GCE

使用 Actor 模型的游戏通讯引擎

用户手册

v1.1

熊小磊（Nousxiong）

July 3, 2014

内容目录

第一步

为了编译 gce，你需要 CMake 和 Boost 1.55 以上版本。

Boost 需要编译如下子库：

- Boost.Atomic
- Boost.Coroutine
- Boost.Context
- Boost.System
- Boost.Regex
- Boost.DateTime
- Boost.Timer
- Boost.Chrono
- Boost.Thread

请使用 stage 模式编译 Boost。

具体编译过程见附录编译 Boost。

获取源代码：

git clone git://github.com/nousxiong/gce.git

或者：

svn checkout https://github.com/nousxiong/gce/trunk

在 **Linux** 下编译：

- (cd 到 gce 代码根目录)
- cd ..
- mkdir gce_build
- cd gce_build
- cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release -DBOOST_ROOT=your_boost_root_dir -DSUB_LIBRARYS="actor amsg" ../gce
- make
- 可选: make install (如果设置了 CMAKE_INSTALL_PREFIX，例如：-DCMAKE_INSTALL_PREFIX=../install)

在 **Windows** 下编译：

- (打开控制台，cd 到 gce 代码根目录)
- cd ..
- mkdir gce_build

- cd gce_build
- cmake -G "Visual Studio 9 2008" -DBOOST_ROOT=your_boost_root_dir -DSUB_LIBRARYS="actor amsg" ..\gce
- (打开生成的 vc 的 sln 文件，选择 ALL_BUILD 工程进行编译)
- 可选: (选择 INSTALL 工程进行编译)(如果设置了 CMAKE_INSTALL_PREFIX，例如：-DCMAKE_INSTALL_PREFIX=..\install)

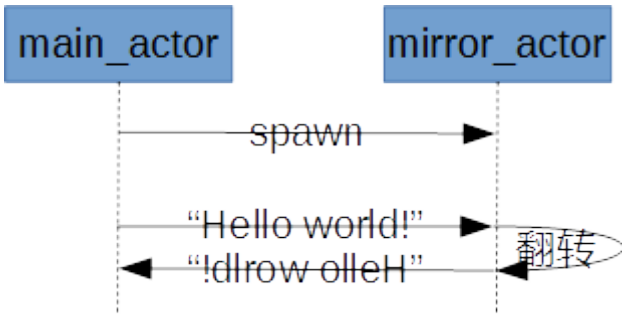
特性概览

- 轻量级、高效的 Actor 模型实现；
- 网络透明的消息传递；
- 基于 erlang 的错误处理模型；
- 消息的类型匹配机制；
- 写时拷贝消息；
- 基于线程、有栈/无栈协程和非阻塞的数种 Actor 实现；
- 与 Boost.Asio 的无缝结合；
- 轻量级集群支持；

支持的编译器

- GCC >= 4.6
- VC >= 9.0 (sp1)

Hello world 程序




```
#include <gce/actor/all.hpp>
#include <boost/assert.hpp>
#include <iostream>
#include <string>
```

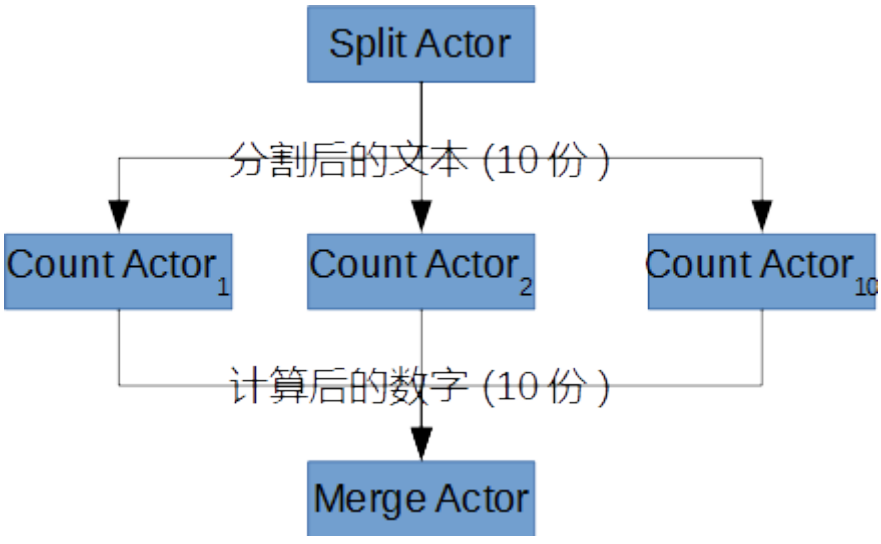
```
void mirror(gce::self_t self)
{
  /// 等待消息
  gce::message msg;
  gce::aid_t aid = self.recv(msg);
  std::string what;
  msg >> what;

  /// 打印 "Hello World!"
  std::cout << what << std::endl;

  /// 回复 "!dlroW olleH"
  gce::message m;
  m << std::string(what.rbegin(), what.rend());
  self.send(aid, m);
}

int main()
{
  /// gce 程序首先需要一个 context，这个对象的生命周期即是 gce 程序的生命周期
  gce::context ctx;

  /// 创建一个基于线程的 actor
  gce::actor<gce::threaded> main_actor = gce::spawn(ctx);

  /// 创建一个基于协程的 actor，调用 mirror 函数
  gce::aid_t mirror_actor = gce::spawn(main_actor, boost::bind(&mirror, _1));

  /// 发送 "Hello World!" 消息给 mirror
  gce::message m;
  m << std::string("Hello World!");
  main_actor.send(mirror_actor, m);

  /// ... 等待回复
  gce::message msg;
  gce::aid_t aid = main_actor.recv(msg);
  BOOST_ASSERT(aid == mirror_actor);
  std::string reply_str;
  msg >> reply_str;

  /// 打印 "!dlroW olleH"
```

```
  std::cout << reply_str << std::endl;

  return 0;
}
```

Actor 模型

- 类似人类社会，使用 Actor 模型为基础的程序，主要由一组相互关联的 actor 组成，这些 actor 相互协作共同完成一件或一系列事务；
- Actor 之间仅使用消息进行交互，不会直接访问到对方；“你不会清楚的知道另一个人心里在想什么，除非他明确的告知你”；
- 每个 actor 都能：向其它 actor 发送消息；创建新的 actor；处理接收到的其它 actor 发来的消息；
- 每个 actor 从程序角度讲其实是程序最小的并行单元，一个程序最小能分割的运行单位，使用 Actor 模型开发程序既是类似现实中组织一群人共同协作完成一件任务的过程：分解任务、尽可能并行执行子任务、综合子任务结果、然后根据情况继续重复这个流程，直到任务被完成；
- 理论上 Actor 之间的通讯是不限制方式的，线程间、进程间、机器间甚至广域网；但实际实现上，gce 和 erlang 所遵循的原则一致：认为运行环境整体在一个受信网络内（比如一个局域网内），可以透明的跨进程跨机器通讯；

