Resume function注意事项-跨线程调用


```
awaitable_t<void> heavy_computing_sequential(int64_t val)
{
    std::cout<<val<<" @"<<std::this_thread::get_id()<<std::endl;
    val = await async_heavy_computing_tasks(val);

    std::cout<<val<<" @"<<std::this_thread::get_id()<<std::endl;
    val = await async_heavy_computing_tasks(val);

    std::cout<<val<<" @"<<std::this_thread::get_id()<<std::endl;
    val = await async_heavy_computing_tasks(val);

    std::cout<<val<<" @"<<std::this_thread::get_id()<<std::endl;
}
```

Resume function注意事项-跨线程调用


```
int main(int argc, char* argv[])
{
    std::cout<<"main thread id is "<<std::this_thread::get_id()<<std::endl;
    heavy_computing_sequential(2);
    std::this_thread::sleep(2s);
    return 0;
}
```

```
main thread id is 11716
2 @11716
4 @12688
16 @264
256 @10472
```

Resume function注意事项-其它限制

1. 避免使用阻塞操作

   诸如sleep()函数，或者是阻塞IO，或者是繁重的计算任务

2. 万分小心的使用TLS功能的函数

   诸如errno()函数，如果主动切换的时机不对，拿到的可能就是别的协程的错误码。

   又如ctime()函数，拿着返回的指针，保存到下一个时刻去用。如果期间发生了协程切换，则很可能拿到的是一个错误的数据。针对这种函数，要么老老实实的根据推荐，使用C11以后的更安全的函数；要么赶紧用一个string给构造一个“拷贝”语义的内容。

3. 考虑前后数据的变化，重新校验数据

   由于协程代码，前后会跨越较长时间，期间的数据可能发生了改变。要留意重新校验这些数据是否合理。

   对于笔者的应用场合，在游戏里判断游戏代币的数量后执行一个数据库操作，执行完毕后，很可能还需要再次校验游戏代币数量。或者选择先扣减代币，在执行后续任务失败后，把扣除的代币又还回去----通常来说，还回去的操作不会失败。

协程方案对比

	stackfull	stackcopy	stackless	
			传统方案	resume function
特点	每协程单独一个栈	所有协程共享一个栈	不需要栈空间，使用堆内存	
内存占用	高	低	低	低
切换代价	小	大	小	小
编码难度	简单	C++下极其困难	困难,通常用宏实现为状态机	简单
系统支持	操作系统支持	需协程库完成底层工作(*)	不需要特殊支持	不需要特殊支持
历史	悠久		有久远应用	NEW
可靠性	高	???	莫名担心	预计可靠
借鉴				C#
范例	libgo,...	libco,...		librf awaitable_tasks

Resume function实践-librf

https://github.com/tearshark/librf

https://github.com/tearshark/resumef

2017 CPP-Summit

Terark.DataIO

声明式序列化，类型推导与性能优化

雷鹏
CTO Terark
2017-11-9

基本特点

- 简洁、一致、可靠
- 高性能：比其它同类产品快一个数量级
  - 对比 boost.serialization, protobuf ……
- 支持的原生类型：
  - 所有基本类型
  - 所有 stl 容器类型（除 stack/queue 之外）
  - 变长 int32/uint32/int64/uint64
  - std::pair，boost::tuple
  - 对 vector 类型有特殊优化

版本控制

- 可选，而非强制
  - 对于小对象，通常不需要版本控制
- boost::serialization 的版本控制
  - 是强制的，每个非原生类型都有版本
  - 该框架的出生就是因为 boost 不能省略版本号
- 仅向后兼容（新代码可读旧数据）
  - protobuf 旧代码可读新数据，可扩展性更好

接口特点

- 非侵入：不污染名字空间
- 声明式语法：简单、一致、清爽、可靠
  - 使用宏声明，只需声明一次
  - 宏的实现高度优化
- 使用标准C++
  - vc2015+
  - gcc 4.8+
  - 其它编译器（版本）未测

