：

```
template <class ValueClass>
class ipv4_hash_map: public __gnu_cxx::hash_map<net::ipv4_node_t*, ValueClass,
net::ipv4_node_hasher, net::ipv4_node_comparer>
{
};
```

7.6.2. ipv4_node_comparer

ipv4_node_t 类型比较是否相等仿函数类。

7.6.3. ipv6_node_hasher

求 ipv6_node_t 类型 hash 值仿函数类，用法示例：

```
template <class ValueClass>
class ipv6_hash_map: public __gnu_cxx::hash_map<net::ipv6_node_t*, ValueClass,
net::ipv6_node_hasher, net::ipv6_node_comparer>
{
};
```

7.6.4. ipv6_node_comparer

ipv6_node_t 类型比较是否相等仿函数类。

7.7. CEpollable

7.7.1. 头文件

#include <net/epollable.h>

7.7.2. 主要功能

1) 对所有可以使用 Epoll 监控对象的抽象，提供一个统一的 Epoll 事件回调接口 handle_epoll_event，供 Epoller 调用
2) 提供对文件句柄的公共操作方法，包括：
   a) 设置和获取 fd 值
   b) 设置和获取被注册的 Epoll 事件
   c) 判断 fd 是否为阻塞模式
   d) 判断 fd 是否为延迟模式
   e) 将 fd 设置为阻塞模式
   f) 将 fd 设置为延迟模式

7.7.3. 成员函数

7.7.3.1. epoll_event_t

```
/***
  * CEpollable::handle_epoll_event 的返回值类型
  */
typedef enum
{
    epoll_none,         /** 调用者什么也不用做 */
    epoll_read,         /** 需要 Epoll 设置为只读事件 */
    epoll_write,        /** 需要 Epoll 设置为只写事件 */
    epoll_read_write,   /** 需要 Epoll 设置为读和写事件 */
    epoll_close,        /** 需要从 Epoll 中剔除，并关闭 */
    epoll_remove,       /** 需要从 Epoll 中剔除，但不关闭 */
    epoll_destroy       /** 需要从 Epoll 中剔除，并且对象应当被销毁 */
}epoll_event_t;
```

7.7.3.2. CEpollable

```
/***
```

```
 * 可 pool 类，所有可使用 Epoll/Pool 监控对象的基类
 */
class CEpollable
{
public:
    CEpollable();
    virtual ~CEpollable();

    /** 关闭句柄 */
    virtual void close();

    /** 得到句柄值 */
    int get_fd() const;

    /** 得到设置的 Epoll 事件 */
    int get_epoll_events() const;

    /***
     * 判断指定 fd 是否为非阻塞的
     * @return: 如果 fd 为非阻塞的，则返回 true，否则返回 false
     * @exception: 如果发生错误，则抛出 CSyscallException 异常
     */
    bool is_nonblock();

    /***
     * 判断指定 fd 是否为非延迟的
     * @return: 如果 fd 为非延迟的，则返回 true，否则返回 false
     * @exception: 如果发生错误，则抛出 CSyscallException 异常
     */
    bool is_nodelay();

    /***
     * 修改对象的非阻塞属性
     * @yes: 是否设置为非阻塞标识，如果为 true，则设置为非阻塞，否则设置为阻塞
     * @exception: 如果发生错误，则抛出 CSyscallException 异常
     */
    void set_nonblock(bool yes);

    /***
     * 修改对象的非延迟属性
     * @yes: 是否设置为非延迟标识，如果为 true，则设置为非延迟，否则设置为延迟
     * @exception: 如果发生错误，则抛出 CSyscallException 异常
     */
    void set_nodelay(bool yes);
```

```
public:
    /***
     * Epoll 事件回调函数
     * @ptr: 对象指针
     * @events: 发生的 Epoll 事件
     * @return: 请参见 epoll_event_t 的说明
     * @exception: 系统调用出错，抛出 CSyscallException 异常
     */
    virtual epoll_event_t handle_epoll_event(void* ptr, uint32_t events);

protected: // 供继承的子类使用
    /***
     * 关闭，如果执行了关闭(_fd 不为-1 时)返回 true，否则返回 false
     * 注意它不会调用 before_close
     */
    bool do_close();

    /** 设置句柄 */
    void set_fd(int fd);
};
```

7.7.4. 名字空间全局函数

以下函数是位于 **net** 名字空间内的全局函数。

```
/***
 * 判断指定 fd 是否具有指定的标志
 * @fd: 文件或套接字等句柄
 * @flags: 指定的标志值
 * @return: 如果具有指定的标志值，则返回 true，否则返回 false
 * @exception: 如果发生错误，则抛出 CSyscallException 异常
 */
bool has_the_flags(int fd, int flags);

/***
 * 判断指定 fd 是否为非阻塞的
 * @fd: 文件或套接字等句柄
 * @return: 如果 fd 为非阻塞的，则返回 true，否则返回 false
 * @exception: 如果发生错误，则抛出 CSyscallException 异常
 */
bool is_nonblock(int fd);

/***
```

```
    * 判断指定 fd 是否为非延迟的
    * @fd: 文件或套接字等句柄
    * @return: 如果 fd 为非延迟的，则返回 true，否则返回 false
    * @exception: 如果发生错误，则抛出 CSyscallException 异常
    */
bool is_nodelay(int fd);

/***
    * 为指定的 fd 增加或删除指定的标志
    * @fd: 文件或套接字等句柄
    * @yes: 是否设置为指定标志，如果为 true，则增加 status 指定的标志，否则去掉 status 指定的标志
    * @flags: 标志值
    * @exception: 如果发生错误，则抛出 CSyscallException 异常
    */
void set_socket_flags(int fd, bool yes, int flags);

/***
    * 为指定 fd 的增加或删除非阻塞标志
    * @fd: 文件或套接字等句柄
    * @yes: 是否设置为非阻塞标志，如果为 true，则设置为非阻塞，否则设置为阻塞
    * @exception: 如果发生错误，则抛出 CSyscallException 异常
    */
void set_nonblock(int fd, bool yes);

/***
    * 为指定 fd 的增加或删除非延迟标志
    * @fd: 文件或套接字等句柄
    * @yes: 是否设置为非延迟标志，如果为 true，则设置为非延迟，否则设置为延迟
    * @exception: 如果发生错误，则抛出 CSyscallException 异常
    */
void set_nodelay(int fd, bool yes);

/** 关闭指定的句柄
    */
void close_fd(int fd);

/***
    * 得到已经发送的文件总字节数
    */
long get_send_file_bytes();

/***
    * 得到已经发送的不包括文件在内的总字节数
```

```
    */
long get_send_buffer_bytes();

/***
    * 得到已经接收的数据总字节数
    */
long get_recv_buffer_bytes();
```

7.8. CEpoller

7.8.1. 头文件

```
#include <net/epoller.h>
```

7.8.2. 主要功能

1) 创建和销毁 Epoll
2) 毫秒级超时 Epoll 监控
3) 将对象加入到 Epoll 中
4) 将对象从 Epoll 中删除
5) 根据顺序号得到相应的发生 Epoll 事件的对象
6) 根据顺序号得到发生的 Epoll 事件

7.8.3. 成员函数

```
/***
    * Epoll 操作封装类
    */
class CEpoller
{
public:
    /***
        * 构造一个 Epoll 对象
        * 不会抛出任何异常
        */
    CEpoller();
    ~CEpoller();

    /***
        * 创建 Epoll，进行初始化
        * @epoll_size: 建议性 Epoll 大小
```

```
    * @exception: 如果出错，抛出 CSyscallException 异常
    */
    void create(uint32_t epoll_size);

    /***
    * 销毁已经创建的 Epoll
    * 不会抛出任何异常
    */
    void destroy();

    /***
    * 以超时方式等待 Epoll 有事件，如果指定的时间内无事件，则超时返回
    * @milliseconds: 最长等待的毫秒数，总是保证等待这个时长，即使被中断
    * @return: 如果在超时时间内，有事件，则返回有事件的对象个数，
    *          否则返回 0 表示已经超时了
    * @exception: 如果出错，抛出 CSyscallException 异常
    */
    int timed_wait(uint32_t milliseconds);

    /***
    * 将一个可 Epoll 的对象注册到 Epoll 监控中
    * @epollable: 指向可 Epoll 对象的指针
    * @events: 需要监控的 Epoll 事件，取值可以为: EPOLLIN 和 EPOLLOUT 等，
    *          具体请查看 Epoll 系统调用说明手册，通常不需要显示设置
    *          EPOLLERR 和 EPOLLHUP 两个事件，因为它们总是会被自动设置
    * @force: 是否强制以新增方式加入
    * @exception: 如果出错，抛出 CSyscallException 异常
    */
    void set_events(CEpollable* epollable, int events, bool force=false);

    /***
    * 将一个可 Epoll 对象从 Epoll 中删除
    * @epollable: 指向可 Epoll 对象的指针
    * @exception: 如果出错，抛出 CSyscallException 异常
    */
    void del_events(CEpollable* epollable);

    /***
    * 根据顺序号得到一个指向可 Epoll 对象的指针
    * @index: 顺序号，请注意 index 必须在 timed_wait 成功的返回值范围内
    * @return: 返回一个指向可 Epoll 对象的指针
    */
    CEpollable* get(uint32_t index) const;
```

```
    /***
      * 根据顺序号得到触发的 Epoll 事件
      * @index: 顺序号，请注意 index 必须在 timed_wait 成功的返回值范围内
      * @return: 返回发生的 Epoll 事件
      */
    uint32_t get_events(uint32_t index) const;
};
```

7.8.4. 示例代码

```
#include <net/epoller.h>
#include <net/listener.h>
#include <net/tcp_waiter.h>

class CMyWaiter: public net::CTcpWaiter
{
private:
    virtual bool handle_epoll_event(void* ptr, uint32_t events)
    {
        if (events & EPOLLIN)
        {
            // 有数据可读
        }
        else if (events & EPOLLOUT)
        {
            // 可以向对端发送数据
        }
        else
        {
            // 网络错误
            return false;
        }
    }
};

class CMyListener: public net::CListener
{
private:
    // 在 handle_epoll_event 中接受新的连接请求
    virtual bool handle_epoll_event(void* ptr, uint32_t events)
    {
        uint32_t peer_ip;
        uint16_t peer_port;
        net::CEpoller* epoller = (net::CEpoller*)ptr;
```

```
        // 接受连接请求
        int new_fd = this->accept(peer_ip, peer_port);

        // 创建一个 waiter 关联到新的 fd 上
        CMyWaiter* waiter =    new CMyWaiter;
        waiter->attach(new_fd);

        // 加入 Epoll 中监控
        epoller->set_events(waiter, EPOLLIN);
    }
};

// 启动一个监听
CMyListener listener;
net::CEpoller epoller;

try
{
    listener.listen("127.0.0.1", 8000);
    epoller.create(10000);
    epoller.set_events(&listener, EPOLLIN);

    for (;;)
    {
        int events_number = epoller.timed_wait(1000);
        if (0 == events_number)
        {
            // 超时继续处理
            continue;
        }
        else
        {
            for (int i=0; i<events_number; ++i)
            {
                int events = epoller.get_events(i);
                CEpollable* epollable = epoller.get(i);

                if (!epollable->handle_epoll_event(&epoller, events))
                {
                    // 需要关闭，和从 Epoll 中删除
                    epollable->close();
                    epoller.del_events(epollable);
                }
```

```
                }
            }
        }
}
catch (sys::CSyscallException& ex)
{
        printf("Exception found: %s.\n",
                sys::CSysUtil::get_error_message(ex.get_errcode()).c_str());
}
```