举个简单的例子，比如现在要运行一个 WordCount（计算单词数量）任务，可以将这个任务细分为 Split、Count 和 Merge 三种子任务，根据需求要有 Split Actor、Count Actor 和 Merge Actor。整个作业的处理流程如下：



- Split Actor 接收到消息后可以文本分割成 10 份，每份发送给一个 Count Actor；
- Count Actor 统计好单词的数目后发送消息给 Merge Actor；
- Merge Actor 收集完 Count Actor 发送的 10 个消息后，合并每个单词的数目，完成 WordCount 任务；

从以上例子可以看出，Actor 系统跟数据驱动系统比如数据流相近，可以自定义任务的流向及其处理过程。Actor 模型被广泛使用在很多并发系统中，比如 Email、Web Service 等等。

消息

class gce::message （gce/actor/message.hpp）

消息是 gce 中 actor 之间通讯的常规手段；一般情况下，请尽量使用这种方式进行通讯，避免 actor 之间直接的数据共享。

消息类型

gce 的消息拥有类型的概念，每个 gce::message 对象同时只能有一种类型。

所谓类型，gce::match_t，本身是一个 uint64 整型，可以用 gce::atom 函数把字符串和 gce::match_t 使用类似哈希的方式相互转换，这个转换是唯一的，只要字符串唯一，转换完的 gce::match_t 也是唯一，反过来也一样；限制在于，字符串最长 13 个字节，只能 26 个小写英文字母和下划线“_”：

```
gce::match_t type = gce::atom(“my_msg_type”);
std::string type_str = gce::atom(type);
assert(type_str == “my_msg_type”);
```

Copy-On-Write

gce 的消息遵循 call-by-value 语义，所以如果同一个消息发送给多个 actor，就会有很多拷贝开销；为了避免无意义的拷贝，gce 的消息使用写时拷贝方法来优化：当同一个消息发送给多个 actor 时，数据不会被拷贝，只是维护一个引用计数，当有任何一个 actor 要更改这个消息时（例如写新数据到消息中），在引用计数大于 1 的情况下（也就是表明除了本身这个要改写消息的 actor，还至少有 1 个以上的 actor 共享这个消息），先拷贝一份新的消息数据，再对这个新拷贝进行更改操作，这样就实现了只在需要的时候进行拷贝，避免了多余的拷贝开销。

“小消息”优化

当消息很小的时候，拷贝的开销基本可以忽略，这个时候，数据是不会被分配在动态内

存上，而是直接分配在栈上，拷贝消息也直接拷贝数据，也不需要写时拷贝优化。

可以在执行 cmake 配置的时候，用-DGCE_SMALL_MSG_SIZE=[数字]，来设定这个“小消息”的最大长度（字节为单位），默认是 128 字节。例：

cmake ... -DGCE_SMALL_MSG_SIZE=128 ...

当一个消息的数据超过这个长度时，会转换为非“小消息”模式，即把数据拷贝到动态内存中，并加上引用计数。

类型匹配

有的时候，用户需要 actor 先处理完一类特定的消息之后，才会继续处理其它消息，比如一个 actor 在收到“init”类型的消息之后才会进行初始化，只有初始化完成后才会开始处理其他消息。

gce 可以让用户在接收消息时候，指定一个或几个想要接收的消息类型，来在接收时进行“匹配”。

gce::message

1. 构造函数

`message();` 构造一个空的消息，类型是 `gce::match_nil`，主要用于接收消息；
`message(gce::match_t);` 构造一个指定类型的消息，主要用于发送消息；

2. 成员函数

`std::size_t size() const;` 返回消息当前的长度；
`gce::match_t get_type() const;` 返回消息当前的类型；
`void set_type(gce::match_t);` 设置消息当前类型；

`template <typename T>`
`message& operator<<(T const&);` 序列化类型为 T 的对象到消息中，T 必须是 C++原生类型（built-in）或使用 GCE_PACK 打包的类型或 stl 容器、字符串、所有重载的 operator<<中的类型之一；
`template <typename T>`
`message& operator>>(T&);` 反序列化到类型 T 的指定参数中，T 必须是 C++原生类型（built-in）或使用 GCE_PACK 打包的类型或 stl 容器、字符串、所有重载的 operator>>中的类型之一；
例：

```
struct my_data
{
  my_data() : i_(0) {}
  my_data(int i, char const* str) : i_(i), str_(str) {}
```

```
  int i_;
  std::string str_;
};
GCE_PACK(my_data, (i_)(str_)); /// 注册 my_data 类型到 gce 中
…
int i = 0;
boost::uint16_t s = 1;
std::vector<my_data> my_data_vec;
my_data md;
my_data_vec.push_back(my_data(0, “my_data”));
my_data_vec.push_back(my_data(1, “my_data1”));
std::map<std::string, my_data> my_data_map;
my_data_map.insert(std::make_pair(std::string(“k”), my_data(2, “my_data2”)));

gce::message m;
m << i << s << md << my_data_vec << my_data_map;
…
int i2;
boost::uint16_t s2;
std::vector<my_data> my_data_vec2;
std::map<std::string, my_data> my_data_map2;
m >> i2 >> s2 >> my_data_vec2 >> my_data_map2;
…
assert(i == i2);
assert(s == s2);
assert(my_data_vec == my_data_vec2);
assert(my_data_map == my_data_map2);
```

Actors

gce/actor/actor.hpp

一些基本特征：

- actor 表示最小的可并行执行的程序单元，同一个 actor 内，程序逻辑永远是无并发顺序执行的；
- actor 之间使用消息进行通讯，不直接共享数据；
- 每个 actor 都拥有一个唯一标识 actor_id（gce::aid_t，gce/actor/actor_id.hpp），其它 actor 只能通过 actor_id 来访问自己（发送消息、链接 actor 等）；
- actor_id 在其 actor 生命周期内是不可变的；
- actor 之间可以进行双向或单向的链接，链接的双方/一方（视链接是双向还是单

向）actor 有一方因为任何原因退出，另一方都会收到一条 gce::exit 类型的消息，内含退出的原因。

- 除去非阻塞 actor 之外 ，其它类型 actor 都可以自由动态创建，没有数量上限；
- 创建新 actor（gce::spawn）必须用已有的 actor 进行，每类 actor 都有对应的 gce::spawn 重载版本；

以下按照类型分别介绍。

基于线程的 ***Actor***

C++程序均以 main 函数为入口，main 函数跑在这个程序所在的进程的主线程上，所以 gce 必须拥有一种基于线程的 actor 来作为起始点：基于线程的 actor。

基于线程的 actor 有以下几个特征：

- 基于线程进行上下文切换。例如，它在进行接收消息时，没收到之前会阻塞当前线程；
- 对同一个基于线程的 actor，同时只能有一个线程运行它；
- 同一个线程可以运行很多个基于线程的 actor；
- 基于线程的 actor 运行在主线程和用户创建的线程上，而非 gce 的内部线程；
- 基于线程的 actor 的创建和运行可以在不同的线程；