基本用法：无版本


```
struct MyData1 {
    int  a, b, c;
    std::string d;
    std::set<int> e;
    var_int32_t f;  // f.t is int32
    var_uint64_t g; // g.t is uint64
    std::map<std::string, int> h;

    // 声明序列化，无版本控制
    DATA_IO_LOAD_SAVE(MyData1, &a&b&c&d&e&f&g&h)
};
```

形式上：

- & 是“前缀操作符”
- 而不是“分隔符”
- 对代码生成器更友好

基本用法：有版本


```
struct MyData2 {
   int  a, b, c;    int32_t d;     uint64_t e;
   std::string f;   std::set<int> g;
   std::map<std::string, std::vector<int> > h;

   // 声明序列化，有版本控制
   DATA_IO_LOAD_SAVE_V(MyData2, 1, // 当前版本
         &a&b&c
         &as_var_int(d) // int32, 作为 var_int32 存储
         &as_var_int(e) // uint64, 作为 var_uint64 存储
         &f&g&h)
};
```

表示带版本的 当前版本号

有版本：向后兼容


```
struct MyData3 : public MyData2 {
  std::multimap<int, std::list<vector<string>>> i;
  unsigned version;
  // 声明序列化，有版本控制，这是新版
  DATA_IO_LOAD_SAVE_V(MyData3, 2 , // 当前版本
    &a&b&c
    &as_var_int(d) // int32, 作为 var_int32 存储
    &as_var_int(e) // uint64, 作为 var_uint64 存储
    &f&g&h
    &vmg.since(2, i) // 版本2 新增了成员i.(vmg是宏定义内的函数的局部变量)
    &vmg.get_version(version) // 如果需要，将版本值存入 version 成员
  )
};
```

版本升级到了 2

Old Fields

大多数情况下应用不需要访问 version, 所以 version 成员可以省略，此时版本机制仍正常工作，只是应用无法得到 version 的值

执行序列化


```
MyData1 d1; // and set d1 values ...
MyData2 d2; // and set d2 values ...
MyData3 d3; // and set d3 values …

BigEndianDataOutput<AutoGrownMemIO> output;
BigEndianDataInput<MemIO> input;
output.resize(1024); // 可选，避免频繁扩张，相当于 vector.reserve

output << d1 << d2 << d3; // 存储,也可使用 & 代替 <<
input = output.written();
input >> d1 >> d2 >> d3; // 载入,也可使用 & 代替 <<
```

为老/库代码添加序列化

// 老/库代码不可修改，在其他文件中声明序列化

```
struct SysData1 {
        int a, b;
        string c;
};
```

```
struct SysData2 {
        float a, b;
        string c;
};
```

表示定义在类的外部: External

```
// 另一个源文件：
DATA_IO_LOAD_SAVE_E(SysData1, &a&b&c) // 无版本
DATA_IO_LOAD_SAVE_EV(SysData2, 1, &a&b&c) // 有版本 1
```

必须与相应的类在同一个 namespace

- 才能利用 Parameter dependent name lookup

架构：严格双层结构

- 逻辑层(DataIO)，模板
  - (Native|BigEndian|LittleEndian)Data(Input|Output)
  - DataIO<StreamType>
    - StreamType 可为 Stream 指
    - 使用指针 更灵活，内嵌 Stre
- 物理层(Stream)，非模板
  - Normal path is inline
  - For Mem: (Min|Seekable
  - For Buffer: (Input|Output)Buffer
  - Stream hierarchy: (I|O)Stream, FileStream…
    - Attach to XxxStreamBuffer

NativeDataInput
NativeDataOutput
Bi
Bi
Li
Li
MemIO
MinMemIO
SeekableMemIO
MemIO

InputBuffer
OutputBuffer

逻辑层：自动推导是否可以 memcpy

- 技巧：类定义内部使用 decltype:

```
class A {
  int a, b, c;
  template<class T>
  auto foo(T& t) -> decltype(t & a & b & c);
};
```