7.9. CListener

7.9.1. 头文件

```
#include <net/listener.h>
```

7.9.2. 主要功能

1) 启动在某端口上的监听
2) 接受连接请求
3) 得到监听的 IP 和端口号

7.9.3. 成员函数

```
/***
  * TCP 服务端监听者类
  * 用于启动在某端口上的监听和接受连接请求
  */
class CListener: public CEpollable
{
public:
        /** 构造一个 TCP 监控者 */
        CListener();

        /***
          * 启动在指定 IP 和端口上的监听
          * @ip: 监听 IP 地址
          * @port: 监听端口号
          * @enabled_address_zero: 是否允许在 0.0.0.0 上监听，安全起见，默认不允许
          * @exception: 如果发生错误，则抛出 CSyscallException 异常
          */
        void listen(const ip_address_t& ip, uint16_t port);
```

```
    void listen(const ipv4_node_t& ip_node, bool enabled_address_zero=false);
    void listen(const ipv6_node_t& ip_node, bool enabled_address_zero=false);

    /***
      * 接受连接请求
      * @peer_ip: 用来存储对端的 IP 地址
      * @peer_port: 用来存储对端端口号
      * @return: 新的 SOCKET 句柄
      * @exception: 如果发生错误，则抛出 CSyscallException 异常
      */
    int accept(ip_address_t& peer_ip, uint16_t& peer_port);

    /** 得到监听的 IP 地址 */
    const ip_address_t& get_listen_ip() const;

    /** 得到监听的端口号 */
    uint16_t get_listen_port() const;
};
```

7.9.4. 示例代码

```
#include "net/net_util.h"
#include "net/listener.h"
#include "sys/sys_util.h"
#include "net/tcp_waiter.h"
#include "util/string_util.h"

// 无参数时：
// 在 0.0.0.0:5174 上监听

// 一个参数时：
// 在 0.0.0.0 上监听
// argv[1]: 端口号

// 两个参数时：
// argv[1]: 监听的 IP 地址
// argv[2]: 监听的端口号
int main(int argc, char* argv[])
{
    uint16_t port;
    net::ip_address_t ip;
    net::CListener listener;

    if (1 == argc) // 无参数
```

```
{
        ip = (char*)NULL;
        port = 5174; // 我(5)要(1)测(7)试(4)
}
else if (2 == argc) // 一个参数
{
        ip = (char*)NULL;
        if (!util::CStringUtil::string2uint16(argv[2], port))
        {
                fprintf(stderr, "Invalid port: %s.\n", argv[2]);
                exit(1);
        }
}
else if (3 == argc) // 两个参数
{
        ip =    argv[1];
        if (!util::CStringUtil::string2uint16(argv[2], port))
        {
                fprintf(stderr, "Invalid port: %s.\n", argv[2]);
                exit(1);
        }
}

try
{
        // 开始监听，允许在 0.0.0.0 上监听
        listener.listen(ip, port, true);
        fprintf(stdout, "Listening at %s:%d.\n", ip.to_string().c_str(), port);

        uint16_t peer_port;
        net::ip_address_t peer_ip;

        while (true)
        {
                // 接受一个连接请求
                int newfd = listener.accept(peer_ip, peer_port);
                fprintf(stdout, "Accepted connect - %s:%d.\n"
                        , peer_ip.to_string().c_str(), peer_port);

                // 将新的请求关联到 CTcpWaiter 上
                net::CTcpWaiter waiter;
                waiter.attach(newfd);

                while (true)
```

```
                {
                    // 接收数据
                    char buffer[IO_BUFFER_MAX];
                    ssize_t retval = waiter.receive(buffer, sizeof(buffer)-1);
                    if (0 == retval)
                    {
                        // 对端关闭了连接
                        fprintf(stdout, "Connect closed by peer %s:%d.\n"
                                , peer_ip.to_string().c_str(), peer_port);
                        break;
                    }
                    else
                    {
                        // 在屏幕上打印接收到的数据
                        buffer[retval] = '\0';
                        fprintf(stdout, "[R] ==> %s.\n", buffer);
                    }
                }
            }
        }
        catch (sys::CSyscallException& ex)
        {
            // 监听或连接异常
            fprintf(stderr, "exception %s at %s:%d.\n"
                    , sys::CSysUtil::get_error_message(ex.get_errcode()).c_str()
                    , ex.get_filename(), ex.get_linenumber());
            exit(1);
        }

        return 0;
}
```

7.10. CListenManager

7.10.1. 头文件

```
#include <net/listen_manager.h>
```

7.10.2. 主要功能

创建多个 CListener，启动在多个端口上的监听。

7.10.3. 成员函数

```
/***
  * 监听管理者模板类
  */
template <class ListenClass>
class CListenManager
{
public:
    CListenManager();

    /***
      * 新增 IP 端口对，这里不做重复检查，所以调用前必须保证 IP 和端口不重复，另外
      * 应当在调用 create 之前调用 add，以设置好需要监听的地址和端口号，只要 IP+port
不重复即可
      * @ip: 监听 IP 地址
      * @port: 监听端口号
      * 不会抛出任何异常
      */
    void add(const char* ip, uint16_t port);

    /***
      * 启动在所有 IP 和端口对上的监听
      * @exception: 如果出错，则抛出 CSyscallException 异常
      */
    void create();

    /***
      * 销毁和关闭在所有端口上的监听
      * 不会抛出任何异常
      */
    void destroy();

    /** 得到监听者个数 */
    uint16_t get_listener_count() const;

    /** 得到指向监听者对象数组指针 */
    ListenClass* get_listener_array() const;
};
```

7.10.4. 示例代码

```
#include <net/listen_manager.h>
```

```
net::CListenManager<net::CListener> listen_manger;

// 在两个端口上监听
listen_manger.add("127.0.0.1", 80);
listen_manger.add("192.168.0.1", 8080);

try
{
    listen_manger.create();
}
catch (sys::CSyscallException& ex)
{
    // 监听失败
    printf("Create listen manager exception: %s.\n",
            sys::CSysUtil::get_error_message(ex.get_errcode()).c_str());
}
```

7.11. CTcpClient

7.11.1. 头文件

```
#include <net/tcp_client.h>
```

7.11.2. 主要功能

提供 TCP 客户端需要的功能：

1) 超时连接，可设置毫秒级的超时时长
2) 异步连接
3) 判断连接是否已经建立
4) 获取对端的 IP 和端口号
5) 接收数据到 Buffer
6) 发送 Buffer 中的数据
7) 直接将数据接收到文件
8) sendfile 函数的包装
9) 完整的接收指定长度的数据
10) 完整的发送指定长度的数据

7.11.3. 成员函数

```
/***
```

```
 * TCP 客户端类，提供客户端的各种功能
 */
class CTcpClient: public CEpollable
{
public:
    CTcpClient();
    ~CTcpClient();

    /** 得到对端端口号 */
    uint16_t get_peer_port() const;

    /** 得到对端 IP 地址 */
    const ip_address_t& get_peer_ip() const;

    /** 设置对端的 IP 和端口号 */
    void set_peer(const ipv4_node_t& ip_node);
    void set_peer(const ipv6_node_t& ip_node);

    /***
     * 设置对端 IP 地址
     * @ip: 对端的 IP 地址，可以为 IPV4，也可以为 IPV6 地址
     */
    void set_peer_ip(const ip_address_t& ip);

    /** 设置对端端口号 */
    void set_peer_port(uint16_t port);

    /** 设置连接的允许的超时毫秒数 */
    void set_connect_timeout_milliseconds(uint32_t milli_seconds);

    /***
     * 异步连接，不管是否能立即连接成功，都是立即返回
     * @return: 如果连接成功，则返回 true，否则如果仍在连接过程中，则返回 false
     * @exception: 连接错误，抛出 CSyscallException 异常
     */
    bool async_connect();

    /***
     * 超时连接
     * 如果不能立即连接成功，则等待由 set_connect_timeout_milliseconds 指定的时长，
     * 如果在这个时长内仍未连接成功，则立即返回
     * @exception: 连接出错或超时，抛出 CSyscallException 异常
     */
    void timed_connect();
```

```
    /** 接收 SOCKET 数据
     * @buffer: 接收缓冲区
     * @buffer_size: 接收缓冲区字节数
     * @return: 如果收到数据，则返回收到的字节数；如果对端关闭了连接，则返回 0；
     *          对于非阻塞连接，如果无数据可接收，则返回-1
     * @exception: 连接错误，抛出 CSyscallException 异常
     */
    ssize_t receive(char* buffer, size_t buffer_size);

    /** 发送 SOCKET 数据
     * @buffer: 发送缓冲区
     * @buffer_size: 需要发送的字节大小
     * @return: 如果发送成功，则返回实际发送的字节数；对于非阻塞的连接，如果不能继续发送，则返回-1
     * @exception: 如果发生网络错误，则抛出 CSyscallException 异常
     */
    ssize_t send(const char* buffer, size_t buffer_size);

    /** 完整接收，如果成功返回，则一定接收了指定字节数的数据
     * @buffer: 接收缓冲区
     * @buffer_size: 接收缓冲区字节大小，返回实际已经接收到的字节数(不管成功还是失败或异常)
     * @return: 如果成功，则返回 true，否则如果连接被对端关闭则返回 false
     * @exception: 如果发生网络错误，则抛出 CSyscallException，对于非阻塞连接，也可能抛出 CSyscallException 异常
     */
    bool full_receive(char* buffer, size_t& buffer_size);

    /** 完整发送，如果成功返回，则总是发送了指定字节数的数据
     * @buffer: 发送缓冲区
     * @buffer_size: 需要发送的字节数，返回实际已经发送了的字节数(不管成功还是失败或异常)
     * @return: 无返回值
     * @exception: 如果网络错误，则抛出 CSyscallException 异常；对于非阻塞连接，也可能抛出 CSyscallException 异常
     * @注意事项：保证不发送 0 字节的数据，也就是 buffer_size 必须大于 0
     */
    void full_send(const char* buffer, size_t& buffer_size);

    /** 发送文件，调用者必须保证 offset+count 不超过文件大小
     * @file_fd: 打开的文件句柄
     * @offset: 文件偏移位置，如果成功则返回新的偏移位置
     * @count: 需要发送的大小
```

```
     */
    ssize_t send_file(int file_fd, off_t *offset, size_t count);
    void full_send_file(int file_fd, off_t *offset, size_t& count);

    /** 采用内存映射的方式接收，并将数据存放文件，适合文件不是太大
     * @file_fd: 打开的文件句柄
     * @size: 需要写入文件的大小，返回实际已经接收到的字节数(不管成功还是失败或异常)
     * @offset: 写入文件的偏移值
     * @return: 如果连接被对端关闭，则返回 false 否则成功返回 true
     * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
     */
    bool full_map_tofile(int file_fd, size_t& size, size_t offset);

    /** 采用 write 调用的方式接收，并将数据存放文件，适合任意大小的文件，但是大文件会导致该调用长时间阻塞
     * @file_fd: 打开的文件句柄
     * @size: 需要写入文件的大小，返回实际已经接收到的字节数(不管成功还是失败或异常)
     * @offset: 写入文件的偏移值
     * @return: 如果连接被对端关闭，则返回 false 否则成功返回 true
     * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
     */
    bool full_write_tofile(int file_fd, size_t& size, size_t offset);

    /***
     * 一次性读一组数据，和系统调用 readv 的用法相同
     * @return: 返回实际读取到的字节数
     * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
     */
    ssize_t readv(const struct iovec *iov, int iovcnt);

    /***
     * 一次性写一组数据，和系统调用 writev 的用法相同
     * @return: 返回实际写入的字节数
     * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
     */
    ssize_t writev(const struct iovec *iov, int iovcnt);

    /** 判断连接是否已经建立
     * @return: 如果连接已经建立，则返回 true，否则返回 false
     */
    bool is_connect_established() const;
    bool is_connect_establishing() const;
```

```
    /** 设置为已经连接状态，仅用于异步连接，其它情况什么都不做
     * async_connect 可能返回正在连接中状态，当连接成功后，需要调用此函数来设置成已经连接状态，否则在调用 close 之前
     * 将一直处于正连接状态之中
     */
    void set_connected_state();

public: // override
    /** 关闭连接 */
    virtual void close();

private:
    /** 连接成功之后被调用 */
    virtual void after_connect();

    /***
     * connect 之前被调用，可以(也可不)重写该方法，
     * 以在 connect 前做一些工作，如修改需要连接的 IP 等
     */
    virtual bool before_connect();

    /** 连接失败，当连接失败时会被调用 */
    virtual void connect_failure();
};
```