1. 创建

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
```

使用 gce::spawn(gce::context&)方法来创建一个基于线程的 actor，此方法是线程安全的。

2. 接收消息

```
/// 接上段代码
gce::message msg;
gce::aid_t aid = a.recv(msg);
/// 处理接收的消息 msg，aid 是消息的发送者
…
```

在收到消息之前 a.recv 会一直阻塞当前线程。

可以设置超时时间来控制阻塞的最长时间：

```
gce::message msg;
/// 设置阻塞超时 5 秒
gce::aid_t aid = a.recv(msg, gce::match(boost::chrono::seconds(5)));
if (aid)
{
```

```
  /// 收到消息，aid 是消息的发送者
}
else
{
  /// 超时
}
```

使用类型匹配：

```
gce::message msg;
gce::aid_t aid =
  a.recv(
    msg,
    gce::match(
      gce::atom("test"),
      gce::atom("some_op"),
      boost::chrono::seconds(5)
    )
  );
if (aid)
{
  /// 收到消息，aid 是消息的发送者，并且收到的消息类型只能是"test"或者"some_op"
  gce::match_t type = msg.get_type();
  assert(type == gce::atom("test") || type == gce::atom("some_op"));
}
else
{
  /// 超时
}
```

可以使用全局版本的 gce::recv（gce/actor/recv.hpp）来替代成员版本的 gce::actor<gce::threaded>::recv，两者区别在于：

- 前者不需要手动创建 gce::message，而是直接传入需要接收的参数，最多 5 个；
- 前者类型匹配也最多只能设定一种类型；
- 前者超时会抛出异常 std::runtime_error 替代返回空 aid；
- 前者会自动增加 gce::exit 的匹配，如果用户没有手动指定 gce::exit 的话，那么当收到 gce::exit 会抛出异常 std::runtime_error；如果用户手动指定 gce::exit 进行类型匹配，则不会抛出异常；

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
```

```
/// 假设“test”消息结构是：int + std::string + std::vector<int>
try
{
  int i;
  std::string str;
  std::vector<int> vec;
  gce::aid_t aid =
    gce::recv(
      a, /// 接收者，这里既是a，我们基于线程的actor
      gce::atom(“test”), /// 类型匹配，消息类型：”test”
      i, str, vec, /// 按照格式，依次接收对应的数据
      boost::chrono::seconds(5) /// 设置超时5秒
    );
  /// 收到”test”消息，i str vec 3个已经读取了数据，aid是消息的发送者
}
catch (std::exception& ex)
{
  /// 超时或消息格式错误或收到gce::exit
}
```

3. 发送消息

```
/// 接上段代码
/// 假设”ret”消息结构是int + std::string
gce::message m(gce::atom(“ret”));
m << (int)0 << std::string();
a.send(aid, m);
```

发送消息不会阻塞当前线程。

可以使用全局版本的gce::send（gce/actor/send.hpp）来替代成员版本的gce::actor<gce::threaded>::send，两者区别在于：

- 前者不需要手动创建gce::message，而是直接传入需要发送的参数，最多5个；

```
gce::send(a, aid, gce::atom(“ret”), int(0), std::string());
```

4. 发送请求/接收回应/回复请求

在消息模式中，请求-回应（request-reply）模式是最常用的，例如RPC（远程过程调用）。gce使用request和reply方法来实现这一模式：

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
…
```

```
/// 假设之前获得了一个 aid
gce::message m(gce::atom("ask"));
m << std::string("what is the answer to the universe and everything?");
gce::response_t res = a.request(aid, m);
gce::message msg;
gce::aid_t sender = a.recv(res, msg, boost::chrono::seconds(5)); /// 阻塞等待 res 标识的回复
if (sender)
{
  BOOST_ASSERT(sender == aid);
  if (msg.get_type() != gce::exit)
  {
    /// 成功接收到回复
    BOOST_ASSERT(msg.get_type() == gce::atom("answer"));
  }
  else
  {
    /// 对方 actor 已经销毁
  }
}
else
{
  /// 超时
}
…
/// 假设 b 就是 aid 所指向的 actor
gce::actor<gce::threaded> b = gce::spawn(ctx);
gce::message msg;
gce::aid_t asker_aid = b.recv(msg);
gce::message m(gce::atom("answer"));
m << std::string("42");
b.reply(asker_aid, m);
```

使用 request 方法发起一个请求消息，这个方法不会阻塞，而是直接返回一个 gce::response_t 数据，每一次 request 返回的 gce::response_t 都是唯一的，不会重复，其表示唯一的这一次请求；

然后用 recv 的 response_t 的重载版本来阻塞接收回复消息；

回复者（上段代码中就是 b）必须使用 reply 方法来回复接收到的请求消息，否则如果用 send 则只当做普通的消息发送，等待回复的 actor 的 recv 不会返回。如果没有可回复的请求消息，则 reply 退化为普通的 send；

可以使用全局版本的 gce::request 和 gce::reply 来替代成员版本，两者区别同全局版

本的 send。

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
…
try
{
  /// 假设之前获得了一个 aid
  gce::response_t res =
    gce::request(
      a, aid,
      gce::atom(“ask”),
      std::string(“what is the answer to the universe and everything?”)
      );
  std::string ret_str;
  gce::aid_t sender =
    gce::recv(
      a, res, ret_str,
      boost::chrono::seconds(5)
      ); /// 阻塞等待 res 标识的回复
  BOOST_ASSERT(sender == aid);
  /// 成功接收到回复
  BOOST_ASSERT(ret_str == “42”);
}
catch (std::exception& ex)
{
  /// gce::recv 超时或者目标 actor 退出后收到 gce::exit 消息
}
…
/// 假设 b 就是 aid 所指向的 actor
gce::actor<gce::threaded> b = gce::spawn(ctx);
gce::aid_t asker_aid = gce::recv(b, gce::atom(“ask”));
gce::reply(b, asker_aid, gce::atom(“answer”), std::string(“42”));
```