只需要 C++11

- DataIO_is_realdump
  - 一个类所有的成员都是 dumpable(可memcpy) 的
  - 并且成员之间没有（为对齐）添加的 padding
- 对 vector 类型进行特殊优化
  - dumpable 类型的所有数据可一次性读写

逻辑层：自动推导是否可以 memcpy

```
#define DATA_IO_DEDUCE_DUMP(Class, Members) \
  template<class DataIO>                    \
  auto Deduce_need_bswap(DataIO*)           \
  ->decltype(terark::DataIO_need_bswap      \
      <DataIO, false>()Members);            \
  template<class DataIO>                    \
  auto Deduce_is_realdump(DataIO*)          \
  ->decltype(terark::DataIO_is_realdump     \
    <DataIO,Class,0,true>(this)Members)     \
  template<class DataIO>                    \
  friend auto Deduce_is_dump(DataIO* dio, Class* This) \
  ->decltype(This->Deduce_is_realdump(dio));
```

非 Native 字节序时，需要对数值类

自动推导初始条件:

- Size = 0
- Dumpable = true

将成员函数的推导结果传递到全局函数，利用“参数依赖”的名字查找

逻辑层：自动推导是否可以 memcpy

```
template<class DataIO, class Class, int Size, bool MembersDumpable>
struct DataIO_is_realdump {
  typedef boost::mpl::bool_<
    MembersDumpable && sizeof(Class)==Size
  > is_dump_t; // the final result
  template<class T>
  DataIO_is_realdump<DataIO, Class,
                     Size+sizeof(T),
    MembersDumpable &&
    decltype(Deduce_is_dump((DataIO*)(0), (T*)(0)))::value
    >
  operator&(const T& x);
};
```

前面的成员是否都 dumpable

是否遍历完了所有成员且成员之间无 padding

递归：当前成员是否 dumpable
利用“参数依赖”的名字查找

物理层：(Min|AutoGrown)? MemIO

- MinMemIO, MemIO
  - 主要用于 read, 多用于解码消息
  - MemIO 有越界检查
  - MinMemIO 无越界检查
- AutoGrownMemIO
  - 主要用于 write
  - 多用于生成消息

物理层：(Input|Output) Buffer

- 缓冲未耗尽/填满时
  - this is normal path, let it inline
  - 现代 cpu 分支预测准确率高达99%以上
  - gcc/icc 更可手工指定 normal path
  - vc2008 在 normal path 中：
    - 可以将 memcpy 优化成寄存器赋值！
- 用于从流（Stream）中读写
  - Stream 可以是文件、网络、各种适配……
  - Buffer 对象引用一个 Stream，按块读写

物理层：Normal path inline


```
byte readByte() {
  if (likely(m_pos < m_end))
    return *m_pos++;
  else
    return fill_and_read_byte();
}
void ensureRead(void* buf, size_t len) {
  if (likely(m_pos+len <= m_end))
    memcpy(buf, m_pos, len); m_pos += len;
  else
    fill_and_ensureRead(buf, length);
}
```

calling non-inline function, may throw EndOfFileException

接口函数

- `DataIO_loadObject(DataIO&, T&)`
  `DataIO_saveObject(DataIO&, const T&)`
- 根据 C++ 函数名字查找规则调用这两个函数
  - 非侵入，这两个函数可和类定义分离
  - 可定义在参数依赖的任何名字空间
  - `std::swap` 是一个反例
- 这两个函数名较长且罕见，不会冲突

Parameter dependent name lookup

执行序列化


```
MyData1 d1; //
MyData2 d2; //
MyData3 d3; /

BigEndianDataOutp                    GrownMemIO> output;
BigEndianDataInput<MemIO> input;
output.resize(1024); // 可选

output << d1 << d2 << d3;
input = output.written();
input >> d1 >> d2 >> d3; /
```