7.11.4. 示例代码 1

```
#include "net/net_util.h"
#include "sys/sys_util.h"
#include "net/tcp_client.h"
#include "util/string_util.h"

// 需要两个参数：
// argv[1]: 连接 IP 地址
// argv[2]: 连接的端口号
int main(int argc, char* argv[])
{
    uint16_t port;
    net::ip_address_t ip;
    net::CTcpClient client;

    if (argc != 3)
    {
```

```
        fprintf(stderr, "usage: %s ip port\n"
                , sys::CSysUtil::get_program_short_name());
        exit(1);
    }

    ip =    argv[1];
    if (!util::CStringUtil::string2uint16(argv[2], port))
    {
        fprintf(stderr, "Invalid port: %s.\n", argv[2]);
        exit(1);
    }

    try
    {
        // 设置 IP 和端口
        client.set_peer_ip(ip);
        client.set_peer_port(port);

        // 执行连接
        client.timed_connect();
        fprintf(stdout, "Connected %s:%d success.\n"
                , ip.to_string().c_str(), port);

        while (true)
        {
            // 从标准输出读入数据，并发送给远端
            char line[LINE_MAX];
            fgets(line, sizeof(line)-1, stdin);

            client.send(line, strlen(line));
        }
    }
    catch (sys::CSyscallException& ex)
    {
        // 连接异常退出
        fprintf(stderr, "exception %s at %s:%d.\n"
                , sys::CSysUtil::get_error_message(ex.get_errcode()).c_str()
                , ex.get_filename(), ex.get_linenumber());
        exit(1);
    }

    return 0;
}
```

7.11.5. 示例代码 2

```
// 同步超时连接示例
#include <sys/sys_util.h>
#include <net/tcp_client.h>

net::CTcpClient client;

client.set_peer_ip("127.0.0.1");      // 设置需要连接到 127.0.0.1
client.set_peer_port(8000);           // 设置连接端口为 8000
client.set_connect_timeout_milliseconds(10000); // 设置连接超时时间为 10 秒

try
{
    client.timed_connect(); // 执行连接，如果连接失败，则会抛出异常 CSyscallException
    ssize_t bytes = send("Hello", sizeof("Hello")); // 向对端发送 Hello
    client.close(); // 关闭连接
}
catch (sys::CSyscallException& ex)
{
    // 连接出错了，关闭连接
    client.close();
    printf("Connect error: %s.\n"
            ,sys::CSysUtil::get_error_message(ex.get_errcode()).c_str());
}
```

7.11.6. 示例代码 3

```
// 演示如何使用异步连接
#include <sys/sys_util.h>
#include <net/tcp_client.h>

class CMyClient: public net::CTcpClient
{
private:
    virtual net::epoll_event_t handle_epoll_event(void* ptr, uint32_t events)
    {
        if (EPOLLOUT & events)
        {
            // 如果处于正在连接状态
            if (this->is_connect_establishing())
            {
                // 将状态设置为已经连接状态
```

```
                    // 只有异步连接时，才会有这一步
                    this->set_connected_state();
                }
                。。。。。
            }
            else
            {
                。。。。。。
            }
        }
};

CMyClient client;

client.set_peer_ip("127.0.0.1");     // 设置需要连接到 127.0.0.1
client.set_peer_port(8000);          // 设置连接端口为 8000

try
{
    if (client.async_connect())
    {
        // 已经连接成功
    }
    else
    {
        // 正在连接当中，将其加入 Epool 中，注意使用 EPOLLOUT 事件
        epoller.set_events(&client, EPOLLOUT);
    }
}
catch (sys::CSyscallException& ex)
{
    // 连接出错了，关闭连接
    client.close();
    printf("Connect error: %s.\n"
            ,sys::CSysUtil::get_error_message(ex.get_errcode()).c_str());
}
。。。。。。
int events_number = epoller.timed_wait(1000);
if (0 == events_number)
{
    // Epoll 超时
    continue;
}
for (int i=0; i<events_number; ++i)
```

```
{
    int events = epoller.get_events(i);
    CEpollable* epollable = epoller.get(i);
    net::epoll_event_t retval = epollable ->handle_epoll_event(&epoller, events);
    if (net::epoll_close == retval)
    {
        // 出错或需要关闭连接
        epollable->close();
        epoller.del_events(epollable);
    }
}
```

7.12. CTcpWaiter

7.12.1. 头文件

```
#include <net/tcp_waiter.h>
```

7.12.2. 主要功能

提供 TCP 服务端的功能（不包括监听部分，监听部分由 CListener 提供）：

1) 接收数据到 Buffer
2) 发送 Buffer 中的数据
3) 直接将数据接收到文件
4) sendfile 函数的包装
5) 完整的接收指定长度的数据
6) 完整的发送指定长度的数据

7.12.3. 成员函数

```
/***
  * TCP 服务端类，提供服务端的各种功能
  */
class CTcpWaiter: public CEpollable
{
public:
    CTcpWaiter();
    ~CTcpWaiter();

    /** 得到对端的 IP 地址，调用 attach 后者才可用 */
    const ip_address_t& get_peer_ip() const { return _peer_ip; }
```

```
    /** 得到对端的端口号，调用 attach 后者才可用 */
    port_t get_peer_port() const { return _peer_port; }

    /***
     * 关联到一个 fd
     * @fd: 被关闭的 fd，fd 为 CListener::accept 的返回值
     * @peer_ip: 对端的 IP 地址
     * @peer_port: 对端的端口号
     */
    void attach(int fd, const ip_address_t& peer_ip, port_t peer_port);

    /** 接收 SOCKET 数据
     * @buffer: 接收缓冲区
     * @buffer_size: 接收缓冲区字节数
     * @return: 如果收到数据，则返回收到的字节数；如果对端关闭了连接，则返回 0；
     *          对于非阻塞连接，如果无数据可接收，则返回-1
     * @exception: 连接错误，抛出 CSyscallException 异常
     */
    ssize_t receive(char* buffer, size_t buffer_size);

    /** 发送 SOCKET 数据
     * @buffer: 发送缓冲区
     * @buffer_size: 需要发送的字节大小
     * @return: 如果发送成功，则返回实际发送的字节数；对于非阻塞的连接，如果不能继续发送，则返回-1
     * @exception: 如果发生网络错误，则抛出 CSyscallException 异常
     * @注意事项：保证不发送 0 字节的数据，也就是 buffer_size 必须大于 0
     */
    ssize_t send(const char* buffer, size_t buffer_size);

    /** 完整接收，如果成功返回，则一定接收了指定字节数的数据
     * @buffer: 接收缓冲区
     * @buffer_size: 接收缓冲区字节大小，返回实际已经接收到的字节数(不管成功还是失败或异常)
     * @return: 如果成功，则返回 true，否则如果连接被对端关闭则返回 false
     * @exception: 如果发生网络错误，则抛出 CSyscallException，对于非阻塞连接，也可能抛出 CSyscallException 异常
     */
    bool full_receive(char* buffer, size_t& buffer_size);

    /** 完整发送，如果成功返回，则总是发送了指定字节数的数据
     * @buffer: 发送缓冲区
     * @buffer_size: 需要发送的字节数，返回实际已经接发送了的字节数(不管成功还是失败或异常)
```

```
    * @return: 无返回值
    * @exception: 如果网络错误，则抛出 CSyscallException 异常；对于非阻塞连接，也可能抛出 CSyscallException 异常
    * @注意事项：保证不发送 0 字节的数据，也就是 buffer_size 必须大于 0
    */
    void full_send(const char* buffer, size_t& buffer_size);

    /** 发送文件，调用者必须保证 offset+count 不超过文件大小
    * @file_fd: 打开的文件句柄
    * @offset: 文件偏移位置，如果成功则返回新的偏移位置
    * @count: 需要发送的大小
    */
    ssize_t send_file(int file_fd, off_t *offset, size_t count);
    void full_send_file(int file_fd, off_t *offset, size_t& count);

    /** 采用内存映射的方式接收，并将数据存放文件，适合文件不是太大
    * @file_fd: 打开的文件句柄
    * @size: 需要写入文件的大小，返回实际已经接收到的字节数(不管成功还是失败或异常)
    * @offset: 写入文件的偏移值
    * @return: 如果连接被对端关闭，则返回 false 否则成功返回 true
    * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
    */
    bool full_map_tofile(int file_fd, size_t& size, size_t offset);

    /** 采用 write 调用的方式接收，并将数据存放文件，适合任意大小的文件，但是大文件会导致该调用长时间阻塞
    * @file_fd: 打开的文件句柄
    * @size: 需要写入文件的大小，返回实际已经接收到的字节数(不管成功还是失败或异常)
    * @offset: 写入文件的偏移值
    * @return: 如果连接被对端关闭，则返回 false 否则成功返回 true
    * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
    */
    bool full_write_tofile(int file_fd, size_t& size, size_t offset);

    /***
    * 一次性读一组数据，和系统调用 readv 的用法相同
    * @return: 返回实际读取到的字节数
    * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
    */
    ssize_t readv(const struct iovec *iov, int iovcnt);

    /***
```

```
        * 一次性写一组数据，和系统调用 writev 的用法相同
        * @return: 返回实际写入的字节数
        * @exception: 如果发生系统调用错误，则抛出 CSyscallException 异常
        */
    ssize_t writev(const struct iovec *iov, int iovcnt);
};
```

7.12.4. 示例代码

```
#include <net/listener.h>
#include <net/tcp_waiter.h>

class CMyWaiter: public net::CTcpWaiter
{
private:
    virtual bool handle_epoll_event(void* ptr, uint32_t events)
    {
        // 根据 events 的类型，接收或发送数据
    }
};

class CMyListener: public net::CTcpListener
{
private:
    virtual bool handle_epoll_event(void* ptr, uint32_t events)
    {
        uint32_t peer_ip;
        int16_t peer_port;

        try
        {
            int new_fd = accept(peer_ip, peer_port);
            CMyWaiter* waiter = new CMyWaiter;
            waiter->attach(fd); // 关联到

            // 放入 Epoll 中
            epoller->set_events(waiter, EPOLLIN);
        }
        catch (sys::CSyscallException& ex)
        {
            // 异常处理
        }

        return true;
```