把 request 和 recv 分开设计是为了并发考虑，这样可以同时 request 多个 actor，然后再循环 recv 他们的回复，最大化并行；

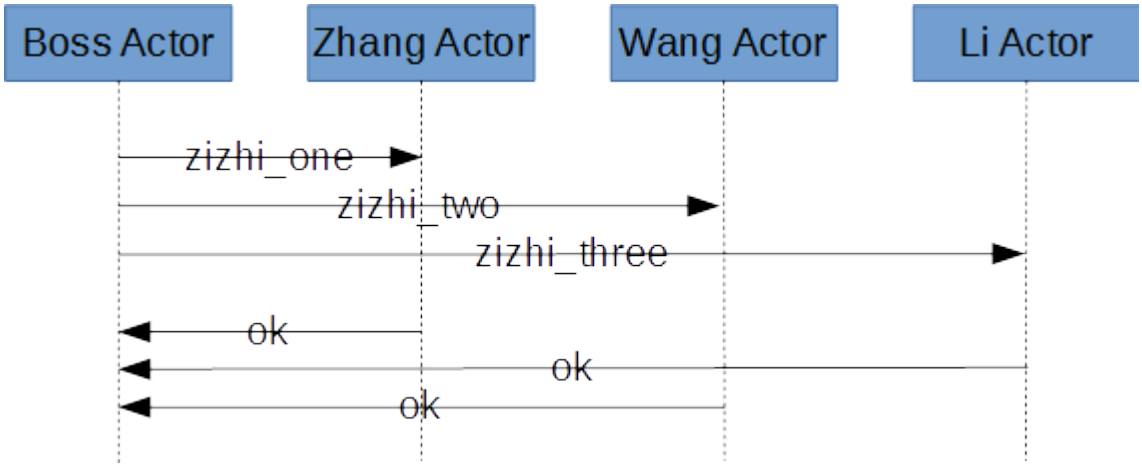
这有点类似多线程中 future 的设计。

对此，举个现实中的例子：

某个公司的老板要做一笔生意，可是做之前必须先去 3 个政府部门获得相应的 3 种不同的资质，这些资质相互之间没有任何关系，可以单独办理；于是，这个老板就让他 3 个部下小张、小王、小李去分别办理这 3 种资质，自己则在公司等待结果；中午的时候，小张回来

了，办完了资质一；下午 3 点小李办完了资质三，而晚上 8 点小王才办完资质二，老板在 8 点开始继续这笔生意的下一步工作。

这段生活中的情景用伪代码表示就是：老板和 3 个办资质的部下都是 actor，老板就让他 3 个部下小张、小王、小李去分别办理这 3 种资质，相当于老板 actor 给 3 个部下 actor 调用 request3 次，然后把得到的 3 个 response_t 存起来，依次 recv，直到 3 个 response_t 的 recv 都成功返回。



boss 代码：

```
gce::context ctx;
gce::actor<gce::threaded> boss_actor = gce::spawn(ctx);
gce::aid_t zhang_actor = gce::spawn(boss_actor, boost::bind(&zhang, _1));
gce::aid_t wang_actor = gce::spawn(boss_actor, boost::bind(&wang, _1));
gce::aid_t li_actor = gce::spawn(boss_actor, boost::bind(&li, _1));

boost::array<gce::response_t, 3> responses;
responses[0] = gce::request(boss_actor, gce::atom(“zizhi_one”));
responses[1] = gce::request(boss_actor, gce::atom(“zizhi_two”));
responses[2] = gce::request(boss_actor, gce::atom(“zizhi_three”));

for (int i=0; i<3; ++i)
{
  boost::int32_t ret = 0;
  gce::recv(boss_actor, responses[i], ret);
  if (ret != 0)
  {
    throw std::runtime_error(“资质申请失败”);
  }
}
```

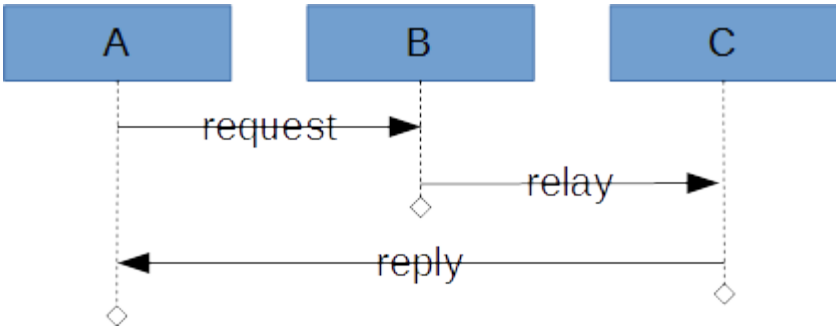
部下代码：

```
void zhang(gce::actor<gce::stackful>& self)
{
  /// 等待老板指示
  gce::aid_t boss_actor = gce::recv(self);
  /// 用wait模拟办事
  gce::wait(self, boost::chrono::hours(4));
  /// 资质办好，回复老板
  gce::reply(self, gce::atom(“zizhi_result”), boost::int32_t(0));
}

void wang(gce::actor<gce::stackful>& self)
{
  /// 等待老板指示
  gce::aid_t boss_actor = gce::recv(self);
  /// 用wait模拟办事
  gce::wait(self, boost::chrono::hours(8));
  /// 资质办好，回复老板
  gce::reply(self, gce::atom(“zizhi_result”), boost::int32_t(0));
}

void li(gce::actor<gce::stackful>& self)
{
  /// 等待老板指示
  gce::aid_t boss_actor = gce::recv(self);
  /// 用wait模拟办事
  gce::wait(self, boost::chrono::hours(6));
  /// 资质办好，回复老板
  gce::reply(self, gce::atom(“zizhi_result”), boost::int32_t(0));
}
```

5. 转发请求

有的时候，actor收到某个请求消息（request发来的消息），并不想要自己处理它，而是转发给另外的actor，这个时候可以使用relay方法来进行转发。

收到转发消息的actor并不知道这是一个转发的请求，返回的aid是原始请求发送这的aid；请求的原始发送者收到回复后，返回的aid是实际处理的那个actor：




```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
gce::actor<gce::threaded> b = gce::spawn(ctx);
gce::actor<gce::threaded> c = gce::spawn(ctx);
…
/// 假设目前有3个aid，aid_a aid_b aid_c
gce::aid_t aid_a = a.get_aid();
gce::aid_t aid_b = b.get_aid();
gce::aid_t aid_c = c.get_aid();
…
/// a发送请求给b
gce::response_t res = gce::request(a, aid_b, gce::atom(“ask”));
…
/// b收到请求后转发给c
gce::message msg;
gce::aid_t aid = b.recv(msg);
BOOST_ASSERT(aid == aid_a);
b.relay(aid_c, msg); /// 转发收到的消息msg
…
/// c收到请求处理，之后回复a
gce::aid_t aid = gce::recv(c, gce::atom(“ask”));
BOOST_ASSERT(aid == aid_a);
gce::reply(c, aid, gce::atom(“answer”));
…
/// a收到回复，发现不是b而是c作的回复
gce::aid_t aid = gce::recv(a, res);
BOOST_ASSERT(aid != aid_b);
BOOST_ASSERT(aid == aid_c);
```

6. 链接/监视

acotr之间可以进行双向或单向的链接，链接的双方/一方（视链接是双向还是单

向）actor 有一方因为任何原因退出，另一方都会收到一条 gce::exit 类型的消息，内含退出的原因。

链接的方式可以在创建 actor 的时候，或调用成员函数 link 和 monitor，link 是双向链接，monitor 则是单向（调用者监视目标，即目标退出会发送 gce::exit，而调用者退出，则不会）：

