C++ 基础库 CO 参考文档

Alvin 2020/07/07
idealvin@qq.com

CO 是一个优雅、高效的 C++ 基础库，支持 Linux, Windows 与 Mac 平台。本文档将介绍 CO 的功能组件及使用方法。

1. 概览

CO 追求极简、高效，不依赖于 boost 等三方库，仅使用了少量的 C++11 特性。

- CO 实现的功能组件：
  - 基本定义(def)
  - 原子操作(atomic)
  - 快速伪随机数生成器(ramdom)
  - LruMap
  - 基本类型快速转字符串(fast)
  - 高效字符流(fastream)
  - 高效字符串(fastring)
  - 字符串操作(str)
  - 命令行参数与配置文件解析库(flag)
  - 高效流式日志库(log)
  - 单元测试框架(unitest)
  - 时间库(time)
  - 线程库(thread)
  - 协程库(co)
  - 高效 json 库(json)
  - 高性能 json rpc 框架(rpc)
  - hash 库(hash)
  - path 库(path)
  - 文件系统操作(fs)
  - 系统操作(os)
- CO 使用的 C++11 特性:
  - auto
  - std::move
  - std::bind
  - std::function
  - std::unique_ptr
  - std::unordered_map
  - std::unordered_set
  - variadic templates

2. 基本定义(def)

include: co/def.h.

2.1 定长整数类型

```
 int8   int16   int32   int64
uint8  uint16  uint32  uint64
```

这些类型在不同平台的长度是一致的，不存在可移植性问题。**Google Code Style** 建议除了 int，不要使用 short, long, long long 等内置整数类型。

def.h 还定义了上述整数类型的最大、最小值：

```
MAX_UINT8  MAX_UINT16  MAX_UINT32  MAX_UINT64
MAX_INT8   MAX_INT16   MAX_INT32   MAX_INT64
MIN_INT8   MIN_INT16   MIN_INT32   MIN_INT64
```

2.2 读写 1、2、4、8 字节

def.h 定义了如下的宏，用于读写 1、2、4、8 字节的数据(注意边界对齐)：

```
load8  load16  load32  load64
save8  save16  save32  save64
```

- 代码示例

```
uint64 v;                 // 8 字节
save32(&v, 7);            // v 的前 4 个字节设为 7
uint16 x = load16(&v);    // 读取 v 的前 2 个字节
```

2.3 DISALLOW_COPY_AND_ASSIGN

这个宏用于禁止 C++ 类中的拷贝构造函数与赋值操作：

- 代码示例

```
class T {
  public:
    T();
    DISALLOW_COPY_AND_ASSIGN(T);
};
```

2.4 force_cast 强制类型转换

force_cast 是对 C 风格强制类型转换的包装：

- 代码示例

```
char c = force_cast<char>(97); // char c = (char) 97;
```

2.5 __forceinline 与 __thread

__forceinline 是 VS 中的关键字，Linux 等平台用下面的宏模拟：

```
#define __forceinline __attribute__((always_inline))
```

__thread 是 gcc 中的关键字，用于支持 TLS，Windows 上用下面的宏模拟：

```
#define __thread __declspec(thread)
```

- 代码示例

```
// 获取当前线程的 id
__forceinline unsigned int gettid() {
    static __thread unsigned int id = 0;
    if (id != 0) return id;
    return id = __gettid();
}
```

2.6 unlikely

unlikely 宏用于分支选择优化(仅支持 gcc、clang)：

```
// 与 if (v == -1) 逻辑上等价，但提示编译器 v == -1 的可能性较小
if (unlikey(v == -1)) {
    cout << "v == -1" << endl;
}
```

3. 原子操作(atomic)

include: co/atomic.h.

atomic 库定义了如下的原子操作：

```
atomic_inc        atomic_dec        atomic_add        atomic_sub
atomic_fetch_inc  atomic_fetch_dec  atomic_fetch_add  atomic_fetch_sub

atomic_or         atomic_and        atomic_xor
atomic_fetch_or   atomic_fetch_and  atomic_fetch_xor

atomic_swap    atomic_compare_swap
atomic_get     atomic_set    atomic_reset
```

上述原子操作适用于 1，2，4，8 字节长度的数据类型。inc，dec，add，sub，or，and，xor 各有一个 fetch 版，区别在于，fetch 版本返回原子操作之前的值，非 fetch 版本返回原子操作之后的值。

- 代码示例

```
bool b = false;
int i = 0;
uint64 u = 0;
void* p = 0;

atomic_inc(&i);                 // return ++i;
atomic_dec(&i);                 // return --i;
atomic_add(&i, 3);              // return i += 3;
atomic_sub(&i, 3);              // return i -= 3;
atomic_fetch_inc(&u);           // return u++;
```

```
atomic_or(&i, 8);               // return i |= 8;
atomic_and(&i, 7);              // return i &= 7;
atomic_xor(&i, 7);              // return i ^= 7;
atomic_fetch_xor(&i, 7);        // v = i; i ^= 7; return v;

atomic_swap(&b, true);          // v = b; b = true; return v;
atomic_compare_swap(&i, 0, 1);  // v = i; if (i == 0) i = 1; return v;

atomic_get(&u);                 // return u;
atomic_set(&u, 7);              // u = 7;
atomic_reset(&i);               // i = 0;

// atomic operations on pointers
atomic_set(&p, 0);
atomic_swap(&p, 8);
atomic_compare_swap(&p, 0, 8);
```

4. 随机数生成器(random)

include: co/random.h.

Random 是一个速度极快的伪随机数生成器，可以连续无重复的生成 1 ~ 2G-2 之间的整数。**leveldb** 用到了这种算法，本库选用了与 leveldb 不同的常数 **16385**，计算速度更快。算法的数学原理可以参考一种快速的随机数生成算法一文。

- 代码示例

```
Random r(7);      // 7 是种子数, 不带参数时, 默认为 1
int n = r.next(); // !! 非线程安全
```

5. LruMap

include: co/lru_map.h.

LRU 是一种常用的缓存策略，当缓存达到容量上限时，优先替换掉最近最少使用的数据。**LruMap** 基于 std::list、std::unordered_map 实现，内部元素是无序的。

- 代码示例

```
LruMap<int, int> m(128);         // capacity: 128
m.insert(1, 23);                 // m.size() > 128 时, 删除内部 list 尾部的元素(最近最少使用)
                                 // !! key 已经存在时, 则不会插入新元素
auto it = m.find(1);             // 找到时, 将 1 放到内部 list 首部
if (it != m.end()) m.erase(it);  // erase by iterator
m.erase(it->first);              // erase by key
```

6. 基本类型快速转字符串(fast)

include: co/fast.h.

fast 库提供了如下的函数：

```
u32toh  u64toh  u32toa  u64toa  i32toa  i64toa  dtoa
```

xtoh 系列函数将整数类型转换为十六进制字符串，内部用一个 table 缓存前 256 个数对应的 16 进制字符串(2个字节)，不同平台测试结果比 snprintf 快 10~25 倍左右。

xtoa 系列函数将整数类型转换为十进制 ascii 字符串，内部用一个 table 缓存前 10000 个数对应的 10 进制字符串(4个字节)，不同平台测试结果比 snprintf 快 10~25 倍左右。

dtoa 采用了 **Milo Yip** 的实现，详情见 **miloyip/dtoa-benchmark**。早期基于 **LruMap** 的实现，已弃用。

- 代码示例

```
char buf[32];
int len = fast::u32toh(255, buf); // "0xff", 返回长度 4
int len = fast::i32toa(-99, buf); // "-99", 返回长度 3
int len = fast::dtoa(0.123, buf); // "0.123", 返回长度 5
```

fast 库还提供一个 **fast::stream** 类，支持流式输出、append 等基本操作，是下述的 **fastream** 与 **fastring** 的基类。

7. 高效字符流(fastream)

include: **co/fastream.h**.

C++ 标准库中的 **std::ostringstream** 性能较差，比 snprintf 慢好几倍。**fastream** 继承自 **fast::stream** 类，支持流式输出与二进制 append 操作。其中流式输出，在不同平台比 snprintf 快 10~30 倍左右。

- 代码示例

```
fastream fs(1024);          // 预分配 1k 内存
fs << "hello world" << 23;  // 流模式

int i = 23;
char buf[8];
fs.append(buf, 8);      // 追加 8 字节
fs.append(&i, 4);       // 追加 4 字节
fs.append(i);           // 追加 4 字节, 与 fs.append(&i, 4) 等价
fs.append((int16) 23);  // 追加 2 字节
fs.append('c');         // 追加单个字符
fs.append(100, 'c');    // 追加 100 个 'c'
fs.append('c', 100);    // 追加 100 个 'c'

fs.c_str();             // 返回 C 风格字符串
fs.str();               // 返回 C++ 字符串, 内存拷贝
fs.data();              // 返回数据指针
fs.size();              // 返回数据长度
fs.capacity();          // 容量

fs.reserve(4096);       // 预分配至少 4k 内存
fs.resize(32);          // size -> 32, buffer 中内容不变
fs.clear();             // size -> 0
fs.swap(fastream());    // 交换
```

- 注意事项

fastream 出于性能上的考虑，在 **append** 操作时不进行安全检查，如下面的代码是不安全的：

```
fastream f;
f.append("hello");
f.append(f.c_str() + 1); // 不安全, 内部不考虑内存重叠的情况,
```

8. 高效字符串(fastring)

include: co/fastring.h.

fastring 是 co 中的字符串类型，与 **fastream** 一样，继承于 **fast::stream**，因此除了基本的字符串操作，它同样也支持流式输出操作:

```
fastring s;
s << "hello world " << 1234567;
```

早期的 fastring 实现中使用了引用计数，导致 fastring 的复制行为与 **std::string** 不同，容易造成混淆。为了更好的取代 std::string，重构的版本中移除了引用计数。

fastring 几乎支持 fastream 的所有操作，但有一点与 fastream 不一样，fastring 在 **append** 时会进行安全检查:

```
fastring s("hello");
fastream f;
f.append("hello");

// 复制的数据与自身数据重叠
s.append(s.c_str() + 1); // 安全，内部检测到内存重叠，进行特殊的处理
f.append(f.c_str() + 1); // 不安全，内部不检测内存重叠的情况，不能保证得到正确的结果
```