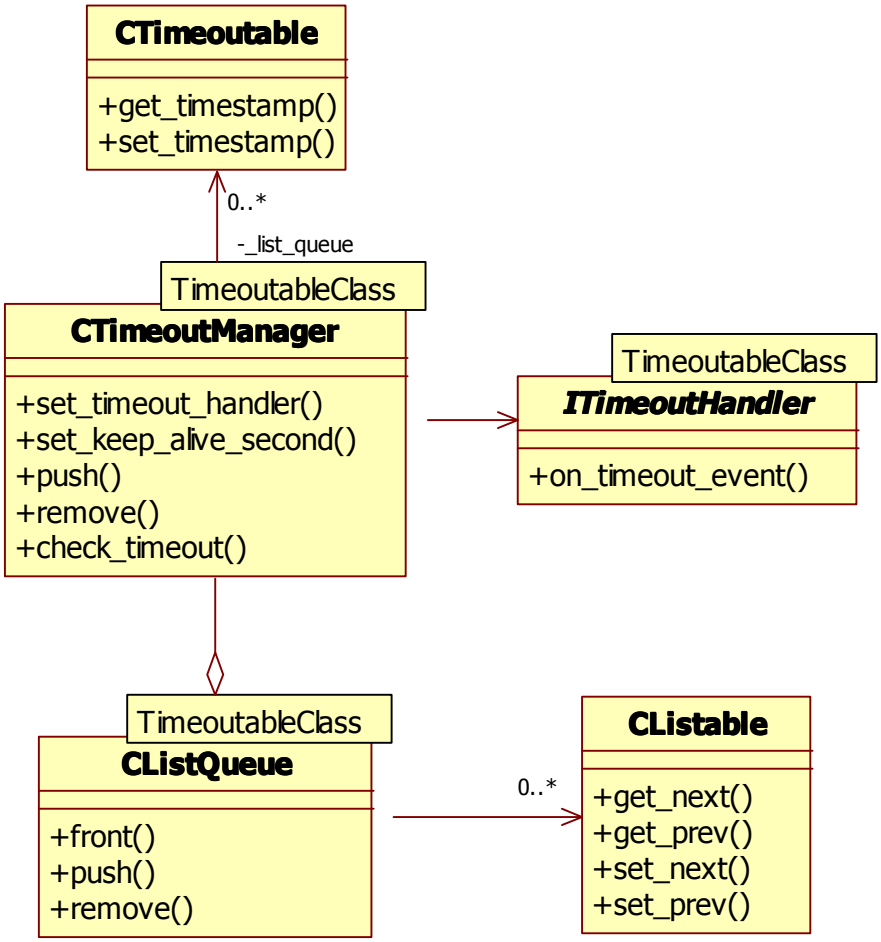
```
    }
};
```

7.13. CTimeoutable 和 CTimeoutManager

7.13.1. 头文件

CTimeoutable 类所在头文件：#include <net/timeoutable.h>
CTimeoutManager 类所在头文件：#include <net/timeout_manager.h>

7.13.2. 类图



要求超时对象，不但是 CTimeoutable 的子类型，同时还必须是 CListable 的子类型。CTimeoutable 用于时间戳，而 CListable 用于快速地将一个可超时对象从队列中删除和取出。

7.13.3. 主要功能

1) 统一可超时对象接口，需要超时管理的对象只需要继承自 CTimeoutable，即可使用获取和更新时间戳功能
2) 提供一个超时队列来管理超时对象
3) 提供 0 遍历高效的超时巡检方法
4) 通过注册回调接口，来异步通知哪对象发生了超时

7.13.4. 适用场景

适用于超时时长相同的类，并且按照时间顺序队列的。如果存在不同超时时长，则需要按不同时长分别创建不同的 CTimeoutManager 进行管理。

7.13.5. 成员函数

7.13.5.1. CTimeoutable

```
/***
  * 可超时对象的基类
  * 不应当直接使用此类，而应当总是继承方式
  */
class CTimeoutable
{
public:
    /*** 得到时间戳 */
    time_t get_timestamp() const

    /** 设置新的时间戳 */
    void set_timestamp(time_t timestamp);
};
```

7.13.5.2. ITimeoutHandler

```
/***
  * 超时处理器抽象接口
  * 发生超时时，回调它的 on_timeout_event 方法
  */
template <class TimeoutableClass>
class CALLBACK_INTERFACE ITimeoutHandler
{
```

```
public:
    /** 超时时被回调 */
    virtual void on_timeout_event(TimeoutableClass* timeoutable) = 0;
};
```

7.13.5.3. CTimeoutManager

```
/***
  * 超时管理模板类，维护一个超时队列，提供超时检测方法
  * TimeoutableClass 要求为 CTimeoutable 的子类型
  * 队列中的所有对象总是按时间先后顺序进入的，因此只需要从队首开始检测哪些超时了
  * 非线程安全类，因此通常一个线程一个 CTimeoutManager 实例，而
  * TimeoutableClass 类型的对象通常也不跨线程，
  * 这保证高效的前提，使得整个超时检测 0 查找
  */
template <class TimeoutableClass>
class CTimeoutManager
{
public:
    /*** 无参数构造函数 */
    CTimeoutManager();

    /** 设置超时秒数，也就是在这个时长内不算超时 */
    void set_timeout_seconds(uint32_t timeout_seconds);

    /** 设置超时处理器 */
    void set_timeout_handler(ITimeoutHandler<TimeoutableClass>* timeout_handler);

    /***
      * 将超时对象插入到超时队列尾
      * @timeoutable: 指向可超时的对象指针
      * @current_time: 当前时间
      */
    void push(TimeoutableClass* timeoutable, time_t current_time);

    /***
      * 将一个可超时对象从队列中删除
      * @timeoutable: 指向可超时的对象指针
      */
    void remove(TimeoutableClass* timeoutable);

    /***
      * 检测队列中哪些对象发生了超时
```

```
        * @current_time: 当前时间
        * 说明: 从队首开始循环遍历哪些对象发生了超时，如果超时发生，则
        * 回调 ITimeoutHandler 的 on_timeout_event 方法，
        * 直接检测到某对象未超时，便利结束
        */
    void check_timeout(time_t current_time);
};
```

7.13.6. 示例代码

```
#include <util/listable.h>
#include <util/list_queue.h>
#include <net/timeoutable.h>
#include <net/timeout_manager.h>

// 对连接进行超时管理
class CConnection: public util::CListable, public net::CTimeoutable
{
};

// 连接超时处理器
class CConnectionTimeoutHandler: public net::ITimeoutHandler
{
private:
    virtual void on_timeout_event(TimeoutableClass* timeoutable)
    {
        CConnection* connection = (CConnection*)timeoutable;
        // 连接超时，关闭连接
        connection->close();
    }
};

CConnectionTimeoutHandler timeout_handler;
net::CTimeoutManager<CConnection> timeout_manager;

// 设置超时时长为一分钟
timeout_manager.set_keep_alive_second(600);
timeout_manager.set_timeout_handler(&timeout_handler);


。。。。。。

CConnection* connection;
。。。。。。
// 需要超时控制的对象放入超时管理器中
```

```
timeout_manager.push(connection, time(NULL));
。。。。。。

// 定时超时检测
timeout_manager.check_timeout(time(NULL));
。。。。。。
```

CTimeoutManager 类是非线程安全的，因此通常一个线程一个 CTimeoutManager 实例，而 CConnection 也不跨线程使用，这保证高效的前提，使得整个超时检测 0 查找。

7.14. CUtil

7.14.1. 头文件

```
#include <net/util.h>
```

7.14.2. 主要功能

1) 判断一个字符串是否为 IPV4 地址
2) 判断一个字符串是否为广播地址
3) 得到所有网卡上的所有 IP 地址
4) 得到指定网卡名上的所有 IP 地址
5) 将整数类型的 IP 转换成字符串类型
6) 转换字符串类型的 IP 地址为整数类型
7) 毫秒级超时 Poll 实现

7.14.3. 成员函数

```
/***
  * 与网络相关的工具类
  */
class CUtil
{
public:
    typedef std::string TIP;   /** IP 地址 */
    typedef std::string TEth; /** 网卡名 */
    typedef std::vector<TIP> TIPArray; /** IP 地址数组 */
    typedef std::vector<std::pair<TEth, TIP> > TEthIPArray; /** 网卡名和 IP 对数组 */

    /** 判断是否为小字节序，如果是返回 true，否则返回 false */
    static bool is_little_endian();

    /***
```

```
    * 将源数据从主机字节序转换成网络字节序
    * @source: 需要转换的主机字节序源数据
    * @result: 存放转换后的网络字节序结果数据
    * @length: 需要转换的字节长度
    */
static void host2net(const void* source, void* result, size_t length);

/***
    * 将源数据从网络字节序转换成主机字节序
    * @source: 需要转换的网络字节序源数据
    * @result: 存放转换后的主机字节序结果数据
    * @length: 需要转换的字节长度
    */
static void net2host(const void* source, void* result, size_t length);

/***
    * 将源数据从主机字节序转换成网络字节序
    * @source: 需要转换的主机字节序源数据
    * @result: 存放转换后的网络字节序结果数据
    */
template <typename DataType>
static void host2net(const DataType& source, DataType& result);

/***
    * 将源数据从网络字节序转换成主机字节序
    * @source: 需要转换的网络字节序源数据
    * @result: 存放转换后的主机字节序结果数据
    */
template <typename DataType>
static void net2host(const DataType& source, DataType& result);

/** 判断给定的字符串是否为主机名或域名 */
static bool is_host_name(const char* str);

/** 判断给定的字符串是否为一个 IPV4 地址
    * @return: 如果给定的字符串是一个 IPV4 地址，则返回 true，否则返回 false
    */
static bool is_valid_ipv4(const char* str);

/***
    * 根据主机名得到一个 IP 地址
    * @hostname: 主机名
    * @ip_array: 存放 IP 的数组
    * @errinfo: 用来保存错误信息
```

```
    * @return: 如果成功返回 true，否则返回 false
    * @exception: 无异常抛出
    */
    static bool get_ip_address(const char* hostname, TIPArray& ip_array, std::string& errinfo);

    /** 得到网卡名和对应的 IP
    * @eth_ip_array: 用于保存所有获取到的 IP 地址
    * @exception: 如果发生错误，抛出 CSyscallException 异常
    */
    static void get_ethx_ip(TEthIPArray& eth_ip_array);

    /** 根据网卡名得到绑定在该网卡上的所有 IP 地址
    * @ethx: 网卡名，如：eth0，如果为 NULL，则表示所有网卡
    * @ip_array: 用于保存所有获取到的 IP 地址
    * @exception: 如果发生错误，抛出 CSyscallException 异常
    */
    static void get_ethx_ip(const char* ethx, TIPArray& ip_array);

    /** 根据整数类型的 IP，得到字符串类型的 IP 地址
    * @ip: 整数类型的 IP
    * @return: 字符串类型的 IP 地址
    */
    static std::string ipv4_tostring(uint32_t ip);
    static std::string ipv6_tostring(const uint32_t * ipv6);

    /** 将一个字符串转换成 IPV4 地址类型
    * @source: 需要转换的字符串
    * @ipv4: 存储转换后的 IPV4 地址
    * @return: 转换成功返回 true，否则返回 false
    */
    static bool string_toipv4(const char* source, uint32_t& ipv4);

    /** 将一个字符串转换成 IPV6 地址类型
    * @source: 需要转换的字符串
    * @ipv6: 存储转换后的 IPV6 地址，必须为连续的 16 字节，如: uint32_t[4]
    * @return: 转换成功返回 true，否则返回 false
    */
    static bool string_toipv6(const char* source, uint32_t* ipv6);

    /** 判断传入的字符串是否为接口名，如：eth0 等
    * @return: 如果 str 是接口名，则返回 true，否则返回 false
    */
    static bool is_ethx(const char* str);
```

```
    /** 判断传入的字符串是否为广播地址
      * @return: 如果 str 为广播地址，则返回 true，否则返回 false
      */
    static bool is_broadcast_address(const char* str);

    /** 超时 POLL 单个 fd 对象
      * @fd: 被 POLL 的单个 fd，注意不是 fd 数组
      * @events_requested: 请求监控的事件
      * @milliseconds: 阻塞的毫米数，总是保证等待这个时长，即使被中断
      * @events_returned: 用来保存返回的事件，如果为 NULL，则无事件返回，通过检测返回事件，可以明确是哪个事件发生了
      * @return: 超时返回 false，有事件返回 true
      * @exception: 网络错误，则抛出 CSyscallException 异常
      */
    static bool timed_poll(int fd, int events_requested, int milliseconds, int* events_returned=NULL);
};
```