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
gce::actor<gce::threaded> b = gce::spawn(ctx);
gce::actor<gce::threaded> c = gce::spawn(ctx);
…
/// 假设目前有 3 个 aid，aid_a aid_b aid_c
gce::aid_t aid_a = a.get_aid();
gce::aid_t aid_b = b.get_aid();
gce::aid_t aid_c = c.get_aid();
…
a.link(aid_b); /// a 链接 b，无论 a 或者 b 谁退出，对方都能收到 gce::exit 消息
…
b.monitor(aid_c); /// b 监视 c，c 退出的时候，b 会收到 gce::exit 消息，b 退出则 c 不会收到
```

7. 等待

可以使用 wait 方法来阻塞一段时间（阻塞的是当前线程），在这段时间内，线程会休眠，让出时间片给其他线程：

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
a.wait(boost::chrono::seconds(5)); /// 阻塞 5 秒
```

同样，也有全局版本的 gce::wait：

```
gce::context ctx;
…
gce::actor<gce::threaded> a = gce::spawn(ctx);
gce::wait(a, boost::chrono::seconds(5)); /// 阻塞 5 秒
```

8. 能从它创建的 actor 类型

基于有栈协程的 actor、基于无栈协程的 actor、非阻塞 actor。

基于有栈协程的***Actor***

基于有栈协程的 actor 是运行在协程（coroutine）上的 actor，如果不太明白协程这个概念，请自行搜索。

基本行为和基于线程的 actor 很像，但不同在于基于有栈协程的 actor 创建的时候，必须指定要运行的函数，这是这个基于有栈协程的 actor 的唯一运行空间。

基于有栈协程的 actor 有以下几个特征：

- 基于协程进行上下文切换。例如，接收消息的时候，在未收到消息之前会阻塞当前协程（但不会阻塞当前线程，阻塞协程相当于挂起协程，让出 cpu 时间片让别的协程继续在这个线程上运行）；
- 创建的时候，用户必须指定运行的函数，这个函数就是这个 actor 的生命周期，当函数以各种方式退出（return 或抛出异常），这个 actor 就结束运行；
- 其运行的协程本身是运行在 gce 的内部线程上，每一次上下文切换，会根据负载情况，自动调度到不同的线程，这个过程对用户透明；
- gce 使用 Boost.Coroutine 实现的协程，而其又是使用 Boost.Context 实现的私有堆栈，其内部实现是使用操作系统的相对应的 api；目前主流的几个操作系统（win、linux 等），在创建协程私有堆栈时会有一个最小限制（4~8k），所以这会导致每个基于有栈协程的 actor 至少占用 4~8k 的内存，因而限制了其同时存在的数量；根据操作系统的分页内存配置，同时存在的数量可能不能超过数万；如有需求超过这个限制，请使用基于无栈协程的 actor；

1. 创建

基于有栈协程的 actor 可以用两种 actor 创建：基于线程的 actor 和自己：

```
gce::context ctx;
…
gce::actor<gce::threaded> base = gce::spawn(ctx);
…
void func(gce::actor<gce::stackful>& self)
{
  /// 基于有栈协程的 actor 的生命周期
}
…
/// 使用基于线程的 actor 创建
gce::aid_t aid = gce::spawn(base, boost::bind(&func, _1));
…
void actor_func(gce::actor<gce::stackful>& self)
{
  /// 使用基于有栈协程的 actor 创建
```

```
  gce::aid_t aid = gce::spawn(self, boost::bind(&func, _1));
}
```

可以在创建的时候指定和这个新 actor 链接类型：

```
gce::spawn(
  base, boost::bind(&func, _1),
  gce::linked /// 指定链接类型为gce::linked（双向），默认是gce::no_link（不链接）
  );
```

当使用有栈协程的 actor 创建时，可以在创建的时候指定是否和其创建者 actor（一般称为 sire）保持同步，即其和其创建者始终不会并行执行，相互之间线程安全：

```
void func(gce::actor<gce::stackful>& self)
{
  gce::spawn(
    self, boost::bind(&func, _1),
    gce::no_link, /// 指定链接类型为gce::no_link（不链接）
    true /// 和sire保持同步，默认为false
    );
}
```

可以在创建的时候指定协程堆栈的大小，单位字节（内部有上下限，超过界限会自动取对应的临界值）：

```
gce::spawn(
  base, boost::bind(&func, _1),
  gce::no_link,
  gce::minimum_stacksize() /// 默认为 gce::default_stacksize，或者手动指定这个值
  );
…
void func(gce::actor<gce::stackful>& self)
{
  gce::spawn(
    self, boost::bind(&func, _1),
    gce::no_link,
    false,
    gce::minimum_stacksize() /// 默认为 gce::default_stacksize，或者手动指定这个值
    );
}
```

2. 接收消息

除了阻塞的是协程而非线程外，同基于线程的 actor。

3. 发送消息

同基于线程的 actor。

4. 发送请求/接收回应/回复请求

同基于线程的 actor。

5. 转发请求

同基于线程的 actor。

6. 链接/监视

同基于线程的 actor。

7. 等待

除了阻塞的是协程而非线程外，同基于线程的 actor。

8. 能从它创建的 **actor** 类型

基于有栈协程的 actor、基于无栈协程的 actor。

基于无栈协程的 ***Actor***

基于无栈协程的 actor 也和基于有栈协程的 actor 一样，运行在协程上，创建时需要传入其运行的函数。但不同之处在于，它的协程并没有私有堆栈，这意味着，它在进行上下文切换的时候，无法保存局部变量，需要用户自己处理（例如，使用类或者结构体的成员变量来替代局部变量）。

使用基于无栈协程的 Actor 写起代码比基于有栈协程的 actor 要麻烦一些，所以一般并不推荐使用，通常在开发时，选用基于有栈协程的 actor，开发完毕后，经过测试确定系统内存消耗的瓶颈在某种或某几种 actor 后，再将其换成基于无栈协程的 Actor。

基于无栈协程的 Actor 有以下几个特征：

- 基于协程进行上下文切换，但不会保存局部变量，用户需要自己维护堆栈；
- 基于无栈协程的 actor 的运行函数必须使用 GCE_REENTER 宏来作为上下文切换后的重入记录点；
- 在上下文切换的函数前使用 GCE_YIELD 宏来提示 actor；
- 基于无栈协程的 actor 是使用 boost::asio::coroutine 实现的；对于 boost::asio::coroutine 解析，可以见这里：http://maipianshuo.com/?p=329