- 代码示例

```
fastring s;                 // 空字符串，无内存分配
fastring s(32);             // 空字符串，预分配内存(容量为32)
fastring s("hello");        // 非空字符串
fastring s(88, 'x');        // 初始化 s 为 88 个 'x'
fastring s('x', 88);        // 初始化 s 为 88 个 'x'
fastring t = s;             // 通过内存拷贝创建一个新的字符串

s << "hello " << 23;        // 流式输出
s += "xx";                  // 追加
s.append("xx");             // 追加  <==>  s += "xx";
s.swap(fastring());         // 交换

s + "xxx";                  // +
s > "xxx";                  // >
s < "zzz"                   // <
s <= "zz"                   // <=
s >= "zz"                   // >=

s.find('c');                // 查找字符
s.find("xx", 3);            // 从 pos 3 开始查找子串
s.rfind('c');               // 反向查找字符
s.rfind("xx");              // 反向查找字符串
s.find_first_of("xy");      // 查找第一次出现的 "xy" 中的字符
s.find_first_not_of("xy"); // 查找第一次出现的非 "xy" 中的字符
s.find_last_of("xy");       // 查找最后一次出现的 "xy" 中的字符
s.find_last_not_of("xy");  // 查找最后一次出现的非 "xy" 中的字符

s.starts_with('x');         // s 是否以 'x' 开头
s.starts_with("xx");        // s 是否以 "xx" 开头
s.ends_with('x');           // s 是否以 'x' 结尾
s.ends_with("xx");          // s 是否以 "xx" 结尾

s.replace("xxx", "yy");     // 将 s 中的 "xxx" 替换为 "yy"
s.replace("xxx", "yy", 3); // 将 s 中的 "xxx" 替换为 "yy"，最多替换 3 次

s.strip();                  // 删除 s 两端的空白字符 " \t\r\n"
```

```
s.strip("ab");              // 删除 s 两端的 'a', 'b'
s.strip("ab", 'l');         // 删除 s 左端的 'a', 'b'
s.strip("ab", 'r');         // 删除 s 右端的 'a', 'b'

s.tolower();                // s 转换为小写
s.toupper();                // s 转换为大写
s.lower();                  // 返回 s 的小写形式，s 本身不变
s.upper();                  // 返回 s 的大写形式，s 本身不变
s.clone();                  // 返回 s 的一份拷贝 (memory copy)
s.match("x*y?z");           // 字符串匹配，* 匹配任意字符串，? 匹配单个字符
```

- 注意事项

fastring 中含有二进制字符时，不要使用 **find** 系列的操作:

- find()
- rfind()
- find_first_of()
- find_first_not_of()
- find_last_of()
- find_last_not_of()

上述方法基于 **strrchr**, **strstr**, **strcspn**, **strspn** 等实现，字符串中包含二进制字符时，不能保证得到正确的结果。

9. 字符串操作(str)

include: co/str.h.

9.1 切分字符串(split)

split 函数将字符串切分成若干个子串，原字符串保持不变，返回切分后的结果。

- 函数原型

```
// @s: 原字符串，fastring 或 const char*
// @c: 分隔符，单个字符或'\0'结尾的字符串
// @n: 切分次数，0 或 -1 表示不限次数，默认为 0
std::vector<fastring> split(s, c, n=0);
```

- 代码示例

```
str::split("x y z", ' ');     // ->  [ "x", "y", "z" ]
str::split("|x|y|", '|');     // ->  [ "", "x", "y" ]
str::split("xooy", "oo");     // ->  [ "x", "y"]
str::split("xooy", 'o');      // ->  [ "x", "", "y" ]
str::split("xooy", 'o', 1);   // ->  [ "x", "oy" ]
```

9.2 修剪字符串(strip)

strip 函数去掉字符串左右两边指定的字符，原字符串保持不变，返回 strip 后的结果。

- 函数原型

```
// @s: 原字符串，fastring 或 const char*
// @c: 需要去掉的字符集，单个字符或字符串
```

```
// @d: 方向, 'l' 或 'L' 修剪左边, 'r' 或 'R' 修剪右边, 默认为 'b', 修剪左右两边
fastring strip(s, c=" \t\r\n", d='b');
```

- 代码示例

```
str::strip("abxxa", "ab");       // -> "xx"    修剪两边
str::strip("abxxa", "ab", 'l');  // -> "xxa"   修剪左边
str::strip("abxxa", "ab", 'r');  // -> "abxx"  修剪右边
```

9.3 替换子串(replace)

replace 函数用于替换字符串中的子串，原字符串保持不变，返回替换后的结果。

- 函数原型

```
// @s:    原字符串, fastring 或 const char*
// @sub:  替换前的子串
// @to:   替换后的子串
// @n:    最大替换次数, 0 或 -1 表示不限制次数, 默认为 0
fastring replace(s, sub, to, n=0);
```

- 代码示例

```
str::replace("xooxoox", "oo", "ee");     // -> "xeexeex"
str::replace("xooxoox", "oo", "ee", 1);  // -> "xeexoox"
```

9.4 字符串转内置类型

str 库提供如下的函数，将字符串转为内置类型：

```
to_int32  to_int64  to_uint32  to_uint64  to_bool  to_double
```

- 函数说明
  - 若转换失败，抛出 `const char*` 类型的异常。
  - 转换为整数时，字符串末尾可带单位 **k, m, g, t, p**，不区分大小写。

- 代码示例

```
bool x = str::to_bool("false");    // "true" or "1" -> true, "false" or "0" -> false
double x = str::to_double("3.14"); // 3.14

int32 x = str::to_int32("-23");    // -23
int64 x = str::to_int64("4k");     // 4096
uint32 x = str::to_uint32("8M");   // 8 << 20
uint64 x = str::to_uint64("8T");   // 8ULL << 40
```

9.5 内置类型转字符串

str 库提供 **from** 函数，将内置类型转为字符串。

- 代码示例

```
fastring s = str::from(true);  // -> "true"
fastring s = str::from(23);    // -> "23"
fastring s = str::from(3.14);  // -> "3.14"
```

9.6 debug string

str 库提供 **dbg** 函数，从指定类型生成一个 debug 字符串。

- 函数原型

```
// @v: 内置类型，字符串类型，或常用的 STL 容器类型(vector, map, set)
fastring dbg(v);
```

- 代码示例

```
std::vector<int> v { 1, 2, 3 };
std::set<int> s { 1, 2, 3 };
std::map<int, int> m { {1, 1}, {2, 2} };
str::dbg(v);    // -> "[1,2,3]"
str::dbg(s);    // -> "{1,2,3}"
str::dbg(m);    // -> "{1:1,2:2}

str::dbg(true); // -> "true"
str::dbg(23);   // -> "23"
str::dbg("23"); // -> "\"23\"", 字符串类型，两边加引号
```

- 当字符串中含有 " 时，dbg() 生成的字符串看起来会有点瑕疵，不过此函数一般用于打日志，应该无伤大雅。

10. 命令行参数与配置文件解析库(flag)

include: co/flag.h.

10.1 基本概念

flag 库是一个类似 **google gflags** 的命令行参数及配置文件解析库，其原理很简单，代码中定义静态全局变量，然后在程序启动时解析命令行参数及配置文件，修改这些全局变量的值。

10.1.1 flag 变量

flag 库中的宏定义的静态全局变量，称为 **flag 变量**。如下面的代码定义了名为 **x** 的 flag 变量，它对应的全局变量名是 **FLG_x**。

```
DEF_bool(x, false, "xxx"); // bool FLG_x = false;
```

flag 库支持 7 种类型的 flag 变量：

```
bool, int32, int64, uint32, uint64, double, string
```

10.1.2 command line flag

命令行参数中，以 **-x=y** 的形式出现，其中 **x** 被称为一个 **command line flag**(以下都简称为 **flag**)。命令行中的 flag，与代码中的 flag 变量是一一对应的(下面不再区分二者)。flag 库为了简便易用，设计得非常灵活：

- -x=y 可以省略前面的 -，简写为 x=y.
- -x=y 也可以写成 -x y.
- x=y 前面可以带任意数量的 -.
- bool 类型的 flag，-b=true 可以简写为 -b，简写时不能省略 -.

```
./exe -b -i=32 s=hello xx  # b,i,s 是 flag，xx 不是 flag
```

10.2 flag 库的初始化

flag 库对外仅提供一个 api 函数 **flag::init()**，用于初始化 flag 库及解析命令行参数、配置文件等。

```
// 主要流程：
// 1. 扫描命令行参数，分成 flag 与非 flag 两类。
// 2. 根据 flag 参数更新 FLG_config 的值，若非空，解析由此指定的配置文件。
// 3. 根据 flag 参数更新 其他 flag 变量的值。
// 4. 若 FLG_mkconf 非空，生成配置文件，退出程序。
// 5. 若 FLG_daemon 为 true，将程序放入后台执行。

// 解析过程中遇到任何错误，输出错误信息，退出程序。
// 解析全部正常，返回非 flag 的参数列表。
std::vector<fastring> init(int argc, char** argv);
```

此函数需要在进入 **main** 函数时调用一次：

```
#include "co/flag.h"

int main(int argc, char** argv) {
    flag::init(argc, argv);
}
```

10.3 代码中定义、声明及使用 flag 变量

10.3.1 定义 flag 变量

flag 库提供了 7 个宏，分别用于定义 7 种不同类型的 flag 变量：

```
DEF_bool  DEF_int32  DEF_int64  DEF_uint32  DEF_uint64  DEF_double  DEF_string
```

下面的代码，分别定义了类型为 bool 与 string 的两个 flag 变量：

```
DEF_bool(b, false, "comments");  // bool FLG_b = false;
DEF_string(s, "x", "comments");  // fastring FLG_s = "x";
```

DEF_xxx 宏带有三个参数，第一个参数是 flag 变量名，第二个参数是默认值，第三个参数是注释。需要注意下面两点：

- flag 变量是全局变量，一般不要在头文件中定义它们。
- flag 变量的名字是唯一的，不能定义两个名字相同的 flag 变量。

10.3.2 声明 flag 变量

与定义类似，flag 库也提供了 7 个宏，分别用于声明 7 种不同类型的 flag 变量：

```
DEC_bool  DEC_int32  DEC_int64  DEC_uint32  DEC_uint64  DEC_double  DEC_string
```