7.14.4. 示例代码 1：字节序转换

```
#include <arpa/inet.h>
#include <net/net_util.h>

int main()
{
    // 转换 2 字节整数
    uint16_t b1;
    uint16_t a1 = 0x0103;

    net::CNetUtil::host2net<uint16_t>(a1, b1);
    printf("a1 = 0x%04x, b1=0x%04x\n", a1, b1);
    if (b1 == htons(a1)) /** 和系统库函数比较，以验证是否正确 */
        printf("host2net success\n");
    else
        printf("host2net failure\n");

    // 转换 4 字节整数
    printf("\n");
    uint32_t b2;
    uint32_t a2 = 0x01016070;

    net::CNetUtil::host2net<uint32_t>(a2, b2);
    printf("a2 = 0x%04x, b2=0x%04x\n", a2, b2);
    if (b2 == htonl(a2)) /** 和系统库函数比较，以验证是否正确 */
```

```
        printf("host2net success\n");
    else
        printf("host2net failure\n");

    // 按长度转换，应用到单字节字符串上，相当于反转字符串
    printf("\n");
    char str[] = "123456789";
    size_t length = strlen(str);
    char* dst = new char[length+1];
    util::delete_helper<char> dh(dst, true); // 自动调用 delete []dst
    net::CNetUtil::host2net(str, dst, length);
    dst[length] = '\0';
    printf("%s ==> %s\n",str, dst);

    return 0;
}
```

7.15. CEpollableQueue

7.15.1. 头文件

```
#include <net/epollable_queue.h>
```

7.15.2. 主要功能

1) 支持通过 Epoll 去检测队列中是否有数据
2) 其它队列的基本功能，但本身并不提供队列功能，而是借助模板参数指定的队列提供

7.15.3. 应用场景

适合用于一个线程需要将一或多个队列中的数据，通过一个或多个 SOCKET 异步地发送出去。

7.15.4. 成员函数

```
/** 可以放入 Epoll 监控的队列
  * GeneralQueueClass 为普通队列类名
  * 为线程安全类
  */
template <class GeneralQueueClass>
class CEpollableQueue: public CEpollable
```

```
{
public:
    /** 构造一个可 Epoll 的队列，注意只可监控读事件，也就是队列中是否有数据
     * @queue_max: 队列最大可容纳的元素个数
     * @exception: 如果出错，则抛出 CSyscallException 异常
     */
    CEpollableQueue(uint32_t queue_max);
    ~CEpollableQueue();

    /** 关闭队列 */
    virtual void close();

    /** 判断队列是否已满 */
    bool is_full() const;

    /** 判断队列是否为空 */
    bool is_empty() const;

    /***
     * 取队首元素
     * @elem: 存储取到的队首元素
     * @return: 如果队列为空，则返回 false，否则返回 true
     */
    bool front(DataType& elem) const;

    /***
     * 弹出队首元素
     * @elem: 存储弹出的队首元素
     * @return: 如果队列为空，则返回 false，否则取到元素并返回 true
     * @exception: 如果出错，则抛出 CSyscallException 异常
     */
    bool pop_front(DataType& elem);

    /***
     * 从队首依次弹出多个元素
     * @elem_array: 存储弹出的队首元素数组
     * @array_size: 输入和输出参数，存储实际弹出的元素个数
     * @exception: 如果出错，则抛出 CSyscallException 异常
     */
    void pop_front(DataType* elem_array, uint32_t& array_size)

    /***
     * 向队尾插入一元素
     * @elem: 待插入到队尾的元素
```

```
     * @return: 如果队列已经满，则返回 false，否则插入成功并返回 true
     * @exception: 如果出错，则抛出 CSyscallException 异常
     */
    bool push_back(DataType elem);

    /** 得到队列中当前存储的元素个数 */
    uint32_t size() const;
};
```

7.15.5. 示例代码

```
#include "sys/thread.h"
#include "net/epoller.h"
#include "sys/sys_util.h"
#include "util/array_queue.h"
#include "sys/datetime_util.h"
#include "net/epollable_queue.h"

#define QUEUE_SIZE    10000 // 队列大小
#define LOOP_NUMBER 10000 // 循环次数

// 用来读取队列中数据的线程，将数据从队列中读出，然后输出到标准输出
class CUTEpollableQueueThread: public sys::CThread
{
public:
    CUTEpollableQueueThread(net::CEpollableQueue<util::CArrayQueue<int> >* queue)
        :_queue(queue)
    {
        uint32_t epoll_size = 10;
        _epoller.create(epoll_size); // 创建 Epoll
        _epoller.set_events(queue, EPOLLIN); // 将队列放入 Epoll 中
    }

    ~CUTEpollableQueueThread()
    {
        _epoller.destroy();
    }

private:
    virtual void run()
    {
        while (!_stop)
        {
            try
```

```
            {
                // Epoll 检测队列中是否有数据
                if (0 == _epoller.timed_wait(1000))
                    continue; // 超时则继续等待

                int m = 0;
                if (_queue->pop_front(m)) // 弹出队首数据
                    fprintf(stdout, "<%s> pop %d from queue.\n"
                                    , sys::CDatetimeUtil::get_current_datetime().c_str(), m);
                else
                    fprintf(stderr, "<%s> get nothing from queue.\n"
                                    , sys::CDatetimeUtil::get_current_datetime().c_str());
            }
            catch (sys::CSyscallException& ex)
            {
                fprintf(stderr, "CUTEpollableQueueThread exception: %s at %s:%d.\n"
                    ,sys::CSysUtil::get_error_message(ex.get_errcode()).c_str()
                    ,ex.get_filename(), ex.get_linenumber());
            }
        }
    }

private:
    net::CEpoller _epoller;
    // 使用 CArrayQueue 作为队列容器
    net::CEpollableQueue<util::CArrayQueue<int> >* _queue;
};

int main()
{
    try
    {
        uint32_t queue_size = QUEUE_SIZE;
        net::CEpollableQueue<util::CArrayQueue<int> > queue(queue_size);
        CUTEpollableQueueThread* thread = new CUTEpollableQueueThread(&queue);

        thread->inc_refcount(); // 线程引用计数增一
        thread->start(); // 启动线程

        // 循环往队列中插入数据
        for (int i=1; i<LOOP_NUMBER; ++i)
        {
            if (queue.push_back(i))
                fprintf(stdout, "<%s> push %d to queue.\n"
```

```
                , sys::CDatetimeUtil::get_current_datetime().c_str(), i);
        else
            fprintf(stderr, "<%s> failed to push %d to queue.\n"
                , sys::CDatetimeUtil::get_current_datetime().c_str(), i);

        // 让线程 sleep 一秒钟
        sys::CSysUtil::millisleep(1000);
    }

    thread->stop(); // 停止线程
    thread->dec_refcount(); // 线程引用计数减一，这个必须在 thread->stop();调用之后
}
catch (sys::CSyscallException& ex)
{
    // 异常处理
    fprintf(stderr, "main exception: %s at %s:%d.\n"
        ,sys::CSysUtil::get_error_message(ex.get_errcode()).c_str()
        ,ex.get_filename(), ex.get_linenumber());
}

return 0;
}
```

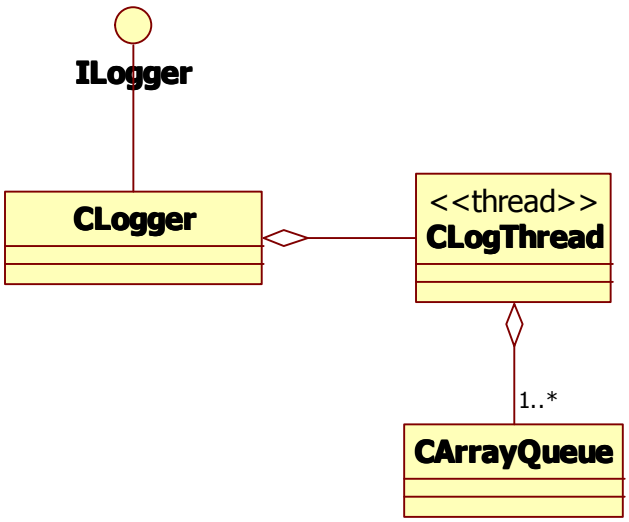
8. logger

8.1. 什么是 mooon logger?

logger 是 mooon 中 sys 库下的一个轻量的写本地日志工具，具备使用简单、结构简单、实现高效和线程安全（非多进程安全）等特点，主要功能如下：

1) 自动在日志行前添加日期、时间、线程号和日志级别；
2) 支持可变参数的文本日志和二进制日志；
3) 支持最大 64K 的一条日志（预先分配大小为 512 字节，这个值可设置）；
4) 支持 detail、debug、info、error、warn 和 fatal 多种日志级别；
5) 支持独立的 trace 日志，方便程序调试，独立的开关控制跟踪日志，允许关闭其它级别的日志，只输出跟踪日志；
6) 日志级别可动态调整；
7) 支持同时将日志打印到标准输出；
8) 可控制是否自动追加换行符，如果行末尾已经有换行符，则不会添加；
9) 可控制是否自动追加结尾点号，如果行末尾已经有结尾点号或换行符，则不会添加；
10) 支持滚动日志，允许设置备份个数和单个日志文件大小。

8.2. 类结构



目前有两种 ILogger 的实现，一种是基于 log4cxx 的，对应于 plugin_log4cxx，另一个是 **sys 库内置的 CLogger**，本文只介绍 CLogger 的实现。如果你喜欢，可以直接以类的方式使用 CLogger，而不以接口的方式使用 ILogger。