其余同基于有栈协程的 actor。

1. 创建

基于有栈协程的 actor 可以用三种 actor 创建：基于线程的 actor、基于有栈协程的 actor 和自己：

```
gce::context ctx;
…
gce::actor<gce::threaded> base = gce::spawn(ctx);
…
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    /// 基于无栈协程的 actor 的生命周期
    try
    {
      GCE_REENTER (self)
      {
        /// 所有基于无栈协程的 actor 的运行函数必须使用宏 GCE_REENTER 来“包住”逻辑代码
        /// 如果想要自己处理异常，必须将 try...catch 放到外面，否则会导致编译错误
      }
    }
    catch (std::exception&)
    {
    }
  }
};
…
/// 使用基于线程的 actor 创建
boost::shared_ptr<my_stackless_actor> a_ptr(
  boost::make_shared<my_stackless_actor>()
  );
gce::aid_t aid =
  gce::spawn<gce::stackless>(
    base,
    boost::bind(&my_stackless_actor::run, a_ptr, _1)
    );
…
void actor_func(gce::actor<gce::stackful>& self)
{
  /// 使用基于有栈协程的 actor 创建
```

```
  boost::shared_ptr<my_stackless_actor> a_ptr(
    boost::make_shared<my_stackless_actor>()
    );
  gce::aid_t aid =
    gce::spawn<gce::stackless>(
      self,
      boost::bind(&my_stackless_actor::run, a_ptr, _1)
      );
}
…
class my_stackless_actor2
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      /// 使用基于无栈协程的actor创建
      GCE_YIELD /// 必须在是有上下文切换的函数之前加上GCE_YIELD宏
      {
        /// 如果使用临时变量则其作用域在超过GCE_YIELD的{}之后就失效
        /// 因此，为了接下来的逻辑，使用成员变量
        /// 使用智能指针的原因是方便资源管理，运行的actor函数引用了本身的对象，存入
        /// 无栈actor中，直到actor销毁
        boost::shared_ptr<my_stackless_actor> a_ptr(
          boost::make_shared<my_stackless_actor>()
          );
        gce::spawn<gce::stackless>(
          self,
          boost::bind(&my_stackless_actor::run, a_ptr, _1),
          aid_ /// 输出aid参数改在这里，由于上下文切换，必须使用非局部变量
          );
      }
    }
  }
private:
  gce::aid_t aid_;
};
```

可以在创建时指定和这个新actor的链接类型：

```
gce::aid_t aid =
  gce::spawn<gce::stackless>(
```

```
    base,
    boost::bind(&my_stackless_actor::run, a_ptr, _1),
    gce::linked
    );
…
gce::aid_t aid =
  gce::spawn<gce::stackless>(
    self,
    boost::bind(&my_stackless_actor::run, a_ptr, _1),
    gce::linked
    );
…
gce::spawn<gce::stackless>(
  self,
  boost::bind(&my_stackless_actor::run, a_ptr, _1),
  aid_,
  gce::linked
  );
```

当使用基于有栈/无栈协程 actor 创建时，可以在创建的时候指定是否和其创建者 actor（一般称为 sire）保持同步，即其和其创建者始终不会并行执行，相互之间线程安全：

```
gce::aid_t aid =
  gce::spawn<gce::stackless>(
    self,
    boost::bind(&my_stackless_actor::run, a_ptr, _1),
    gce::no_link,
    true /// 和 sire 保持同步
    );
…
gce::spawn<gce::stackless>(
  self,
  boost::bind(&my_stackless_actor::run, a_ptr, _1),
  aid_,
  gce::linked,
  true /// 和 sire 保持同步
  );
```

由于是无栈的，所以不需要创建时指定栈大小。

2. 接收消息

和基于有栈协程的 actor 不同，基于无栈协程的 actor 的接收消息需要将返回的 aid 参

数放入 recv 方法的第一个参数，而且由于需要上下文切换，需要在调用方法的前面加上 GCE_YIELD 宏：

```
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      /// 由于上下文切换，所以这里作为输出参数 aid_ 和 msg_ 必须不能是局部变量
      GCE_YIELD self.recv(aid_, msg_);
      /// 或
      GCE_YIELD
      {
        self.recv(aid_, msg_);
      }
    }
  }
private:
  gce::aid_t aid_;
  gce::message msg_;
};
```

跟随{}的 GCE_YIELD 宏，是可以在{}中创建局部变量，但仅限{}中使用。

可以使用全局版本的 gce::recv（gce/actor/recv.hpp）来替代成员版本的 gce::actor<gce::stackless>::recv，两者区别在于：

- 前者不需要手动创建 gce::message，而是直接传入需要接收的参数，最多 5 个；
- 前者类型匹配也最多只能设定一种类型；
- 前者超时会立刻结束 actor 替代返回空 aid；
- 前者会自动增加 gce::exit 的匹配，如果用户没有手动指定 gce::exit 的话，那么当收到 gce::exit 会立刻结束 actor；如果用户手动指定 gce::exit 的匹配，则不会；

```
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      /// 由于上下文切换，所以这里作为输出参数 aid_ 和 msg_ 必须不能是局部变量
      GCE_YIELD gce::recv(self, aid_, msg_);
```

```
      /// 或
      GCE_YIELD
      {
        gce::recv(self, aid_, msg_);
      }
    }
  }
private:
  gce::aid_t aid_;
  gce::message msg_;
};
```

3. 发送消息

同基于有栈协程的 actor。

4. 发送请求/接收回应/回复请求

发送请求和回复请求同基于有栈协程的 actor。

接收回应则因为上下文切换，需要使用 GCE_YIELD 宏：

```
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      …
      GCE_YIELD self.recv(aid_, msg_); /// 收到某 actor 的消息
      GCE_YIELD
      {
        gce::message m(gce::atom("ask"));
        m << std::string("how old are you?");
        gce::response_t res = self.request(aid_, m);
        aid_ = gce::aid_t(); /// 清空 aid_，用于新的 recv
        self.recv(res, aid_, msg_, boost::chrono::seconds(5));
      }
      if (aid_)
      {
        /// 成功收到回应
      }
      else
```

```
      {
        /// 超时
      }
    }
  }
private:
  gce::aid_t aid_;
  gce::message msg_;
};
```

可以使用全局的 gce::request 和 gce::recv：

```
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      …
      GCE_YIELD gce::recv(self, aid_, msg_); /// 收到某 actor 的消息
      GCE_YIELD
      {
        gce::response_t res =
          gce::request(
            self, aid_, gce::atom("ask"),
            std::string("how old are you?")
            );
        aid_ = gce::aid_t(); /// 清空 aid_，用于新的 recv
        gce::recv(self, res, aid_, str_, boost::chrono::seconds(5));
      }
      /// 成功收到回应，若失败，则 actor 会退出
      …
    }
  }
private:
  gce::aid_t aid_;
  gce::message msg_;
  std::string str_;
};
```

5. 转发请求

同基于有栈协程的 actor。

6. 链接/监视

同基于有栈协程的 actor。

7. 等待

上下文切换，使用 GCE_YIELD 宏：

```
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      GCE_YIELD self.wait(boost::chrono::seconds(5));
    }
  }
};
```