下面的代码声明了一个 int32 类型的变量：

```
DEC_int32(i32); // extern int32 FLG_i32;
```

DEC_xxx 宏只有一个参数，接收 flag 变量名。一个 flag 变量只能定义一次，但可以声明多次，可以在任何需要的地方声明它们。声明一般用于引用其它地方定义的 flag 变量。

10.3.3 使用 flag 变量

定义或声明 flag 变量后，就可以像普通变量一样使用它们：

```
DEC_bool(b);
if (!FLG_b) std::cout << "b is false" << std::endl;

DEF_string(s, "hello", "xxx");
FLG_s += " world";
std::cout << FLG_s << std::endl;
```

10.4 命令行中使用 flag

10.4.1 命令行中设置 flag 变量的值

假设程序中定义了如下的 flag：

```
DEF_bool(x, false, "bool x");
DEF_bool(y, false, "bool y");
DEF_int32(i, -32, "int32");
DEF_uint64(u, 64, "uint64");
DEF_double(d, 3.14, "double");
DEF_string(s, "hello world", "string");
```

程序启动时，可以通过命令行参数修改 flag 变量的值：

- **-x=y** 也可以写成 **-x y** 或者 **x=y**

```
./xx -i=8 u=88 -s="hello world"
./xx -i 8 -u 88 -s "hello world"
```

- bool flag 设置为 true 时，可以略去其值

```
./xx -x     # -x=true
```

- 多个单字母命名的 bool flag，可以合并设置为 true

```
./xx -xy    # -x=true -y=true
```

- 整型 flag 可以带单位 **k, m, g, t, p**，不区分大小写

```
./xx i=-4k  # i=-4096
```

- 整型 flag 可以传 8, 16 进制数

```
./xx i=032  # i=26     8 进制
./xx u=0xff # u=255   16 进制
```

10.4.2 查看帮助信息

```
$ ./xx --help
usage:
    ./xx --                   print flags info
    ./xx --help               print this help info
    ./xx --mkconf             generate config file
    ./xx --daemon             run as a daemon (Linux)
```

```
    ./xx xx.conf              run with config file
    ./xx config=xx.conf       run with config file
    ./xx -x -i=8k -s=ok       run with commandline flags
    ./xx -x -i 8k -s ok       run with commandline flags
    ./xx x=true i=8192 s=ok   run with commandline flags
```

10.4.3 查看 flag 变量列表

```
$ ./xx --
--config: .path of config file
         type: string        default: ""
         from: ../../base/flag.cc
--mkconf: .generate config file
         type: bool          default: false
         from: ../../base/flag.cc
```

10.5 程序启动时指定配置文件

命令行中可以用 flag **config** 指定程序的配置文件：

```
./xx config=xx.conf
./xx xx.conf     # 若配置文件名以 .conf 或 config 结尾，且是程序命令行的第一个非 flag 参数，则可省略 config=
./xx -x xx.conf  # -x 是 flag，xx.conf 是第一个非 flag 参数
```

另外也可以在代码中调用 **flag::init()** 之前，修改 **FLG_config** 的值，指定配置文件。

10.6 自动生成配置文件

程序启动时，可以用 **--mkconf** 自动生成配置文件:

```
./xx --mkconf          # 在 xx 所在目录生成 xx.conf
./xx --mkconf -x u=88  # 配置项可以用指定的值替代默认值
```

配置项(flag) 按级别、所在文件名、所在代码行数进行排序。定义 flag 时可以在注释开头加上 **#n** 指定级别，**n** 必须是 0 到 99 之间的整数，不指定时默认为 10。

```
// 指定 -daemon 级别为 0，级别小的排在前面
DEF_bool(daemon, false, "#0 run program as a daemon");
```

- 特别说明
  - 注释以 **.** 开头的 flag，带有**隐藏**属性，不会生成到配置文件中，但 **./xx --** 可以查看。
  - 注释为空的 flag，带有**隐身**属性，不会生成到配置文件中，**./xx --** 也无法查看。

10.7 配置文件的格式

flag 库的配置文件格式，也比较灵活：

- 忽略行前、行尾的空白字符，书写更自由，不容易出错。
- **#** 或 **//** 表示注释，支持整行注释与行尾注释。
- 引号中的 **#** 或 **//** 不是注释。
- 一行一个 flag，形式强制统一为 x = y，看起来一目了然。
- **=** 号前后可以任意添加空白字符，书写更自由。
- 可以用 **** 续行，以免一行太长，影响美观。

- 字符串不支持转义，以免产生歧义。

下面是一个配置文件的示例：

```
# config file: xx.conf
  daemon = false            # 后台运行程序 (daemon 由 flag 库内部定义)
  boo = true                # 不能像命令行中那样简写为 -boo

  s =                       # 空字符串
  s = hello \
      world                 # s = "helloworld"
  s = "http://github.com"   # 引号中的 # 与 // 不是注释
  s = "I'm ok"              # 字符串中含有单引号，两端可以加双引号
  s = 'how are "U"'         # 字符串中含有双引号，两端可以加单引号

  i32 = 4k                  # 4096, 整型可以带单位 k,m,g,t,p, 不区分大小写
  i32 = 032                 #  8 进制，i32 = 26
  i32 = 0xff                # 16 进制，i32 = 255
  pi = 3.14159              # double 类型
```

11. 高效流式日志库(log)

include: co/log.h.

11.1 基本介绍

log 库是一个类似 **google glog** 的 C++ 流式日志库，打印日志比 printf 系列的函数更方便、更安全：

```
LOG << "hello world" << 23;
```

log 库内部实现中采用异步方式，日志先写入缓存，达到一定量或超过一定时间后，由后台线程一并写入文件，性能在不同平台比 glog 提升了 20~150 倍左右。

下表是在不同平台连续打印 100 万条(每条 50 字节左右) info 级别日志的测试结果：

log vs glog	google glog	co/log
win2012 HHD	1.6MB/s	180MB/s
win10 SSD	3.7MB/s	560MB/s
mac SSD	17MB/s	450MB/s
linux SSD	54MB/s	1023MB/s

11.2 Api 介绍

log 库对外仅提供两个 api 函数：

```
void init();
void close();
```

log::init() 需要在 **main** 函数开头调用一次。由于 log 库依赖于 flag 库，所以 main 函数一般得像下面这样写：

```
#include "co/flag.h"
#include "co/log.h"
```

```
int main(int argc, char** argv) {
    flag::init(argc, argv);
    log::init();
}
```

log::close() 将缓存中的日志写入文件，并退出后台写日志的线程。log 库内部会捕获 **SIGINT, SIGTERM, SIGQUIT** 等信号，在程序退出前将缓存中的日志写入文件。

11.3 打印不同级别的日志

日志分为 debug, info, warning, error, fatal 5 个级别，可以分别用宏 DLOG, LOG, WLOG, ELOG, FLOG 打印 5 种不同级别的日志：

```
DLOG << "this is DEBUG log " << 23;
LOG << "this is INFO log " << 23;
WLOG << "this is WARNING log " << 23;
ELOG << "this is ERROR log " << 23;
FLOG << "this is FATAL log " << 23;
```

打印 **fatal** 日志，表示程序出现了致命错误，log 库会打印当前线程的函数调用栈信息，并终止程序的运行。

11.4 条件日志(LOG_IF)

log 库也提供 **IF** 版的宏，接受一个条件参数，当满足指定条件时才打印日志。

- 代码示例

```
DLOG_IF(cond) << "this is DEBUG log " << 23;
LOG_IF(cond) << "this is INFO log " << 23;
WLOG_IF(cond) << "this is WARNING log " << 23;
ELOG_IF(cond) << "this is ERROR log " << 23;
FLOG_IF(cond) << "this is FATAL log " << 23;
```

11.5 每 N 条打印一次日志(LOG_EVERY_N)

log 库提供 **LOG_EVERY_N** 等宏，支持每 N 条打印一次日志，这些宏内部使用原子操作，保证线程安全性。

- 代码示例

```
// 打印第 1, 33, 65..... 条日志
DLOG_EVERY_N(32) << "this is DEBUG log " << 23;
LOG_EVERY_N(32) << "this is INFO log " << 23;
WLOG_EVERY_N(32) << "this is WARNING log " << 23;
ELOG_EVERY_N(32) << "this is ERROR log " << 23;
```

FLOG 没有这个功能，因为 FLOG 一打印，程序就挂了。

11.6 打印前 N 条日志(LOG_FIRST_N)

log 库提供 **LOG_FIRST_N** 等宏，支持打印前 N 条日志。这些宏内部同样使用原子操作，保证线程安全性。

- 代码示例

```
// 打印前 10 条日志
DLOG_FIRST_N(10) << "this is DEBUG log " << 23;
LOG_FIRST_N(10) << "this is INFO log " << 23;
```

```
WLOG_FIRST_N(10) << "this is WARNING log " << 23;
ELOG_FIRST_N(10) << "this is ERROR log " << 23;
```

11.7 CHECK: 加强版的 assert

log 库提供了一系列的 CHECK 宏，可视为加强版的 assert，这些宏在 DEBUG 模式下也不会被清除。

- 代码示例

```
CHECK(1 + 1 == 2) << "say something here";
CHECK_EQ(1 + 1, 2);  // ==
CHECK_NE(1 + 1, 2);  // !=
CHECK_GE(1 + 1, 2);  // >=
CHECK_LE(1 + 1, 2);  // <=
CHECK_GT(1 + 1, 2);  // >  greater than
CHECK_LT(1 + 1, 2);  // <  less than
```

CHECK 失败时，LOG 库会先调用 log::close() 写日志，再打印当前线程的函数调用栈信息，然后退出程序。

11.8 配置项

- log_dir

  指定日志目录，默认为当前目录下的 **logs** 目录，不存在时将会自动创建。

  ```
  DEF_string(log_dir, "logs", "Log dir, will be created if not exists");
  ```

- log_file_name

  指定日志文件名(不含路径)，默认为空，使用程序名作为日志文件名。

  ```
  DEF_string(log_file_name, "", "name of log file, using exename if empty");
  ```