8.3. ILogger 接口

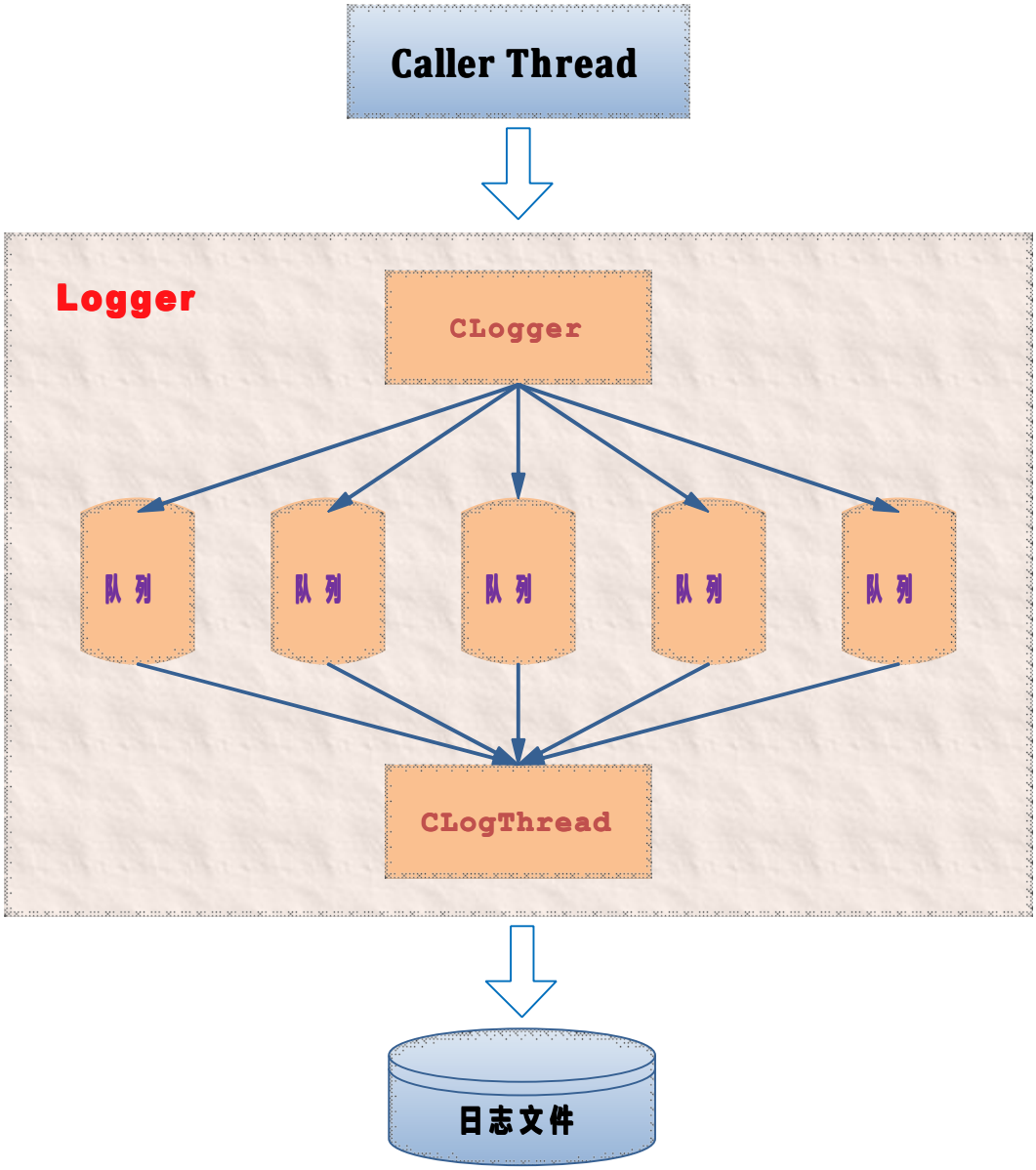
```
/**
 * 日志对外接口，接口文件隶属 sys 库
 */
class ILogger
{
public:
    /** 是否允许同时在标准输出上打印日志 */
    virtual void enable_screen(bool enabled) = 0;
    /** 是否允许跟踪日志，跟踪日志必须通过它来打开 */
    virtual void enable_trace_log(bool enabled) = 0;
    /** 是否自动在一行后添加结尾的点号，如果最后已经有点号，则不会再添加 */
    virtual void enable_auto_adddot(bool enabled) = 0;
    /** 是否自动添加换行符，如果已经有换行符，则不会再自动添加换行符 */
    virtual void enable_auto_newline(bool enabled) = 0;
    /** 设置日志级别，跟踪日志级别不能通过它来设置 */
    virtual void set_log_level(log_level_t log_level) = 0;
    /** 设置单个文件的最大建议大小 */
    virtual void set_single_filesize(uint32_t filesize) = 0;
    /** 设置日志文件备份个数，不包正在写的日志文件 */
    virtual void set_backup_number(uint16_t backup_number) = 0;
```

```
    /** 是否允许二进制日志 */
    virtual bool enabled_bin() = 0;
    /** 是否允许 Detail 级别日志 */
    virtual bool enabled_detail () = 0;
    /** 是否允许 Debug 级别日志 */
    virtual bool enabled_debug() = 0;
    /** 是否允许 Info 级别日志 */
    virtual bool enabled_info() = 0;
    /** 是否允许 Warn 级别日志 */
    virtual bool enabled_warn() = 0;
    /** 是否允许 Error 级别日志 */
    virtual bool enabled_error() = 0;
    /** 是否允许 Fatal 级别日志 */
    virtual bool enabled_fatal() = 0;
    /** 是否允许 Trace 级别日志 */
    virtual bool enabled_trace() = 0;

    virtual void log_detail (const char* format, ...) = 0;
    virtual void log_debug(const char* format, ...) = 0;
    virtual void log_info(const char* format, ...)   = 0;
    virtual void log_warn(const char* format, ...)   = 0;
    virtual void log_error(const char* format, ...) = 0;
    virtual void log_fatal(const char* format, ...) = 0;
    virtual void log_trace(const char* format, ...) = 0;
    virtual void log_bin(const char* log, uint16_t size) = 0;

    /** 写二进制日志 */
    virtual void bin_log(const char* log, uint16_t size) = 0;
};
```

8.4. 写日志宏

```
MYLOG_DETAIL(format, ...)
MYLOG_DEBUG(format, ...)
MYLOG_INFO(format, ...)
MYLOG_WARN(format, ...)
MYLOG_ERROR(format, ...)
MYLOG_FATAL(format, ...)
MYLOG_TRACE(format, ...)
MYLOG_BIN(log, size)
```

8.5. 工作原理



Logger 创建一个独立的线程，专门用来将队列中的日志写入磁盘。而日志线程可以配置多个队列，每个调用者线程的日志可以配置总是只存储到同一个队列，也可以不分队列存储，不分队列存放效率是最高的，但同一个线程输出的日志将是无时间顺序（秒级顺序错乱）的。

引入多队列，可以降低调用者线程和日志线程之间的碰撞，从而保证高效。

日志并不一定是按一条条写入磁盘的，日志线程总是一次性的借助 **writev** 系统调用将一个队列中当前所有的日志一次性写入磁盘，这样就减少了 I/O 次数。

但同时，Logger 仍旧保持了日志写入磁盘的及时性，在调用者线程和日志线程间设置了通知事件，如果所有队列均为空，则日志线程进入睡觉等待状态，任意调用线程将日志存入日志队列后，调用线程会检查日志线程是否处于睡觉等待状态，如果是则唤醒它，否则只是将日志放入队列。

8.6. 日志格式

日志格式为：[YYYY-MM-DD HH:MM:SS][8 位十六进制线程号][日志级别名称]日志内容，如：

[2010-09-01 21:10:20][0x8E92B4A1][INFO]测试日志.
[2010-09-01 21:10:23][0x6342B4A1][INFO]测试日志

8.7. 示例代码

```
#include <sys/logger.h> // 包含日志接口头文件

int main()
{
    sys::CLogger* logger = new sys::CLogger();

    try
    {
        // 创建日志器，日志存放目录为当前目录，日志文件名为 test.log
        logger->create(".", "test.log");
    }
    catch (CSyscallException& ex)
    {
        fprintf(stderr, "Failed to start log thread for %d at %s:%d"
            , ex.get_errcode(), ex.get_filename(), ex.get_linenumber());
        return 1;
    }

    // 下面这一句初始化全局的日志器，以方便使用日志宏，
    // 否则使用日志宏时，日志将打印到标准输出上。
    sys::ILogger* sys::g_logger = logger;

    // 允许同时在屏幕上输出日志
    logger->enable_screen(true);

    // 写日志，要使用日志宏，必须保证已经初始化 sys::g_logger
    MYLOG_DEBUG("%s", "测试日志");

    // 不再需要日志器了，比如进程退出之前
    logger->destroy(); // 日志器销毁之前会保证队列中的所有日志都写入文件
    delete logger;
    return 0;
}
```

9. 数据库连接池

9.1. 简述

mooon db wrapper 不是一个 DB，仅是对现有的 DB API 的封装，使得使用更为简单。项目地址：http://code.google.com/p/mooon，可使用 SVN 下载最新代码。开发和交流论坛：http://bbs.hadoopor.com/index.php?gid=67，可了解项目最新动态。

9.2. 相关头文件和库文件

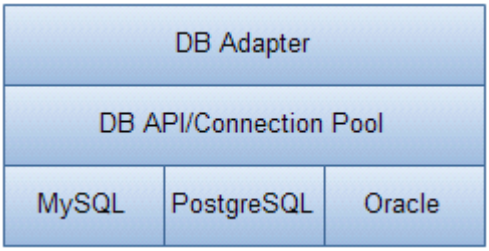
接口头文件在 sys 库中，MySQL 的实现头文件在 plugin_mysql 中，所以需要包含两个头文件和链接两个库文件，如下：

#include <sys/db.h>

#include <plugin/plugin_mysql/plugin_mysql.h>

两个库文件为：libsys.so 和 libxmysql.so。

9.3. 分层结构

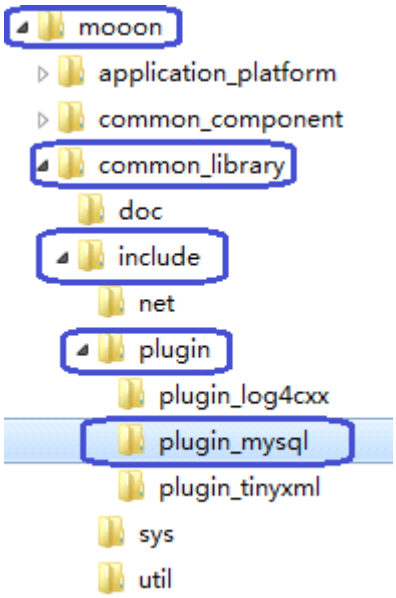


共分三层，最底层为各类数据库提供的 API，在它之上封装成 mooon db api，提供数据库连接池功能。mooon db API 本身与具体的数据库无关，通过不同的实现可支持不同的数据库，目前已经有 MySQL 的实现。

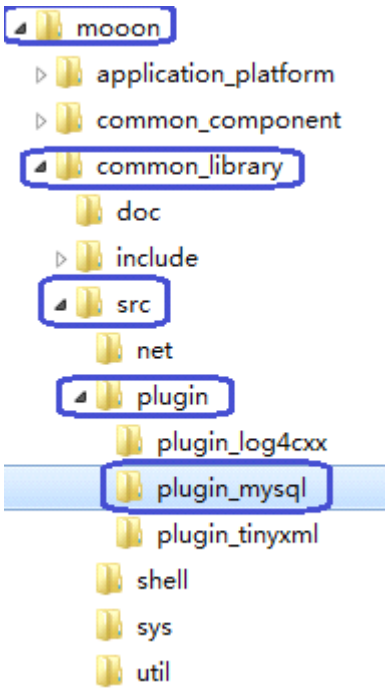
DB Adapter 是基于 DB API 的更高级抽象，内置数据库操作线程和队列，专门负责与数据库的读写交互，为上层应用提供一个异步回调的数据库操作，从而可解除应用和 DB 之间的一个强耦合。

9.4. 源码目录结构

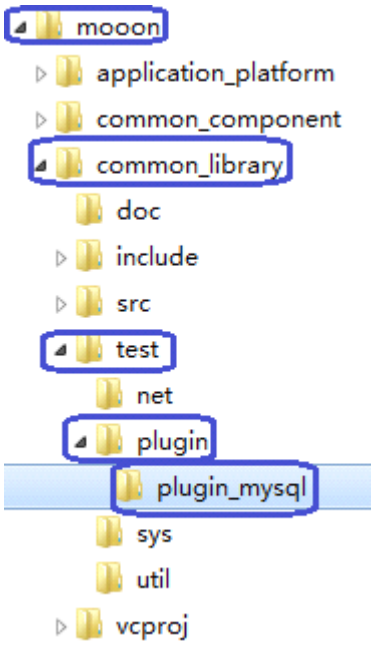
9.4.1. 头文件



9.4.2. CPP 文件



9.4.3. 测试代码



9.5. DB API

9.5.1. 异常类

数据库错误不能错误码的形式返回，而是采用异常的方式，可使得代码结构变得更为简洁。

9.5.1.1. CDBException

```
class CDBException
{
public:
    /***
      * 构造一个异常对象
      * 请注意不应当显示调用构造函数
      */
    CDBException(const char* sql, const char* error_message, int error_number=0, const char*
filename=__FILE__, int line_number=__LINE__);

    /** 返回执行出错的 SQL 语句，如果不是执行 SQL 语句，则仅返回一个字符串结尾符
*/
    const char* get_sql() const;
```

```
    /** 返回数据库的出错信息 */
    const char* get_error_message() const;

    /** 返回数据库的出错代码 */
    int get_error_number() const;

    /** 返回执行数据库操作时出错的文件名 */
    const char* get_filename() const;

    /** 返回执行数据库操作时出错的代码行 */
    int get_line_number() const;
};
```