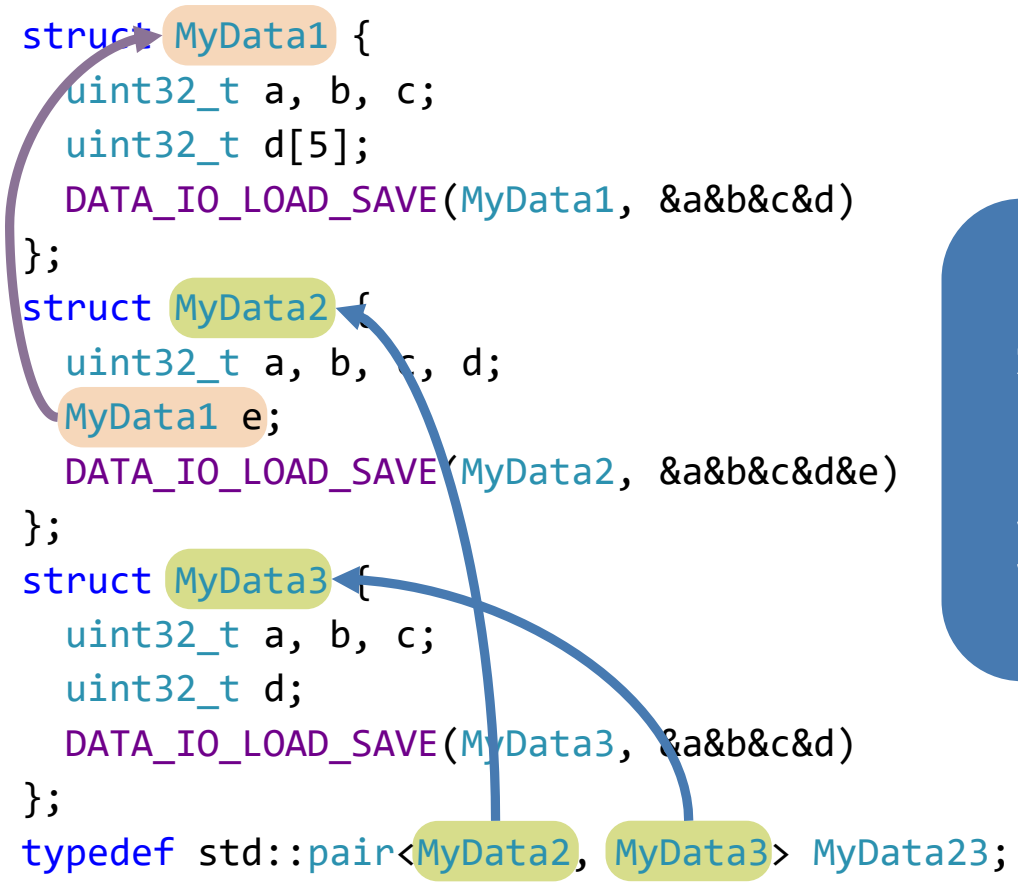
还可以是：

- MinMemIO
- InputBuffer
- FileStream ...

这里使用 MinMemIO 时，在极端情况下有超高性能，因为它只有一个指针，读写数据时没有越界检查，比较危险，需谨慎使用

性能：耗时对比（以微秒为单位）


```
struct MyData1 {
  uint32_t a, b, c;
  uint32_t d[5];
  DATA_IO_LOAD_SAVE(MyData1, &a&b&c&d)
};
struct MyData2 {
  uint32_t a, b, c, d;
  MyData1 e;
  DATA_IO_LOAD_SAVE(MyData2, &a&b&c&d&e)
};
struct MyData3 {
  uint32_t a, b, c;
  uint32_t d;
  DATA_IO_LOAD_SAVE(MyData3, &a&b&c&d)
};
typedef std::pair<MyData2, MyData3> MyData23;
```