- min_log_level

  指定打印日志的最小级别，用于屏蔽低级别的日志，默认为 0，打印所有级别的日志。

  ```
  DEF_int32(min_log_level, 0, "write logs at or above this level, 0-4 (debug|info|warning|error|fatal)");
  ```

- max_log_file_size

  指定日志文件的最大大小，默认 256M，超过此大小，生成新的日志文件，旧的日志文件会被重命名。

  ```
  DEF_int64(max_log_file_size, 256 << 20, "max size of log file, default: 256MB");
  ```

- max_log_file_num

  指定日志文件的最大数量，默认是 8，超过此值，删除旧的日志文件。

  ```
  DEF_uint32(max_log_file_num, 8, "max number of log files");
  ```

- max_log_buffer_size

  指定日志缓存的最大大小，默认 32M，超过此值，丢掉一半的日志。

  ```
  DEF_uint32(max_log_buffer_size, 32 << 20, "max size of log buffer, default: 32MB");
  ```

- cout

  终端日志开关，默认为 false。若为 true，将日志也打印到终端。

```
DEF_bool(cout, false, "also logging to terminal");
```

11.9 功能及性能测试

LOG 库的测试代码见 **test/log_test.cc**.

```
# 在 co 根目录执行下述命令
xmake -b log    # build log

# 打印不同类型的日志
xmake r log

# 日志也打印到终端
xmake r log -cout

# min_log_level 指定输出日志的最小级别
xmake r log -min_log_level=1   # 0-4: debug,info,warning,error,fatal

# 性能测试，单线程连续打印 100 万条 info 级别的日志
xmake r log -perf
```

12. 单元测试框架(unitest)

include: **co/unitest.h**.

unitest 是一个单元测试框架，与 **google gtest** 类似，但更简单易用。

12.1 定义测试单元及用例

- 代码示例

```
#include "co/unitest.h"
#include "co/os.h"

// 定义一个名为 os 的测试单元，os 有 3 个不同的测试用例
// 运行单元测试程序时，可用参数 -os 指定运行此单元测试中的用例
DEF_test(os) {
    DEF_case(homedir) {
        EXPECT_NE(os::homedir(), "");
    }

    DEF_case(pid) {
        EXPECT_GE(os::pid(), 0);
    }

    DEF_case(cpunum) {
        EXPECT_GT(os::cpunum(), 0);
    }
}
```

12.2 运行测试用例

co/unitest 下有一些单元测试代码，可按下述步骤编译、执行：

```
# 在 co 根目录执行下述命令
xmake -b unitest    # build unitest

# 运行所有测试用例
xmake r unitest -a

# 仅运行 os 单元中的测试用例
xmake r unitest -os
```

13. 时间库(time)

include: co/time.h.

13.1 monotonic time

monotonic time 在多数平台实现为自系统启动开始的时间，一般用于计时，比系统时间稳定，不受系统时间的影响。

- 代码示例

```
int64 us = now::us(); // 微秒
int64 ms = now::ms(); // 毫秒
```

13.2 时间字符串(now::str())

now::str() 基于 **strftime** 实现，以指定格式返回当前系统时间的字符串形式。

- 函数原型

```
// fm: 时间输出格式
fastring str(const char* fm="%Y-%m-%d %H:%M:%S");
```

- 代码示例

```
fastring s = now::str();     // "2018-08-08 08:08:08"
fastring s = now::str("%Y"); // "2028"
```

13.3 sleep

Linux 平台支持微秒级的 sleep，但 Windows 平台难以实现。因此，time 库中仅支持毫秒、秒级的 sleep。

- 代码示例

```
sleep::ms(10); // sleep for 10 milliseconds
sleep::sec(1); // sleep for 1 second
```

13.4 计时器(Timer)

Timer 基于 monotonic 时间实现，对象创建时，即开始计时。

```
Timer t;
sleep::ms(10);
```

```
int64 us = t.us(); // 微秒
int64 ms = t.ms(); // 毫秒

t.restart();       // 重新开始计时
```

14. 线程库(thread)

include: co/thread.h.

14.1 互斥锁(Mutex)

Mutex 是多线程编程中常用的一种互斥锁，同一时刻，只能有一个线程抢到锁，其他线程必须等待锁被释放。

还有一种读写锁，同一时刻，允许多个线程读，但最多只有一个线程写。在实际应用中，读写锁性能较差，本库因此移除了读写锁。

与 **Mutex** 相对应的，有一个 **MutexGuard** 类，用于互斥锁的自动获取、释放。

- 代码示例

```
Mutex m;
m.lock();          // 获取锁，若锁已被其他线程占用，则当前线程会阻塞
m.unlock();        // 释放锁
m.try_lock();      // 获取锁，若锁已被其他线程占用，返回 false，当前线程不会阻塞

MutexGuard g(m);  // 构造函数中调用 m.lock() 获取锁，析构函数中调用 m.unlock() 释放锁
```

14.2 同步事件(SyncEvent)

SyncEvent 是多线程编程中常用的一种同步机制，适用于生产者-消费者模型。

- SyncEvent 构造函数说明

```
// manual_reset: 是否在 wait 结束时手动设置 event 的同步状态
// signaled:     event 的初始状态是否为 signaled
SyncEvent(bool manual_reset=false, bool signaled=false);
```

- 代码示例

```
SyncEvent ev;
ev.wait();                 // 线程 A，等待事件同步，wait() 函数自动将 event 状态设置为 unsignaled
ev.signal();               // 线程 B，事件同步通知

SyncEvent ev(true, false); // 启用 manual_reset，等待的线程需要手动设置 event 同步状态
ev.wait(1000);             // 线程 A，等待 1000 毫秒，直到事件同步或超时
ev.reset();                // 线程 A，手动设置 event 状态为 unsignaled
ev.signal();               // 线程 B，事件同步通知
```

14.3 线程(Thread)

Thread 类是对线程的封装，创建 Thread 对象时，线程就会启动，线程函数执行完时，线程自动退出。

Thread 类除构造、析构函数外，仅提供两个方法:

- **join()**, 阻塞，等待线程函数执行完，然后退出线程
- **detach()**, 不阻塞，线程函数执行完时，自动释放系统资源
- 代码示例

```
// 启动线程
Thread x(f);                        // void f();
Thread x(f, p);                     // void f(void*);  void* p;
Thread x(&T::f, &t);                // void T::f();  T t;
Thread x(std::bind(f, 7));          // void f(int v);
Thread x(std::bind(&T::f, &t, 7));  // void T::f(int v);  T t;

// 阻塞，等线程函数执行完
x.join();

// 启动线程，并销毁 Thread 对象，线程独立于 Thread 对象运行
Thread(f).detach();
```

14.4 获取当前线程的 id

current_thread_id() 用于获取当前线程的 id，thread 库利用 **TLS** 保存线程 id，每个线程只需一次系统调用。

Linux glibc 从 **2.30** 版本开始增加了 gettid 系统调用，为避免冲突，thread 库移除了早期提供的 gettid 接口，改为 **current_thread_id**。

- 代码示例

```
int id = current_thread_id();
```

14.5 基于 TLS 的 thread_ptr

thread_ptr 用法与 **std::unique_ptr** 类似，但内部使用了 **TLS** 机制，每个线程设置并拥有自己的 ptr。

- 代码示例

```
struct T {
    void run() {
        cout << current_thread_id() << endl;
    }
};

thread_ptr<T> pt;

// 在 thread 1 的线程函数中执行
if (pt == NULL) pt.reset(new T);
pt->run();  // 打印 thread 1 的 id

// 在 thread 2 的线程函数中执行
if (pt == NULL) pt.reset(new T);
pt->run();  // 打印 thread 2 的 id
```

14.6 定时任务调度器(TaskSched)

TaskSched 类用于定时任务的调度，内部由单线程调度所有任务，但可以从任意线程添加任务。

- TaskSched 提供的方法

- run_in
- run_every
- run_daily

```
// @f: std::function<void()> 类型的函数对象

// n 秒后执行 f 一次
void run_in(f, n);

// 每 n 秒执行一次 f
void run_every(f, n);

// 每天的 hour:min:sec 执行一次
// @hour: 0-23, 默认为 0
// @min:  0-59, 默认为 0
// @sec:  0-59, 默认为 0
void run_daily(f, hour=0, min=0, sec=0);
```

- 代码示例

```
TaskSched s;                        // 启动任务调度线程
s.run_in(f, 3);                     // 3 秒后执行 f 一次     void f();
s.run_every(std::bind(f, 0), 3);    // 每 3 秒执行一次 f     void f(int);
s.run_daily(f);                     // 每天 00:00:00 执行一次 f
s.run_daily(f, 23);                 // 每天 23:00:00 执行一次 f
s.run_daily(f, 23, 30);             // 每天 23:30:00 执行一次 f
s.stop();                           // 退出任务调度线程
```

15. 协程库(co)

include: co/co.h.

15.1 基本概念

- 协程是运行于线程中的轻量级调度单位.
- 协程之于线程，类似于线程之于进程.
- 一个进程中可以存在多个线程，一个线程中可以存在多个协程.
- 协程所在的线程一般被称为调度线程.
- 一个协程发生 io 阻塞或调用 sleep 等操作时，调度线程会挂起此协程.
- 一个协程挂起时，调度线程会切换到其他等待执行的协程运行.
- 协程的切换是在用户态进行的，比线程间的切换更快.