9.5.2. 抽象接口

提供基础的数据库操作功能，而且也数据库无关，通过不同的实现，可支持不再的数据库。

9.5.2.1. IRecordrow

```
/***
  * 记录行接口，代表 Table 中的一行数据
  */
class IRecordrow
{
public:
    /***
      * 通过字段编号取得字段的值
      */
    virtual const char* get_field_value(uint16_t index) const = 0;
};
```

9.5.2.2. IRecordset

```
/***
  * 记录集接口
  */
class IRecordset
{
public:
    /***
      * 得到记录集的行数
```

```
        * 对于 MySQL，如果 query 时，参数 is_stored 为 false，则该函数不能返回正确的值，
        * 所以应当只有在 is_stored 为 true，才使用该函数
        */
    virtual size_t get_row_number() const = 0;

    /***
        * 得到字段个数
        */
    virtual uint16_t get_field_number() const = 0;

    /***
        * 判断记录集是否为空
        */
    virtual bool is_empty() const = 0;

    /***
        * 检索结果集的下一行
        * @return: 如果没有要检索的行返回 NULL，否则返回指向记录行的指针，这时必须
调用 release_recordrow，否则有内存泄漏
        */
    virtual IRecordrow* get_next_recordrow() const = 0;

    /***
        * 释放 get_next_recordrow 得到的记录行
        */
    virtual void free_recordrow(IRecordrow* recordrow) = 0;
};
```

9.5.2.3. IDBConnection

```
/***
  * 一般性的数据库连接接口
  * 使用引用计数管理生命周期
  */
class IDBConnection
{
public:
    /** 对引用计数值增一 */
    virtual void inc_refcount() = 0;

    /***
        * 对引用计数值减一
        * 如果减去之后，引用计数值为 0，则执行自删除
```

```
    * @return: 如果减去之后引用计数为 0，则返回 true，这个时候对象自身也被删除了
    */
    virtual bool dec_refcount() = 0;

    /** 是否允许自动提交 */
    virtual void enable_autocommit(bool enabled) = 0;

    /***
    * 用来判断数据库连接是否正建立着
    */
    virtual bool is_established() const = 0;

    /***
    * 数据库查询类操作，包括：select, show, describe, explain 和 check table 等
    * @is_stored: 是否将所有记录集拉到本地存储
    * @return: 如成功返回记录集的指针，这时必须调用 release_recordset，
    *          否则有内存泄漏
    * @exception: 如出错抛出 CDBException 异常
    */
    virtual IRecordset* query(bool is_stored, const char* format, ...) = 0;

    /***
    * 释放 query 得到的记录集
    */
    virtual void free_recordset(IRecordset* recordset) = 0;

    /***
    * 数据库 insert 和 update 更新操作
    * @return: 如成功返回受影响的记录个数
    * @exception: 如出错抛出 CDBException 异常
    */
    virtual size_t update(const char* format, ...) = 0;
};
```

9.5.2.4. IDBPoolConnection

```
/***
  * 用于数据库连接池的数据库连接接口
  */
class IDBPoolConnection
{
public:
    /** 是否允许自动提交 */
    virtual void enable_autocommit(bool enabled) = 0;
```

```
    /***
     * 用来判断数据库连接是否正建立着
     */
    virtual bool is_established() const = 0;

    /***
     * 数据库查询类操作，包括：select, show, describe, explain 和 check table 等
     * @is_stored: 是否将所有记录集拉到本地存储
     * @return: 如成功返回记录集的指针，这时必须调用 release_recordset
     *          ，否则有内存泄漏
     * @exception: 如出错抛出 CDBException 异常
     */
    virtual IRecordset* query(bool is_stored, const char* format, ...) = 0;

    /***
     * 释放 query 得到的记录集
     */
    virtual void free_recordset(IRecordset* recordset) = 0;

    /***
     * 数据库 insert 和 update 更新操作
     * @return: 如成功返回受影响的记录个数
     * @exception: 如出错抛出 CDBException 异常
     */
    virtual size_t update(const char* format, ...) = 0;
};
```

9.5.2.5. IDBConnectionPool

```
/***
 * 数据库连接池接口
 */
class IDBConnectionPool
{
public:
    /***
     * 得到全小写形式的数据库类型名，如：mysql 和 postgresql 等
     */
    virtual const char* get_type_name() const = 0;

    /***
     * 线程安全函数
```

```
    * 从数据库连接池中获取一个连接
    * @return: 如果当前无可用的连接，则返回 NULL，否则返回指向数据库连接的指针
    * @exception: 不会抛出任何异常
    */
    virtual IDBPoolConnection* get_connection() = 0;

    /***
    * 线程安全函数
    * 将已经获取的数据库连接放回到数据库连接池中
    * @exception: 不会抛出任何异常
    */
    virtual void put_connection(IDBPoolConnection* db_connection) = 0;

    /***
    * 创建连接池
    * @pool_size: 数据库连接池中的数据库连接个数
    * @db_ip: 需要连接的数据库 IP 地址
    * @db_port: 需要连接的数据库服务端口号
    * @db_name: 需要连接的数据库池
    * @db_user: 连接数据库用的用户名
    * @db_password: 连接数据库用的密码
    * @exception: 如出错抛出 CDBException 异常
    */
    virtual void create(uint16_t pool_size
                        ,const char* db_ip
                        ,uint16_t db_port
                        ,const char* db_name
                        ,const char* db_user
                        ,const char* db_password) = 0;

    /***
    * 销毁已经创建的数据库连接池
    */
    virtual void destroy() = 0;

    /***
    * 得到连接池中的连接个数
    */
    virtual uint16_t get_connection_number() const = 0;
};
```

9.5.2.6. IDBConnectionFactory

```
/***
```

```
 * 数据库连接工厂，用于创建 DBGeneralConnection 类型的连接
 */
class IDBConnectionFactory
{
public:
    /***
     * 创建 DBGeneralConnection 类型的连接
     * 线程安全
     * @db_ip: 需要连接的数据库 IP 地址
     * @db_port: 需要连接的数据库服务端口号
     * @db_name: 需要连接的数据库池
     * @db_user: 连接数据库用的用户名
     * @db_password: 连接数据库用的密码
     * @return: 返回一个指向一般性数据库连接的指针
     * @exception: 如出错抛出 CDBException 异常
     */
    virtual IDBConnection* create_general_connection(
                                        const char* db_ip
                                      , uint16_t db_port
                                      , const char* db_name
                                      , const char* db_user
                                      , const char* db_password) = 0;

    /***
     * 创建数据库连接池
     * @return: 返回指向数据库连接池的指针
     * @exception: 如出错抛出 CDBException 异常
     */
    virtual IDBConnectionPool* create_connection_pool() = 0;

    /***
     * 销毁数据库连接池
     * @db_connection_pool: 指向需要销毁的数据库连接池的指针，
     *                      函数返回后，db_connection_pool 总是被置为 NULL
     */
    virtual void destroy_connection_pool(IDBConnectionPool*& db_connection_pool) = 0;
};
```

9.5.3. 助手类

强烈建议：

能使用助手类的时候尽可能地使用它，可以带来不必要的麻烦，而且可以简化代码结构。

9.5.3.1. DBConnectionPoolHelper

```
/***
  * DBConnectionPool 助手类，用于自动销毁数据库连接池
  */
class DBConnectionPoolHelper
{
public:
    DBConnectionPoolHelper(IDBConnectionFactory* db_connection_factory,
                           IDBConnectionPool*& db_connection_pool);

    /** 析构函数，自动调用 destroy_connection_pool */
    ~DBConnectionPoolHelper();
};
```

9.5.3.2. DBPoolConnectionHelper

```
/***
  * DB 连接助手类，用于自动释放已经获取的 DB 连接
  */
class DBPoolConnectionHelper
{
public:
    DBPoolConnectionHelper(IDBConnectionPool* db_connection_pool
                           , IDBPoolConnection*& db_connection);
    /** 析构中将自动调用 put_connection */
    ~DBPoolConnectionHelper();
};
```

9.5.3.3. RecordsetHelper

```
/***
  * 记录集助手类，用于自动调用 free_recordset
  */
template <class DBConnectionClass>
class RecordsetHelper
{
public:
    RecordsetHelper(DBConnectionClass* db_connection, IRecordset* recordset);
    /** 析构中将自动调用 free_recordset */
    ~RecordsetHelper();
};
```

9.5.3.4. RecordrowHelper

```
/***
  * 记录行助手类，用于自动调用 free_recordrow
  */
class RecordrowHelper
{
public:
    RecordrowHelper(IRecordset* recordset, IRecordrow* recordrow);
    /** 析构中将自动调用 free_recordrow */
    ~RecordrowHelper();
};
```

9.6. 示例

9.6.1. 源代码

```
#include "sys/db.h"
#include "sys/ref_countable.h"
#include "plugin/plugin_mysql/plugin_mysql.h"

// 演示连接池 IDBPoolConnection 和一般连接 IDBConnection 的使用

// 往标准输出按行输出表中的所有记录
template <class DBConnectionClass>
void print_table(DBConnectionClass* db_connection, const char* sql)
{
    size_t row = 0; // 当前行数

    // 执行一条查询语句
    sys::IRecordset* recordset = db_connection->query(false, "%s", sql);
    uint16_t field_number = recordset->get_field_number();

    // 自动释放
    sys::RecordsetHelper<DBConnectionClass> recordset_helper(db_connection, recordset);

    for (;;)
    {
        // 取下一行记录
        sys::IRecordrow* recordrow = recordset->get_next_recordrow();
        if (NULL == recordrow) break;
```

```
            // 自动释放
            sys::RecordrowHelper recordrow_helper(recordset, recordrow);

            // 循环打印出所有字段值
            fprintf(stdout, "ROW[%04d] ==>\t", row++);
            for (uint16_t col=0; col<field_number; ++col)
            {
                const char* field_value = recordrow->get_field_value(col);
                fprintf(stdout, "%s\t", field_value);
            }
            fprintf(stdout, "\n");
        }
    }

    int main()
    {
        std::string sql = "SELECT * FROM test"; // 需要查询的 SQL 语句
        std::string db_ip = "127.0.0.1";
        std::string db_name = "test";
        std::string db_user = "root";
        std::string db_password = "";

        // 得到连接工厂
        sys::IDBConnectionFactory* db_connection_factory =
                                        plugin::get_mysql_connection_factory();

        try
        {
            // 创建一个一般连接
            sys::IDBConnection* general_connection = db_connection_factory->create_connection
                                            (db_ip.c_str()
                                            , 3306
                                            , db_name.c_str()
                                            , db_user.c_str()
                                            , db_password.c_str());

            // 使用引用计数帮助类，自动进行引用计数增一和减一
            sys::CRefCountHelper<sys::IDBConnection> rch(general_connection);

            // 创建一个连接池
            sys::IDBConnectionPool* db_connection_pool =
                                            db_connection_factory->create_connection_pool();
            // 用于自动销毁数据库连接池
            sys::DBConnectionPoolHelper dph(db_connection_factory, db_connection_pool);
```

```
        // 创建数据库连接池
        db_connection_pool->create(10
                                   , db_ip.c_str()
                                   , 3306
                                   , db_name.c_str()
                                   , db_user.c_str()
                                   , db_password.c_str());

        do // 这个循环无实际意义，仅为简化代码结构
        {
            // 从数据库连接池中取一个连接
            sys::IDBPoolConnection* pool_connection =
                                       db_connection_pool->get_connection();
            if (NULL == pool_connection)
            {
                fprintf(stderr, "Database pool is empty.\n");
                break;
            }

            // 自动释放
            sys::DBPoolConnectionHelper db_connection_helper(db_connection_pool
                                                             , pool_connection);

            printf("The following are from General Connection:\n");
            print_table<sys::IDBConnection>(general_connection, sql.c_str());

            printf("\n\nThe following are from Pool Connection:\n");
            print_table<sys::IDBPoolConnection>(pool_connection, sql.c_str());
        } while(false);
    }
    catch (sys::CDBException& ex)
    {
        fprintf(stderr, "Create database connection pool error: %s.\n", ex.get_error_message());
    }

    return 0;
}
```