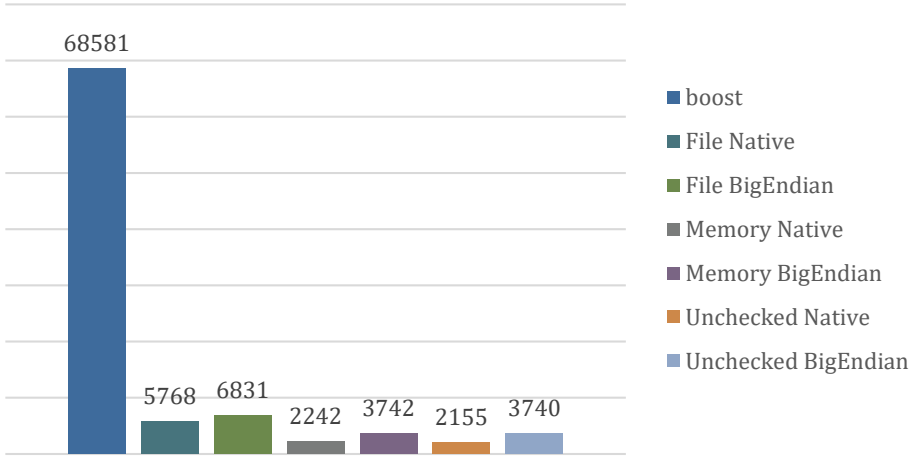


都是简单对象，

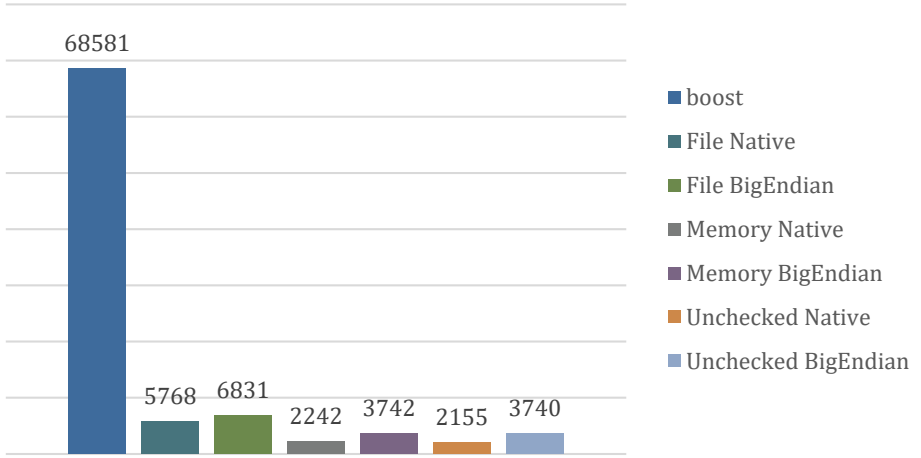
编译器容易优化

性能：耗时对比（以微秒为单位）

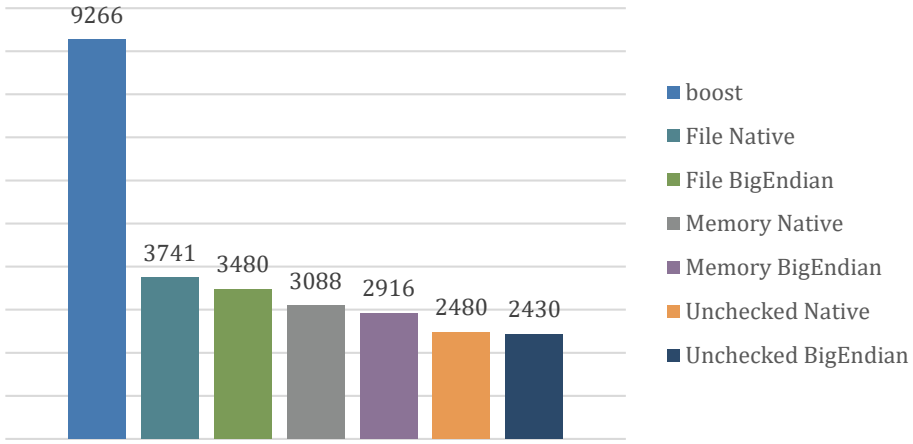
Write: vector<MyData23>



Read: vector<MyData23>



Write: vector<**string**>



Read: vector<**string**>