协程非常适合写网络程序，可以实现同步的编程方式，不需要异步回调，大大减轻了程序员的思想负担。

co 协程库实现的是一种 **golang** 风格的协程，有下面几个特性:

- 内置多个调度线程，默认为系统 CPU 核数.
- 同一调度线程中的协程共用一个栈，协程挂起时，会将栈上数据 copy 出来，切换回来时再将数据 copy 到栈上. 这种方式大大降低了内存占用，单机可以轻松创建上百万协程.
- 各协程之间为平级关系，可以在任何地方(包括在协程中)创建新的协程.

co 协程库在 linux, mac, windows 等平台，分别基于 **epoll**, **kqueue**, **iocp** 实现。

co 协程库中 context 切换的相关代码，取自 **ruki** 的 **tbox**，而 tbox 则参考了 **boost** 的实现，在此表示感谢！

15.2 创建协程(go)

golang 中用关键字 **go** 创建协程，与之类似，co 库中提供 **go()** 方法创建协程。

创建协程与创建线程类似，需要指定一个协程函数，go() 方法的第一个参数就是协程函数:

```
void go(void (*f)());
void go(void (*f)(void*), void* p);  // p 指定函数参数

template<typename T>
void go(void (T::*f)(), T* p);       // p 绑定 T 类对象

void go(const std::function<void()>& f);
void go(std::function<void()>&& f);
```

实际测试发现，创建 **std::function** 类型的对象开销较大，因此 go() 特别对 **void f()** 、**void f(void*)** 、**void T::f()** 类型的函数进行了优化，实际应用中，应该优先使用这三类函数。

严格来说，go() 方法只是将 **callback** 分配到一个调度线程中，真正创建协程是由调度线程完成的。但从用户的角度看，逻辑上可以认为 go() 创建了协程，并分配到指定的调度线程中，等待被执行。

- 代码示例

```
go(f);                      // void f();
go(f, p);                   // void f(void*);   void* p;
go(&T::f, p);               // void T::f();     T* p;
go(std::bind(f, 7));        // void f(int);
go(std::bind(&T::f, p, 7)); // void T::f(int);  T* p;
```

15.3 协程 api

除 **go()** 之外，co 协程库还提供了如下的几个 api (位于 namespace co 中):

```
void sleep(unsigned int ms);
void stop();
int max_sched_num();
int sched_id();
int coroutine_id();
```

- **sleep** 在协程中调用时，调度线程会挂起此协程，切换到其他等待执行的协程运行。
- **stop** 会退出所有的调度线程，一般在进程退出前调用。
- **max_sched_num** 返回支持的最大调度线程数，目前这个值是系统 cpu 核数。
- **sched_id** 返回当前调度线程的 id，若当前线程不是调度线程，则返回 -1。id 的取值范围是 0 到 **max_sched_num-1** 。
- **coroutine_id** 返回当前协程的 id，若当前线程不是协程，则返回 -1。不同调度线程中的协程可能拥有相同的 id。
- 代码示例

```
// 每隔 1 秒打印出当前的 sched_id 与 coroutine_id
void f() {
    while (true) {
        co::sleep(1000);
        LOG << "sid: " << co::sched_id() << " cid: " << co::coroutine_id();
    }
}
```

```
int main(int argc, char** argv) {
    flag::init(argc, argv);
    log::init();

    for (int i = 0; i < 32; ++i) go(f);

    sleep::sec(8);  // 防止主线程立即退出
    co::stop();     // 退出所有调度线程
    return 0;
}
```

15.4 网络编程

co 包装了常用的 socket api，以支持一般的网络编程。这些 api 都在 **namespace co** 中，除了少数几个，一般必须在协程中调用。与原生 api 不同的是，这些 api 在 io 阻塞或调用 sleep 等操作时，调度线程会挂起当前协程，切换到其他等待执行的协程运行。

15.4.1 常用的 socket api

co 提供了一些常用的 socket api:

```
sock_t socket(int domain, int type, int proto);
sock_t tcp_socket(int af=AF_INET); // @af: address family, AF_INET, AF_INET6, etc.
sock_t udp_socket(int af=AF_INET); // @af: address family, AF_INET, AF_INET6, etc.

close  shutdown   bind   listen   accept   getsockopt
recv   recvfrom   send   sendto   connect  setsockopt

int recvn(sock_t fd, void* buf, int n, int ms=-1);
```

co 提供的 api 大部分形式上与原生的 socket api 一致，用法也几乎一样，只是有些细微的差别，说明如下:

- 原生 api 参数中的 **struct sockaddr*** 替换成了 **void***，免去手动转换的麻烦。
- **socket**, **tcp_socket**, **udp_socket** 用于创建 socket，创建的 socket 在 linux/mac 平台是非阻塞的，在 windows 平台则是 **overlapped** 的，无需用户另行设置。
- **close** 可以多带一个参数 **@ms** (默认为 0)，将当前协程挂起若干毫秒，再关闭 socket。
- **shutdown** 用单个字符 **@c** 指定关闭方向，**'r'** 关闭读，**'w'** 关闭写，默认关闭读写。

```
int shutdown(sock_t fd, char c='b');
```

- **accept** 返回的 socket 是非阻塞或 overlapped 的，无需用户另行设置。
- **connect, recv, recvn, recvfrom, send, sendto** 可以多带一个参数，指定超时时间 **@ms** (默认为 -1)。超时发生时，这些 api 返回 -1，并设置 errno 为 **ETIMEDOUT**。
- **recvn** 接收 **@n** 字节的 tcp 数据，全部接收完返回 n，连接断开返回 0，其他错误返回 -1。
- **注意:** accept, connect, recv, recvn, recvfrom, send 与 sendto 必须在协程中调用。
- **特别注意:** **close** 与 **shutdown** 虽然不会阻塞，但为了正常完成内部的清理工作，必须在协程所在的调度线程中调用。一般而言，在一个协程中进行 recv, send 等操作时，那么最好也在这个协程中调用 close, shutdown 关闭 socket。

上述 api 发生错误时返回 -1，可以用 **co::error()** 获取错误码，**co::strerror()** 查看错误描述。

15.4.2 常用的 socket option 设置

co 提供了下面的几个 api，用于设置常用的 socket 选项:

```
void set_reuseaddr(sock_t fd);                // 设置 SO_REUSEADDR
void set_tcp_nodelay(sock_t fd);              // 设置 TCP_NODELAY
void set_tcp_keepalive(sock_t fd);            // 设置 SO_KEEPALIVE
void set_send_buffer_size(sock_t fd, int n);  // 设置发送缓冲区大小
void set_recv_buffer_size(sock_t fd, int n);  // 设置接收缓冲区大小
```

15.4.3 其他 api

```
// 填充 ip 地址
bool init_ip_addr(struct sockaddr_in* addr, const char* ip, int port);
bool init_ip_addr(struct sockaddr_in6* addr, const char* ip, int port);

// ip 地址转换成字符串
fastring ip_str(struct sockaddr_in* addr);
fastring ip_str(struct sockaddr_in6* addr);

// 发送一个 RST，非正常关闭 tcp 连接，避免进入 timedwait 状态，多用于服务端
// @ms: 默认为 0，将当前协程挂起若干毫秒后，再发送 RST
void reset_tcp_socket(sock_t fd, int ms=0);

int error();                    // 返回当前错误码
const char* strerror();         // 返回当前错误码对应的字符串
const char* strerror(int err);  // 返回 @err 对应的字符串
```

15.4.4 hook 系统 api

在协程中调用 co 库的 socket api 不会阻塞，但一些三方库中调用的是系统的 socket api，仍然可能阻塞。为了解决这个问题，需要 hook 系统的 api，迫使三方库调用 hook 后的 api。

co 库目前在 linux/mac 平台已支持 hook，下面是 hook 的函数列表:

```
sleep   usleep   nanosleep

accept  accept4  connect   close  shutdown
read    readv    recv    recvfrom   recvmsg
write   writev   send    sendto     sendmsg
select  poll  gethostbyaddr  gethostbyname

gethostbyaddr_r  gethostbyname2   // linux
gethostbyname_r  gethostbyname2_r // linux

epoll_wait  // linux
kevent      // mac
```

用户一般不需要关心 api hook，有兴趣可以查看 **hook** 的源码实现。

15.4.5 基于协程的一般网络编程模式

协程可以实现高性能的同步网络编程方式。以 TCP 程序为例，服务端一般采用一个连接一个协程的模式，为每个连接创建新的协程，在协程中处理连接上的数据；客户端没必要一个连接一个协程，一般使用连接池，多个协程共用连接池中的连接。

- 服务端处理连接数据的一般模式:

```
void server_fun() {
    while (true) {
        co::recv(...);  // 接收客户端请求数据
        process(...);   // 业务处理
        co::send(...);  // 发送结果到客户端
    }
    co::close(...);     // 关闭 socket
}
```

- 客户端处理连接数据的一般模式:

```
void client_fun() {
    co::send(...);   // 发送请求数据到服务端
    co::recv(...);   // 接收服务端响应数据
    process(...);    // 业务处理
}
```

15.4.6 基于协程的 tcp server/client 示例

- server 代码示例

```
struct Connection {
    sock_t fd;   // conn fd
    fastring ip; // peer ip
    int port;    // peer port
};

void on_new_connection(void* p) {
    std::unique_ptr<Connection> conn((Connection*)p);
    sock_t fd = conn->fd;
    co::set_tcp_keepalive(fd);
    co::set_tcp_nodelay(fd);

    char buf[8] = { 0 };

    while (true) {
        int r = co::recv(fd, buf, 4);
        if (r == 0) {           // 客户端关闭了连接
            co::close(fd);      // 调用 close 正常关闭连接
            break;
        } else if (r == -1) {  // 异常错误, 直接 reset 连接
            co::reset_tcp_socket(fd, 1024);
            break;
        } else {
            LOG << "recv " << buf;
            LOG << "send pong";
            co::send(fd, "pong", 4);
        }
    }
}

void server_fun() {
    sock_t fd = co::tcp_socket();
    co::set_reuseaddr(fd);

    sock_t connfd;
    int addrlen = sizeof(sockaddr_in);
    struct sockaddr_in addr;
    co::init_ip_addr(&addr, "127.0.0.1", 7788);

    co::bind(fd, &addr, sizeof(addr));
```

```
    co::listen(fd, 1024);

    while (true) {
        connfd = co::accept(fd, &addr, &addrlen);
        if (connfd == -1) continue;

        Connection* conn = new Connection;
        conn->fd = connfd;
        conn->ip = co::ip_str(&addr);
        conn->port = ntoh16(addr.sin_port);

        // 为每个客户端连接创建一个新协程，在协程中处理连接上的数据
        co::go(on_new_connection, conn);
    }
}

go(server_fun);  // 启动 server 协程
```