可以使用全局的 gce::wait：

```
class my_stackless_actor
{
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      GCE_YIELD gce::wait(self, boost::chrono::seconds(5));
    }
  }
};
```

8. 能从它创建的 actor 类型

基于无栈协程的 actor。

非阻塞 *Actor*

非阻塞 actor 是专门为网游客户端程序设计的 actor。网游客户端经常需要在每帧或每几帧接收一次消息，这种接收，只是看下消息队列中有否消息，有会返回消息，没有也不会等待，而是留待下次接收进行“查看一下”；gce 的非阻塞 actor 的 recv 方法经过优化，使得这种方式很高效，即便每秒数万次 recv 消耗时间也是几乎可以忽略的。

目前介绍的几种 actor 中，只有它的语义和其它几种不同，它的特征有：

- 非阻塞 actor 不会导致上下文切换，所有的方法，包括 recv 都是立刻返回的；
- 非阻塞 actor 的数量是固定的，只能在 gce::context 的构造函数参数中进行设置，之后不能改变；
- 只能由基于线程的 actor 创建；
- 非阻塞 actor 运行在创建它的基于线程的 actor 所运行的用户线程上，对同一个非阻塞 actor，同时只能有一个线程运行它；
- 同一个线程可以运行多个非阻塞 actor；
- 它不能创建任何 actor，包括自己；

1. 创建

只能使用基于线程的 actor 创建。

```
gce::context ctx;
…
gce::actor<gce::threaded> base = gce::spawn(ctx);
gce::actor<gce::nonblocked> a = gce::spawn(base);
```

2. 接收消息

不会导致上下文切换，无论成员还是全局的无阻塞 actor 的 recv 都回立即返回，只是“看一眼”消息队列中有否符合条件的消息而已。

```
/// 接上段代码
void update(float dt)
{
  /// 这个函数在网游客户端每帧都调用一次，假设 fps 是 60，则平均一秒 60 次
  gce::message msg;
  gce::aid_t aid = a.recv(msg);
  if (aid)
  {
    /// 成功接收到消息，假设消息格式为 std::string
    std::string str;
    msg >> str;
  }
  else
  {
    /// 无消息
  }
}
```

可以使用全局的 gce::recv 来直接接收数据，和之前的几种 actor 的不同之处在于，如果没有符合条件的消息（aid 返回空），不会抛出异常，而只是返回空 aid：

```
/// 接上段代码
void update(float dt)
{
  /// 这个函数在网游客户端每帧都调用一次，假设 fps 是 60，则平均一秒 60 次
  gce::message msg;
  std::string str;
  gce::aid_t aid = gce::recv(a, msg, gce::atom("op"), str);
  if (aid)
  {
    /// 成功接收到消息，假设消息格式为 std::string
    /// str 已经成功设置数据
  }
  else
  {
    /// 无消息
  }
}
```

3. 发送消息

同基于有栈协程的 actor。

4. 发送请求/接收回应/回复请求

同基于有栈协程的 actor。

5. 转发请求

同基于有栈协程的 actor。

6. 链接/监视

同基于有栈协程的 actor。

7. 等待

由于没有上下文切换，无阻塞 actor 没有这个功能；

8. 能从它创建的 actor 类型

无。

服务

有的 actor 需要长时间保持运行，类似一个服务，由于 aid 在 actor 结束后就会失效，所以需要有一个 well-known 的唯一名字来作为服务的访问方式，这样作为服务的 actor

在结束和重启之后都能用名字访问到它。

gce 中服务 actor 的唯一名字数据由 gce::svcid_t（gce/actor/service_id.hpp）表示；gce 中无论成员还是全局的 send、relay、request、reply 均有 gce::svcid_t 的重载版本。

1. 注册服务

actor 需要注册才能成为服务，使用全局方法 gce::register_service 来注册一个 actor 为服务。

注意：

- 只有基于有栈协程的 actor 和基于无栈协程的 actor 才能成为服务；
- 链接和监视则不能对 svcid_t 进行操作，只能针对 aid；

```
gce::attributes attrs;
attrs.id_ = gce::atom(“my_ctx”);
gce::context ctx;
gce::actor<gce::threaded> a = gce::spawn(ctx);
…
void func(gce::actor<gce::stackful>& self)
{
  /// 注册服务，从这以后其它 actor 可以直接用对应的 svcid 来访问
  gce::register_service(self, gce::atom(“my_service”));
}
…
/// 之后在别的地方，其它 actor：
gce::svcid_t svcid =
  gce::make_svcid(
    gce::atom(“my_ctx”),
    gce::atom(“my_service”)
    );
/// 可以直接用 svcid 来发送消息给对应的服务 actor
gce::send(a, svcid, std::string(“hi”));
```

2. 注销服务

使用 gce::deregister_serivce 来注销一个服务：

```
void func(gce::actor<gce::stackful>& self)
{
  /// 注册服务，从这以后其它 actor 可以直接用对应的 svcid 来访问
  gce::register_service(self, gce::atom(“my_service”));
  …
  /// 注销服务，从这以后名字”my_serivce”不能再访问到这个 actor
```

```
  gce::deregister_service(self, gce::atom("my_service"));
}
```

集群

和 erlang 一样，gce 的 actor 是可以跨机器跨进程通讯的，与进程内通讯的方式（上述 4 种 actor 的描述）没有什么不同。

gce::context 拥有一个 id，默认是空，用户可以设置（可以用 atom）；如果要构建跨进程跨机器的 gce 集群，每个进程中的 gce::context 的 id 必须唯一、而且不能是空，这是构建 gce 集群的前提。

aid 中有其来自哪个 gce::context 的信息，这也是能透明跨机器通讯的原因之一。

libs/actor/example/cluster 和 libs/actor/example/cluster_client 两个代码示例演示了 gce 集群的基本结构。

连接 *gce::context*

构建一个 gce 集群的第一步是要连接集群内的 gce::context 对象。

每个 gce::context 可以看作一个集群内逻辑上的节点，用户可以根据需要自己自由连接这些节点——网络拓扑是星型还是环形还是什么型，均可。

连接的方式目前仅支持 tcp。

连接的过程类似 tcp 连接，一个 gce::context 通过 gce::bind 来监听一个地址，等待其它 gce::context 使用 gce::connect 连接进来。

1. bind

注意：

- 非阻塞 actor 无法使用 bind；
- 一旦成功 bind，这个监听会一直持续，直到程序结束，无论调用者 actor 是否退出；

```
gce::attributes attrs;
attrs.id_ = gce::atom("node_master");
gce::context ctx(attrs); /// 创建一个 id 为"node_master"的 gce::context
…
void func(gce::actor<gce::stackful>& self)
{
  /// 监听本地 14923 端口的 tcp 连接
  gce::bind(self, "tcp://127.0.0.1:14923");
}
```