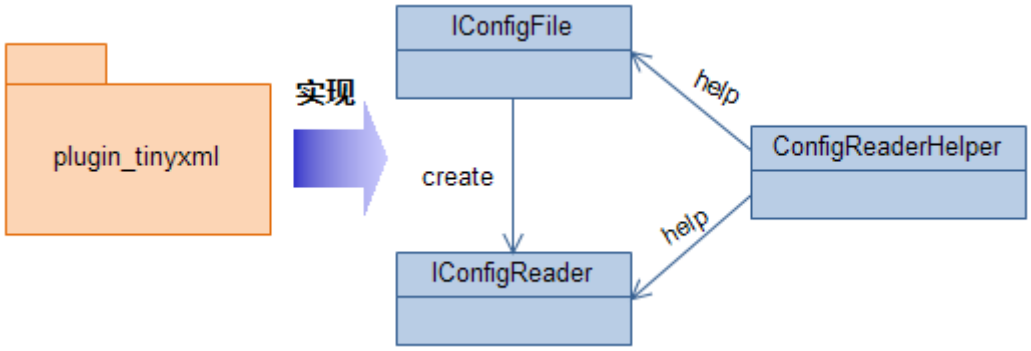
9.6.2. 编译运行

进入$mooon/common_library/test/plugin/plugin_mysql 目录（其中$mooon 为 mooon 源代码所在目录），运行 Make 编译，成功后执行 run.sh 即可运行测试代码，如：sh run.sh。

10. XML 配置文件读取

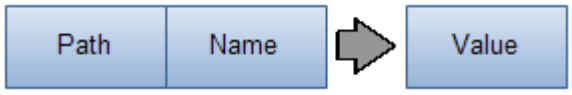
10.1. 介绍

提供使用简单，但功能强的基于 XML 的配置文件读取功能，可以支持不同的 XML 解析库，目前已经支持 tinyxml 的配置文件读取功能。



在上图中，plugin_tinyxml 是对配置读取接口的实现，而 ConfigReaderHelper 则为一帮助类，用于自动释放已经得到的 IConfigReader。

每个配置项，由配置路径 Path、配置名称 Name 和配置值 Value 三项组成，如下图所示：



通过输入一个 Path 和 Name，即可得到相应的配置值，允许存在多个相同的 Path，如：

```
<A>
    <B>
        <C />
    </B>
    <B>
        <C/>
    </B>
</A>
```

所有路径均以斜杠“/”打头，并以斜杠“/”分隔，如：/A/B/C。

小技巧：

可以通过 ie 等浏览器检测 XML 文件格式是否正确，方法是用浏览器打开 XML 文件。

10.2. 主要功能

1) 读取单个字符串类型的配置
2) 读取单个各种整数类型的配置
3) 以数组方式读取字符串类型的配置
4) 以数组方式读取各种整数类型的配置

5) 读取指定路径的子配置数组
6) 配置路径支持**正则表达式**匹配

10.3. 相关文件

头文件：
#include <sys/config_file.h> // 接口头文件
#include <plugin/plugin_tinyxml/plugin_tinyxml.h> // 基于 tinyxml 的实现头文件

名字空间：
接口定义在 sys 名字空间中，而基于 tinyxml 的实现，位于 plugin 名字空间。

库文件：
libsys.so 和 libxtinyxml.so

10.4. IConfigReader 接口

```
/**
  * 读取配置接口
  * <A>
  *     <B>
  *         <C age="32"/>
  *     </B>
  * </A>
  *
  * path: /A/B/C
  * name: age
  * value: 32
  */
class IConfigReader
{
public:
    /** 判断指定路径是否存在 */
    virtual bool path_exist(const std::string& path) = 0;

    /** 判断指定路径下的指定配置名是否存在 */
    virtual bool name_exist(const std::string& path, const std::string& name) = 0;

    /***
      * 得到单个字符串类型的配置值
      * @path: 配置路径
      * @name: 配置名称
      * @value: 用于存储得到的配置值
```

```
     * @return: 如果成功得到字符串类型的配置值，则返回 true，否则返回 false
     */
    virtual bool get_string_value(const std::string& path, const std::string& name, std::string& value) = 0;
    virtual bool get_int16_value(const std::string& path, const std::string& name, int16_t& value) = 0;
    virtual bool get_int32_value(const std::string& path, const std::string& name, int32_t& value) = 0;
    virtual bool get_int64_value(const std::string& path, const std::string& name, int64_t& value) = 0;
    virtual bool get_uint16_value(const std::string& path, const std::string& name, uint16_t& value) = 0;
    virtual bool get_uint32_value(const std::string& path, const std::string& name, uint32_t& value) = 0;
    virtual bool get_uint64_value(const std::string& path, const std::string& name, uint64_t& value) = 0;

    /***
     * 得到多个字符串类型的配置值
     * @path: 配置路径
     * @name: 配置名称
     * @value: 用于存储得到的配置值的数组
     * @return: 如果成功得到字符串类型的配置值，则返回 true，否则返回 false
     */
    virtual bool get_string_values(const std::string& path, const std::string& name, std::vector<std::string>& values) = 0;
    virtual bool get_int16_values(const std::string& path, const std::string& name, std::vector<int16_t>& values) = 0;
    virtual bool get_int32_values(const std::string& path, const std::string& name, std::vector<int32_t>& values) = 0;
    virtual bool get_int64_values(const std::string& path, const std::string& name, std::vector<int64_t>& values) = 0;
    virtual bool get_uint16_values(const std::string& path, const std::string& name, std::vector<uint16_t>& values) = 0;
    virtual bool get_uint32_values(const std::string& path, const std::string& name, std::vector<uint32_t>& values) = 0;
    virtual bool get_uint64_values(const std::string& path, const std::string& name, std::vector<uint64_t>& values) = 0;

    /***
     * 得到子配置读取器
     * @path: 配置路径
     * @sub_config_array: 用于存储子配置读取器的数组
     * @return: 如果成功得到子配置读取器，则返回 true，否则返回 false
```

```
        */
    virtual bool get_sub_config(const std::string& path, std::vector<IConfigReader*>&
sub_config_array) = 0;
};
```

10.5. IConfigFile

```
class IConfigFile
{
public:
    /** 打开配置文件
      * @return: 如果打开成功返回 true，否则返回 false
      * @exception: 无异常抛出
      */
    virtual bool open(const std::string& xmlfile) = 0;

    /** 关闭打开的配置文件 */
    virtual void close() = 0;

    /** 获取一个 IConfigReader 对象，请注意 get_config_reader
      * 和 release_config_reader 的调用必须成对，否则会内存泄漏
      * @return: 如果成功返回 IConfigReader 类型的指针，否则返回 NULL
      */
    virtual IConfigReader* get_config_reader() = 0;
    virtual void free_config_reader(IConfigReader* config_reader) = 0;

    /** 得到发生错误的行号 */
    virtual int get_error_row() const = 0;

    /** 得到发生错误的列号 */
    virtual int get_error_col() const = 0;

    /** 得到出错信息 */
    virtual std::string get_error_message() const = 0;
};
```

10.6. ConfigReaderHelper

```
/***
  * ConfigReader 帮助类，用于自动释放已经获取的 ConfigReader
  */
class ConfigReaderHelper
{
```

```
public:
    /** 构造一个 ConfigReader 帮助类对象 */
    ConfigReaderHelper(IConfigFile* config_file, IConfigReader*& config_reader);
    /** 用于自动释放已经得到的 ConfigReader */
    ~ConfigReaderHelper();
};
```

10.7. 全局配置变量

extern IConfigFile* **g_config;**

一个本局配置的声明，而非定义，是否定义由调用者决定。在这里声明一个全局配置，是为了方便读取配置。

10.8. 示例代码

假设有如下一配置文件：

```
<?xml version="1.0" encoding="gb2312"?>
<JWS>

  <default_listen ip="eth0" port="8000" />
  <thread number="1" waiter_number="1000"  keep_alive_second="60" />

  <virtual_host domain_name="www.hadoopor.com">
    <listen ip="eth0" port="8000" />
    <document root="/" index="index.htm" />
  </virtual_host>

  <virtual_host domain_name="bbs.hadoopor.com">
    <listen ip="eth0" port="8000" />
    <document root="/" index="index.htm" />
  </virtual_host>

</JWS>
```

```
// 对在 sys/config_file.h 文件中声明的全局 g_config 进行定义
sys::IConfigFile* sys::g_config = NULL;

// 实例化配置文件接口
sys::g_config = plugin::create_config_file();

// 得到根配置读接口
sys::IConfigReader* config_reader = g_config->get_config_reader();
```

```
// 读取 default_listen 的配置
std::string ip; // 监听 IP 地址
if (!config_reader->get_string_value("/JWS/listen", "ip", ip))
{
    MYLOG_WARN("Not configured default host at \"/JWS/listen:ip\".\n");
}

uint16_t port; // 监听端口号
if (!config_reader->get_uint16_value("/JWS/listen", "port", port))
{
    // 没有得到端口号
    MYLOG_WARN("Default listen port not configured at \"/JWS/listen:port\".\n");
}

// 读取 threadnumber 的值
uint16_t _thread_number;
if (!config_reader->get_uint16_value("/JWS/thread", "number", thread_number))
{
    MYLOG_ERROR("Can not get thread number from \"/JWS/thread:number\".\n");
}

std::vector<sys::IConfigReader*> sub_config_array;
// 获取所有的子配置
if (!config_reader->get_sub_config("/JWS/virtual_host", sub_config_array))
{
    MYLOG_WARN("Not found virtual host at \"/JWS/virtual_host\".\n");
}

for (std::vector<sys::IConfigReader*>::size_type i=0; i<sub_config_array.size(); ++i)
{
    // 使用 ConfigReaderHelper 帮助类
    // 如果不使用帮助类，则需要显示的调用 release_config_reader，如：
    // g_config->free_config_reader(sub_config_array[i]);
    sys::ConfigReaderHelper crh(g_config, sub_config_array[i]);

    std::string domain_name;
    if (!sub_config_array[i]->get_string_value(
                        "/virtual_host" // 这个是相对于"/JWS/virtual_host"的路径
                      , "domain_name"
                      , domain_name))
    {
        MYLOG_ERROR("Domain name not found at \"/virtual_host:domain_name\".\n");
    }
```

```
        std::string document_root;
        if (!sub_config_array[i]->get_string_value(
                                "/virtual_host/document", // 这是相对路径
                                "root"
                                , document_root))
        {
            MYLOG_WARN("Can not get document root .\n");
        }

        if (!sub_config_array[i]->get_string_value(
                                "/virtual_host/document" // 这是相对路径
                               , "index",
                                directory_index))
        {
            MYLOG_WARN("Can not get directory index.\n");
        }
}
```