- client 代码示例

```
void client_fun() {
    sock_t fd = co::tcp_socket();

    struct sockaddr_in addr;
    co::init_ip_addr(&addr, "127.0.0.1", 7788);

    co::connect(fd, &addr, sizeof(addr), 3000);
    co::set_tcp_nodelay(fd);

    char buf[8] = { 0 };

    for (int i = 0; i < 7; ++i) {
        co::sleep(1000);
        LOG << "send ping";
        co::send(fd, "ping", 4);
        co::recv(fd, buf, 4);
        LOG << "recv " << buf;
    }

    co::close(fd);
}

go(client_fun);  // 启动 client 协程
```

15.5 协程的同步机制

co 协程库实现了与线程类似的同步机制，熟悉多线程编程的开发人员，很容易从线程切换到协程编程。

15.5.1 协程锁(co::Mutex)

co::Mutex 与线程库中的 **Mutex** 类似，只是需要在协程环境中使用。协程锁获取失败时，调度线程会挂起当前协程，调度线程自身不会阻塞。

另外，co 还提供一个 **co::MutexGuard** 类，用法与线程库中的 **MutexGuard** 一样。

- 代码示例

```
co::Mutex mtx;
int v = 0;
```

```
void f1() {
    co::MutexGuard g(mtx);
    ++v;
}

void f2() {
    co::MutexGuard g(mtx);
    --v;
}

go(f1);
go(f2);
```

15.5.2 协程同步事件(co::Event)

co::Event 与线程库中的 **SyncEvent** 类似，但需要在协程环境中使用。调用 **wait()** 方法时，调度线程会挂起当前协程，调度线程自身不会阻塞。

- 代码示例

```
co::Event ev;
int v = 0;

void f1() {
    // ev.wait(100);  // 等待 100 ms
    ev.wait();        // 永久等待
    if (v == 2) v = 1;
}

void f2() {
    v = 2;
    ev.signal();
}

go(f1);
go(f2);
```

15.6 协程池

15.6.1 co::Pool

线程支持 **TLS** 机制，协程也可以支持类似的 **CLS** 机制，但考虑到系统中可能创建上百万协程，CLS 似乎不怎么高效，co 最终放弃了 CLS，取而代之实现了 **co::Pool** 类:

```
class Pool {
  public:
    Pool();
    Pool(std::function<void*()>&& ccb, std::function<void(void*)>&& dcb, size_t cap=(size_t)-1);

    void* pop();
    void push(void* p);

  private:
    void* _p;
};
```

- 构造函数

第二个构造函数中的参数 **ccb** 与 **dcb** 可用于创建、销毁元素，**cap** 则用于指定 pool 的最大容量。此处的最大容量是对单个线程而言，如 cap 设置为 1024，调度线程有 8 个，则总的最大容量实际上是 8192。另外注意，最大容量只有在同时指定了 dcb 时有效。

- pop

此方法从 pool 中拉取一个元素。pool 为空时，若设置了 ccb，则调用 ccb 创建一个元素并返回；若没有设置 ccb，则返回 NULL。

- push

此方法将元素放回 pool 中，若元素为 NULL 指针，则直接忽略。若超过最大容量，且指定了 dcb，则直接调用 dcb 销毁元素，而不放入 pool 中。

co::Pool 类是协程安全的，调用 pop, push 方法不需要加锁，但必须在协程中调用。

- 代码示例

```
co::Pool p;

void f {
    Redis* rds = (Redis*) p.pop();      // 从 pool 中拉取一个 redis 连接
    if (rds == NULL) rds = new Redis;   // pool 为空时，创建新的 redis 连接

    rds->get("xx");                     // 调用 redis 的 get 方法
    p.push(rds);                        // 用完 redis，放回 pool 中
}

go(f);
```

15.6.2 co::PoolGuard

co::PoolGuard 是一个模板类，它在构造时从 co::Pool 拉取元素，析构时将元素放回 co::Pool 中。另外，它还重载了 **operator->**，可以像智能指针一样使用它。

- 代码示例

```
// 指定 ccb，dcb，用于 Redis 的自动创建与销毁
co::Pool p(
    []() { return (void*) new Redis; },  // 指定 ccb
    [](void* p) { delete (Redis*)p; }    // 指定 dcb
);

void f() {
    co::PoolGuard<Redis> rds(p);  // rds 可视为一个 Redis* 指针
    rds->get("xx");               // 调用 redis 的 get 方法
}

go(f);
```

使用 CLS 机制，100w 协程需要建立 100w 连接，但使用 pool 机制，100w 协程可能只需要共用少量的连接。Pool 看起来比 CLS 更高效、更合理，这也是本协程库不支持 CLS 的原因。

15.7 配置项

co 库支持的配置项如下:

- co_sched_num

调度线程数，默认为系统 CPU 核数，目前的实现中，这个值必须 <= CPU 核数。

- co_stack_size

  协程栈大小，默认为 1M。每个调度线程都会分配一个栈，调度线程内的协程共用这个栈。

- co_max_recv_size

  co::recv 一次能接收的最大数据长度，默认为 1M，超过此大小，分批接收。

- co_max_send_size

  co::send 一次能发送的最大数据长度，默认为 1M，超过此大小，分批发送。

16. 高效 json 库(json)

include: co/json.h.

json 库的设计原则是精简、高效、易用，其性能堪比 **rapidjson**，如果使用 **jemalloc**，**parse** 与 **stringify** 的性能会进一步提升。

- json 库的特性
  - 支持 null、bool、int、double、string 五种基本类型.
  - 支持 array、object 两种复合类型.
  - 所有类型统一用一个 **Json** 类表示.
  - Json 类内部仅一个指针数据成员，**sizeof(Json) == sizeof(void*)**.
  - Json 内置引用计数，复制操作仅增加引用计数(**原子操作，线程安全**)，不进行内存拷贝.
  - 内置一个简单的内存分配器(Jalloc)，对大部分内存分配操作进行优化.

16.1 基本类型

- 代码示例

```
Json x;                         // null
x.is_null();                    // 判断是否为 null

Json x = false;                 // bool 类型
x.is_bool();                    // 判断是否为 bool 类型
bool b = x.get_bool();          // 获取 bool 类型的值

Json x = 123;                   // int 类型
int i = x.get_int();            // 获取 int 类型的值

Json x = (int64) 23;            // int 类型, 64位
int64 i = x.get_int64();        // 返回 64 位整数

Json x = 3.14;                  // double 类型
double d = x.get_double();      // 获取 double 类型的值

Json x = "hello world";         // 字符串类型
Json x(s, n);                   // 字符串类型 (const char* s, size_t n)
x.is_string();                  // 判断是否为字符串类型
x.size();                       // 返回字符串的长度
const char* s = x.get_string(); // 返回字符串指针, 字符串以 '\0' 结尾
```

16.2 array 类型

array 是一种数组类型，可以存储任意类型的 Json 对象。

```
Json x = json::array();      // 创建空数组，不同于 null
x.is_array();                // 判断是否为 array 类型
x.size();                    // 返回 array 中元素个数
x.empty();                   // 判断 array 是否为空

Json x;                      // null，调用 push_back 后自动变成 array 类型
x.push_back(false);          // 添加 bool 类型的值
x.push_back(1);              // 添加 int 类型的值
x.push_back(3.14);           // 添加 double 类型的值
x.push_back("hello");        // 添加 string 类型的值
x.push_back(x);              // 添加 array 类型的对象
x.push_back(obj);            // 添加 object 类型的对象

// 访问 array 成员
x[0].get_bool();
x[1].get_int();

// 遍历 array
for (uint32 i = 0; i < x.size(); ++i) {
    Json& v = x[i];
}
```

16.3 object 类型

object 类型内部以 key-value 形式存储，value 可以是任意类型的 Json 对象，key 则有下面几条限制：

- key 必须是 '\0' 结尾的 C 字符串.
- key 中不能包含双引号 ".

```
Json x = json::object();       // 创建空 object 对象，不同于 null
x.is_object();                 // 判断是否为 object 类型
x.size();                      // 返回 object 中元素个数
x.empty();                     // 判断 object 是否为空

Json x;                        // null，调用 add_member() 后自动变成 object 类型
x.add_member("name", "Bob");   // 添加字符串对象
x.add_member("age", 23);       // 添加整数类型
x.add_member("height", 1.68);  // 添加 double 类型
x.add_member("array", array);  // 添加 array 类型
x.add_member("obj", obj);      // 添加 object 类型

// has_member 与 [] 各需查找一次
x.has_member("name");          // 判断是否包含成员 "name"
x["name"].get_string();        // 获取成员的值

// key 不存在时返回 null
Json v = x.find("age");        // Json 内置引用计数，返回对象不会影响性能.
if (v.is_int()) v.get_int();

if (!(v = x.find("obj")).is_null()) {
    do_something();
}

// 遍历
for (auto it = x.begin(); it != x.end(); ++it) {
    const char* key = it->key;  // key
```

```
    Json& v = it->value;        // value
}
```

16.4 json 转字符串

Json 类提供 **str()** 与 **pretty()** 方法，将 Json 转化成字符串:

```
Json x;
fastring s = x.str();     // 返回字符串
fastring s = x.pretty();  // 返回 pretty 字符串

fastream fs;
fs << x;                  // 与 fs << x.str() 同，但效率更高
LOG << x;                 // 日志库基于 fastream 实现，可以直接打印 json 对象
```

另外 Json 类还提供一个 **dbg()** 方法，将 Json 转化成 debug 字符串，Json 内部较长的字符串类型可能被截断:

```
Json x;
fastring s = x.dbg();
LOG << x; // 实际上相当于 LOG << x.dbg();
```

16.5 字符串转 json

json::parse() 或者 Json 类中的 **parse_from()** 方法可以将字符串转化成 Json 对象:

```
Json x;
fastring s = x.str();

// parse 失败时，y 为 null
Json y = json::parse(s);
Json y = json::parse(s.data(), s.size());
y.parse_from(x.str());
```

16.6 注意事项

16.6.1 添加与查找成员

object 类型，内部用数组保存 key-value 对，这样可以保持成员添加时的顺序，但同时增加了查找成员的开销。**operator[]** 会进行查找操作，实际应用中应该尽量避免使用。

- 添加成员时用 add_member 取代 operator[]

```
// add_member 不查找，直接将成员添加到尾部
x.add_member("age", 23);  // 比 x["age"] = 23 效率更高
```

- 查找成员时用 find 取代 operator[]

```
// 传统的成员访问，3 次查找操作，效率低
if (x.has_member("age") && x["age"].is_int()) {
    int i = x["age"].get_int();
}

// 用 find 取代 []，只需一次查找操作
Json v = x.find("age");
if (v.is_int()) {
```

```
    int i = v.get_int();
}
```

16.6.2 字符串类型中的特殊字符

json 字符串内部以 '\0' 结尾，应该避免在字符串中包含二进制字符。

json 字符串支持包含 `"` 与 `\`，也支持 `\r, \n, \t` 等转义字符。但包含这些特殊字符，会降低 **json::parse()** 的性能，实际应用中应该尽量少用。

```
Json x = "hello\r\n\t";      // ok, 字符串中包含转义字符
Json x = "hello\"world";     // ok, 字符串中包含 "
Json x = "hello\\world";     // ok, 字符串中包含 \
```

17. 高性能 json rpc 框架(rpc)

include: co/rpc.h.

rpc 框架基于协程实现，内部使用 **tcp/json** 作为传输协议，简单的测试显示单线程 qps 可以达到 **12w+**。json 与基于结构体的二进制协议相比，至少有下面几个好处：

- 抓包可以直接看到传输的 json 对象，方便调试。
- rpc 调用直接传输 json 对象，不需要定义各种结构体，大大减少代码量。
- rpc 调用参数形式一致，固定为 **(const Json& req, Json& res)**，很容易自动生成代码。
- 可以实现通用的 rpc client，不需要为不同的 rpc server 生成不同的 client 代码。

17.1 rpc server 接口介绍

rpc server 的接口非常简单：

```
namespace rpc {
class Service {
  public:
    virtual ~Service() = default;
    virtual void process(const Json& req, Json& res) = 0;  // 业务处理
};

class Server {
  public:
    virtual ~Server() = default;
    virtual void start() = 0;                // 启动 rpc server 协程
    virtual void add_service(Service*) = 0;  // server 启动前必须先添加 Service 的实现
};

// 创建一个 rpc server, passwd 非空时, 客户端连接后需要进行密码认证
Server* new_server(const char* ip, int port, const char* passwd="");
} // rpc
```

rpc::Server 接收客户端连接，为每个连接创建一个新协程，新协程接收客户端请求，然后同步调用 **rpc::Service** 提供的 **process()** 方法处理请求，最后将结果发送回客户端。

具体的业务处理，需要继承 rpc::Service 并实现 process() 方法。实际上，process() 的代码是自动生成的，用户只需要实现具体的 rpc 调用方法。

17.2 实现一个 rpc server

17.2.1 定义 proto 文件

下面是一个简单的 proto 文件 **hello_world.proto**:

```
// # 或 // 表示 注释
package xx  // namespace xx

service HelloWorld {
    hello,
    world,
}

hello.req {
    "method": "hello"
}

hello.res {
    "method": "hello",
    "err": 200,
    "errmsg": "200 ok"
}

world.req {
    "method": "world"
}

world.res {
    "method": "world",
    "err": 200,
    "errmsg": "200 ok"
}
```

package xx 表示将代码生成到命名空间 **xx** 中，还可以用 package xx.yy.zz 生成嵌套命名空间。

service HelloWorld 定义一个继承 rpc::Service 的 service 类，hello, world 是它提供的两个 rpc 方法。

hello.req, hello.res, world.req, world.res 是请求参数及响应结果的示例，生成代码时不需要这些。

- 需要注意，一个 proto 文件只能定义一个 service。

17.2.2 生成 service 代码

代码生成器见 **co/gen** 目录。

- 生成 gen

```
xmake -b gen    // 在 co 根目录执行此命令，构建 gen
```

- 生成 service 代码

```
gen hello_world.proto
```

下面是生成的 C++ 头文件 **hello_world.h**:

```
#pragma once

#include "co/rpc.h"
```

```cpp
#include "co/hash.h"
#include <unordered_map>

namespace xx {

class HelloWorld : public rpc::Service {
  public:
    typedef void (HelloWorld::*Fun)(const Json&, Json&);

    HelloWorld() {
        _methods[hash64("ping")] = &HelloWorld::ping;
        _methods[hash64("hello")] = &HelloWorld::hello;
        _methods[hash64("world")] = &HelloWorld::world;
    }

    virtual ~HelloWorld() {}

    virtual void process(const Json& req, Json& res) {
        Json& method = req["method"];
        if (!method.is_string()) {
            res.add_member("err", 400);
            res.add_member("errmsg", "400 req has no method");
            return;
        }

        auto it = _methods.find(hash64(method.get_string(), method.size()));
        if (it == _methods.end()) {
            res.add_member("err", 404);
            res.add_member("errmsg", "404 method not found");
            return;
        }

        (this->*it->second)(req, res);
    }

    virtual void ping(const Json& req, Json& res) {
        res.add_member("method", "ping");
        res.add_member("err", 200);
        res.add_member("errmsg", "pong");
    }

    virtual void hello(const Json& req, Json& res) = 0;

    virtual void world(const Json& req, Json& res) = 0;

  private:
    std::unordered_map<uint64, Fun> _methods;
};

} // xx
```

可以看到 HelloWrold 的构造函数已经将 hello, world 方法注册到内部的 map 中，process() 方法根据 req 中的 **method** 字段，找到并调用对应的 rpc 方法。用户只需继承 **HelloWorld** 类，实现具体进行业务处理的 hello, world 方法即可。

业务处理方法可能在不同的线程中调用，实现时需要注意线程安全性。业务处理方法内部需要连接到其他网络服务时，可以用协程安全的 **co::Pool** 管理这些网络连接。

生成的头文件可以直接放到 server 代码所在目录，客户端不需要用到。客户端只需参考 proto 文件中的 req/res 定义，就知道怎么构造 req 发起 rpc 调用了。

17.2.3 具体的业务实现

下面的示例代码 **hello_world.cc** 给出了一个简单的实现:

```
#include "hello_world.h"

namespace xx {

class HelloWorldImpl : public HelloWorld {
  public:
    HelloWorldImpl() = default;
    virtual ~HelloWorldImpl() = default;

    virtual void hello(const Json& req, Json& res) {
        res.add_member("method", "hello");
        res.add_member("err", 200);
        res.add_member("errmsg", "200 ok");
    }

    virtual void world(const Json& req, Json& res) {
        res.add_member("method", "world");
        res.add_member("err", 200);
        res.add_member("errmsg", "200 ok");
    }
};

} // xx
```

17.2.4 启动 rpc server

启动 rpc server 一般只需要如下的三行代码:

```
rpc::Server* server = rpc::new_server("127.0.0.1", 7788, "passwd");
server->add_service(new xx::HelloWorldImpl);
server->start();
```

注意调用 **start()** 方法会创建一个协程，server 在协程中运行，防止主线程退出是用户需要关心的事。

17.3 rpc client

rpc client 的接口如下:

```
class Client {
  public:
    virtual ~Client() = default;
    virtual void ping() = 0; // send a heartbeat
    virtual void call(const Json& req, Json& res) = 0;
};

Client* new_client(const char* ip, int port, const char* passwd="");
} // rpc
```

rpc::new_client() 创建一个 rpc client，服务端若设置了密码，客户端需要带上密码进行认证。

call() 方法发起 rpc 调用，不同的 rpc 请求可以用 req 中的 method 字段标志。

ping() 方法用于给 server 端发送心跳。

- 特别提醒
  - `rpc::Client` 创建时，并没有立即建立连接，第一次发起 `rpc` 请求才会建立连接。

- **delete rpc::Client** 会关闭连接，这个操作一般需要在协程内进行。

下面是一个简单的 rpc client 示例：

```
void client_fun() {
    rpc::Client* c = rpc::new_client("127.0.0.1", 7788, "passwd");

    for (int i = 0; i < 10000; ++i) {
        Json req, res;
        req.add_member("method", "hello");
        c->call(req, res);  // 调用 hello 方法
    }

    delete c; // 在协程内 delete，是安全关闭连接所需要的
}

int main(int argc, char** argv) {
    go(client_fun); // 创建协程
    while (1) sleep::sec(7);
    return 0;
}
```

需要注意，一个 **rpc::Client** 对应一个连接，不要在多个线程中使用同一个 rpc::Client。多线程环境中，可以使用 **co::Pool** 管理客户端连接，下面是一个例子:

```
co::Pool p(
    std::bind(&rpc::new_client, "127.0.0.1", 7788, "passwd"),
    [](void* p) { delete (rpc::Client*) p; }
);

void client_fun() {
    co::PoolGuard<rpc::Client> c(p);

    for (int i = 0; i < 10; ++i) {
        Json req, res;
        req.add_member("method", "hello");
        c->call(req, res);  // 调用 hello 方法
    }
}

// 创建 8 个协程
for (int i = 0; i < 8; ++i) {
    go(client_fun);
}
```

17.4 配置项

rpc 库支持的配置项如下:

- rpc_max_msg_size

  rpc 消息最大长度，默认为 **8M**。

- rpc_recv_timeout

  rpc 接收数据超时时间，单位为毫秒，默认 **1024** 毫秒。

- rpc_send_timeout

  rpc 发送数据超时时间，单位为毫秒，默认 **1024** 毫秒。

- rpc_conn_timeout

  rpc 连接超时时间，单位为毫秒，默认 **3000** 毫秒。

- rpc_conn_idle_sec

  rpc 保持空闲连接的时间，单位为秒，默认 **180** 秒。一个连接超过此时间没有收到任何数据，server 可能会关掉此连接。

- rpc_max_idle_conn

  最大空闲连接数，默认为 **128**。连接数超过此值，server 会关掉一些空闲连接(rpc_conn_idle_sec 时间内没有接收到数据的连接)。

- rpc_log

  是否打印 rpc 日志，默认为 **true**。

18. hash 库(hash)

include: **co/hash.h**.

hash 库提供了如下的几个函数：

- hash64

计算 64 位的 hash 值，内部使用 **murmur 2 hash** 算法。

- hash32

计算 32 位的 hash 值，32 位系统使用 murmur 2 的 32 位版本，64 位系统直接取 **hash64** 的低 32 位。

- md5sum

计算字符串或指定长度数据的 md5 值，返回 **32** 字节的字符串。

- crc16

计算字符串或指定长度数据的 crc16 值，实现取自 **redis**。

- base64_encode

base64 编码，不添加 **\r, \n**，实际应用中，没有必要添加。

- base64_decode

base64 解码，解码失败时抛出 const char* 类型的异常。

- 代码示例

```
uint64 h = hash64(s);               // 计算字符串 s 的 hash 值
uint64 h = hash64(s, n);            // 计算指定长度数据的 hash 值
uint32 h = hash32(s);               // 计算 32 位 hash 值
fastring s = md5sum("hello world"); // 计算字符串的 md5，返回结果为 32 字节
uint16 x = crc16("hello world");    // 计算字符串的 crc16
fastring e = base64_encode(s);      // base64 编码，不会抛出异常
fastring d = base64_decode(e);      // base64 解码，d 应该与 s 相同
```

19. path 库(path)

include: co/path.h.

path 库移植于 **golang**，path 分隔符必须为 '/'。

- **path::clean()**

返回路径的最短等价形式，路径中连续的分隔符会被清除掉。

```
path::clean("./x//y/");     // return "x/y"
path::clean("./x/..");      // return "."
path::clean("./x/../..");   // return ".."
```

- **path::join()**

将任意数量的字符串拼接成一个完整的路径，返回 path::clean() 处理后的结果。

```
path::join("x", "y", "z");  // return "x/y/z"
path::join("/x/", "y");     // return "/x/y"
```

- **path::split()**

将路径切分为 dir, file 两部分，若路径中不含分隔符，则 dir 部分为空。返回结果满足性质 **path = dir + file**。

```
path::split("/");     // -> { "/", "" }
path::split("/a");    // -> { "/", "a" }
path::split("/a/b");  // -> { "/a/", "b" }
```

- **path::dir()**

返回路径的目录部分，返回 path::clean() 处理后的结果。

```
path::dir("a");      // return "."
path::dir("a/");     // return "a"
path::dir("/");      // return "/"
path::dir("/a");     // return "/";
```

- **path::base()**

返回路径最后的一个元素。

```
path::base("");      // return "."
path::base("/");     // return "/"
path::base("/a/");   // return "a"  忽略末尾的分隔符
path::base("/a");    // return "a"
path::base("/a/b");  // return "b"
```

- **path::ext()**

函数返回路径中文件名的扩展名。

```
path::ext("/a.cc");  // return ".cc"
path::ext("/a.cc/"); // return ""
```

20. 文件系统操作(fs)

include: co/fs.h.

fs 库最小限度的实现了常用的文件系统操作，不同平台 path 分隔符建议统一使用 '/'。

20.1 元数据操作

- 代码示例

```
bool x = fs::exists(path);  // 判断文件是否存在
bool x = fs::isdir(path);   // 判断文件是否为目录
int64 x = fs::mtime(path);  // 获取文件的修改时间
int64 x = fs::fsize(path);  // 获取文件的大小

fs::mkdir("a/b");           // mkdir a/b
fs::mkdir("a/b", true);     // mkdir -p a/b

fs::remove("x/x.txt");      // rm x/x.txt
fs::remove("a/b");          // rmdir a/b    删除空目录
fs::remove("a/b", true);    // rm -rf a/b

fs::rename("a/b", "a/c");   // 重命名
fs::symlink("/usr", "x");   // 软链接 x -> /usr, windows 需要 admin 权限
```

20.2 文件的基本读写操作

fs 库实现了 **fs::file** 类，支持文件的基本读写操作。

- fs::file 类的特性
  - 支持 **r, w, a, m** 四种读写模式，前三种与 `fopen` 保持一致，**m** 与 **w** 类似，但不会清空已存在文件的数据。
  - 不支持缓存，直接读写文件。
  - 支持 **move** 语义，可以将 `file` 对象直接放入 `STL` 容器中。
- 代码示例

```
fs::file f;               // 后续可调用 f.open() 打开文件
fs::file f("xx", 'r');    // 读模式打开文件

// 自动关闭之前打开的文件
f.open("xx", 'a');        // 追加写，文件不存在时创建
f.open("xx", 'w');        // 一般写，文件不存在时创建，文件存在时清空数据
f.open("xx", 'm');        // 修改写，文件不存在时创建，文件存在时不清数据

if (f) f.read(buf, 512);  // 读取最多 512 字节
f.write(buf, 32);         // 写入 32 字节
f.write("hello");         // 写入字符串
f.write('c');             // 写入单个字符
f.close();                // 关闭文件，file 析构时会调用 close()
```

20.3 文件流(fs::fstream)

fs::file 不支持缓存，写小文件性能较差，为此，fs 库另外提供了 **fs::fstream** 类。

- fs::fstream 类的特性
  - 只写不读，仅支持 **w, a** 两种模式。
  - 可以自定义缓存大小，默认为 **8k**。

- 支持 **move** 语义，可将 fstream 对象放入 STL 容器中。

- 代码示例

```
fs::fstream s;                    // 默认缓存为 8k
fs::fstream s(4096);              // 指定缓存为 4k
fs::fstream s("path", 'a');       // 追加模式，缓存默认为 8k
fs::fstream s("path", 'w', 4096); // 写模式，指定缓存为 4k

s.open("path", 'a');              // 打开文件，自动关闭之前已经打开的文件
if (s) s << "hello world" << 23;  // 流式写
s.append(data, size);             // 追加指定长度的数据
s.flush();                        // 将缓存中数据写入文件
s.close();                        // 关闭文件，析构时会自动关闭
```

21. 系统操作(os)

include: co/os.h.

```
os::homedir();   // 返回 home 目录路径
os::cwd();       // 返回当前工作目录路径
os::exepath();   // 返回当前进程路径
os::exename();   // 返回当前进程名
os::pid();       // 返回当前进程 id
os::cpunum();    // 返回 cpu 核数
os::daemon();    // 后台运行，仅支持 Linux 平台
```

22. 编译

CO 使用 xmake 进行编译，同时提供 cmake 支持(由 izhengfan 贡献)。

- 编译器
  - Linux: **gcc 4.8+**
  - Mac: **clang 3.3+**
  - Windows: **vs2015+**

- 安装 xmake

  windows, mac 与 debian/ubuntu 可以直接去 xmake 的 **release** 页面下载安装包，其他系统请参考 xmake 的 **Installation** 说明。

  xmake 在 linux 上默认禁止 root 用户编译，**ruki** 说不安全，可以在 **~/.bashrc** 中加上下面的一行，启用 root 编译:

  ```
  export XMAKE_ROOT=y
  ```

- 快速上手

```
# 所有命令都在 co 根目录执行，后面不再说明
xmake       # 默认编译 libco 与 gen
xmake -a    # 编译所有项目 (libco, gen, co/test, co/unitest)
```

- 编译 libco

```
xmake build libco       # 编译 libco
xmake -b libco          # 与上同
xmake b libco           # 与上同，可能需要较新版本的 xmake
```

- 编译及运行 unitest 代码

  co/unitest 是单元测试代码，用于检验 libco 库功能的正确性。

```
xmake build unitest     # build 可以简写为 -b
xmake run unitest -a    # 执行所有单元测试
xmake r unitest -a      # 同上
xmake r unitest -os     # 执行单元测试 os
xmake r unitest -json   # 执行单元测试 json
```

- 编译及运行 test 代码

  co/test 包含了一些测试代码。co/test 目录下增加 **xxx_test.cc** 源文件，然后在 co 根目录下执行 **xmake build xxx** 即可构建。

```
xmake build flag        # 编译 flag_test.cc
xmake build log         # 编译 log_test.cc
xmake build json        # 编译 json_test.cc
xmake build rapidjson   # 编译 rapidjson_test.cc
xmake build rpc         # 编译 rpc_test.cc

xmake r flag -xz        # 测试 flag 库
xmake r log             # 测试 log 库
xmake r log -cout       # 终端也打印日志
xmake r log -perf       # log 库性能测试
xmake r json            # 测试 json
xmake r rapidjson       # 测试 rapidjson
xmake r rpc             # 启动 rpc server
xmake r rpc -c          # 启动 rpc client
```

- 编译 gen

```
xmake build gen

# 建议将 gen 放到系统目录下(如 /usr/local/bin/).
gen hello_world.proto
```

  proto 文件格式可以参考 **hello_world.proto**。

- 安装

```
# 默认安装头文件、libco、gen
xmake install -o pkg          # 打包安装到 pkg 目录
xmake i -o pkg                # 同上
xmake install -o /usr/local   # 安装到 /usr/local 目录
```

cmake 编译

- 构建 libco 库和 gen

  在 Unix 系统命令行下，使用 cmake/make 进行构建：

```
cd co
mkdir build && cd build
```

```
cmake ..
make -j8
```

构建完成后会在 **build/lib** 目录下生成 libco 库文件，在 **buildb/bin** 目录下生成 **gen** 可执行文件。

- 构建 test 和 unitest

  默认不开启 test 和 unitest 的构建，如需开启，可如下设置：

  ```
  cmake .. -DBUILD_TEST=ON -DBUILD_UNITEST=ON
  cmake .. -DBUILD_ALL
  ```

- 安装 co 库

  在 Unix 命令行下，在 **make** 完成后，可进行安装：

  ```
  make install
  ```

  此命令会将头文件、库文件，以及 gen 可执行文件复制到安装目录下的相应子目录。 Linux 下默认的安装位置是 **/usr/local/**，故 **make install** 时可能需要 root 权限。

  如需更改安装位置，需在 cmake 时设置 **CMAKE_INSTALL_PREFIX** 参数：

  ```
  cmake .. -DCMAKE_INSTALL_PREFIX=pkg
  ```

23. 结语

这份文档其实还可以写得更详细一点，终因语言乏力、精力有限作罢，只能说声抱歉了。文档中难免有些疏漏、错误之处，敬请海涵与指正！

- 有问题请提交到 **github**.
- 赞助、商务合作请联系 **idealvin@qq.com**.
- 小赏作者请扫码:

Alvin idealvin@qq.com