2. connect

注意：

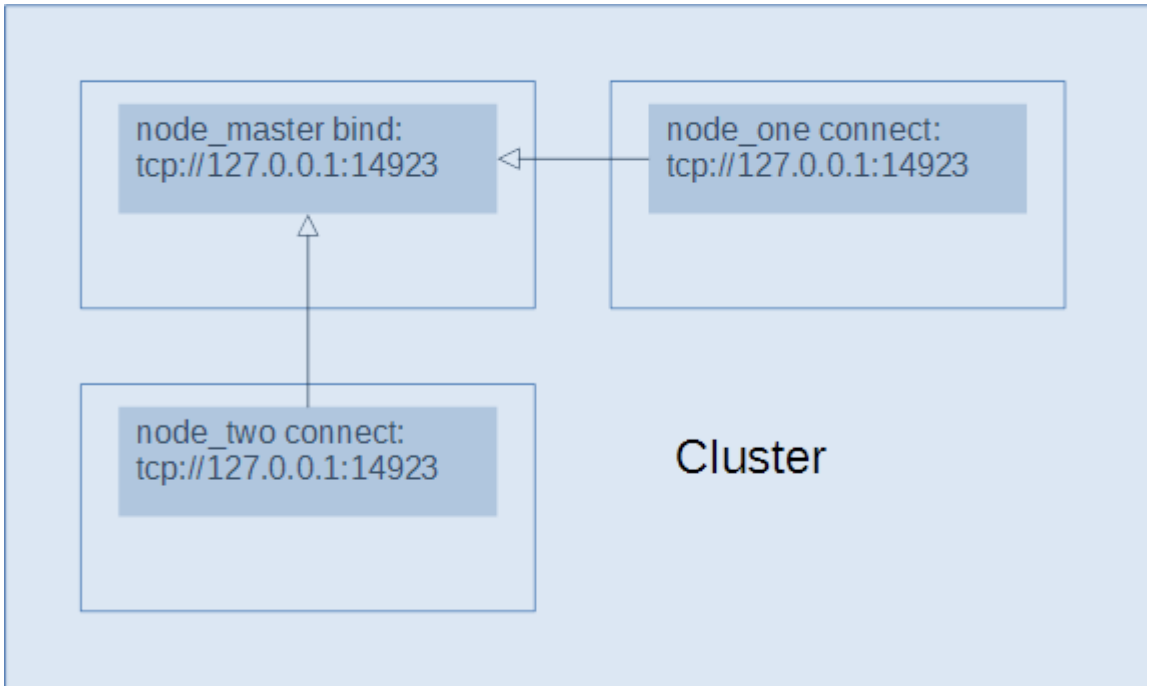
- 非阻塞 actor 无法使用 connect；
- 一旦成功 connect，这个连接会一直持续，直到程序结束，无论调用者 actor 是否退出；
- 如果网络中断或者对方进程崩溃，connect 会按照一定的时间间隔尝试重连；

```
gce::attributes attrs;
attrs.id_ = gce::atom("node_one");
gce::context ctx(attrs); /// 创建一个 id 为"node_one"的 gce::context
…
void func(gce::actor<gce::stackful>& self)
{
  /// 使用 tcp 连接本地 14923 端口、id 为"node_master"的 gce::context
  gce::connect(self, gce::atom("node_master"), "tcp://127.0.0.1:14923");
}
```

当连接成功后，用户就可以自由使用本地的 send、recv、link 等方法透明的对来自另外 gce::context 的 aid 或 svcid 的 actor 进行通讯。

3. 路由

路由是一种中转节点，用于那些不方便直接连接的 gce::context 之间中转消息。gce::bind 的参数中指明是否这个监听的地址是路由，之后其他连接进来的 gce::context 就可以和其它连接这个路由的 gce::context 通讯：



```
gce::attributes attrs;
attrs.id_ = gce::atom(“router”);
gce::context ctx(attrs); /// 创建一个id为“router”的gce::context，作为路由
…
void func(gce::actor<gce::stackful>& self)
{
  /// 监听本地14923端口的tcp连接，true表示是路由节点，默认为false
  gce::bind(self, “tcp://127.0.0.1:14923”, true);
}
…
gce::attributes attrs;
attrs.id_ = gce::atom(“node_one”);
gce::context ctx(attrs); /// 创建一个id为“node_one”的gce::context
…
void func(gce::actor<gce::stackful>& self)
{
  /// 使用tcp连接本地14923端口、id为“router”的gce::context，true表示目标是路由节点
  gce::connect(self, gce::atom(“router”), “tcp://127.0.0.1:14923”, true);
}
…
gce::attributes attrs;
attrs.id_ = gce::atom(“node_two”);
gce::context ctx(attrs); /// 创建一个id为“node_two”的gce::context
…
void func(gce::actor<gce::stackful>& self)
{
  /// 使用tcp连接本地14923端口、id为“router”的gce::context，true表示目标是路由节点
  gce::connect(self, gce::atom(“router”), “tcp://127.0.0.1:14923”, true);
  /// 之后node_one和node_two虽然没有直接连接，但也可以直接通讯了，
  /// 它们交互的消息均经过router中转
}
```

net_option

可以使用gce::net_option（gce/actor/net_option.hpp）来配置bind和connect的网络相关参数：

```
gce::net_option opt;
/// 心跳周期，最小单位秒
opt.heartbeat_period_ = boost::chrono::seconds(15);
/// 心跳超时次数上限，超过这个值，判定为连接错误
```

```
opt.heartbeat_count_ = 5;
gce::bind(self, “tcp://127.0.0.1:14923”, false, opt);
…
/// 连接错误后，重连之间的间隔时间，最小单位秒
opt.reconn_period_ = boost::chrono::seconds(10);
/// 重连尝试次数，超过这个值后，会发送 gce::exit 消息；然后重复尝试这个次数；
opt.reconn_try_ = 3;
/// 初始连接时，连接错误后，重连之间的间隔时间，最小单位秒
opt.init_reconn_period_ = boost::chrono::seconds(10);
/// 初始连接时，重连尝试次数；
opt.init_reconn_try_ = 3;
gce::connect(self, “tcp://127.0.0.1:14923”, false, opt);
```

重连策略

当内部心跳超时（net_option::heartbeat_period_）并且超过心跳超时次数（net_option::heartbeat_count_）后，这个连接即被判定为断开。

接着，gce 在 connect 端会尝试重连，而 bind 端会简单的销毁这个 socket。当重连失败后，会等待一段时间（net_option::reconn_period_）之后再进行下次重连，直到超过重连次数（net_option::reconn_try_）；

超过重连次数后，会对所有和这个连接有关系的 actor 发送 gce::exit 消息，然后继续循环上一段进行的重连，直到成功连接或者程序结束；

只有第一次连接的时候，才会使用 net_option::init_reconn_period_和 net_option::init_reconn_try_来施行重连。

基于 **erlang** 的错误处理模型

gce 使用 erlang 的 actor 错误处理模型。

actor 模型鼓励 actor 之间的相互链接，成为树形层次结构，分层管理。

一般处于非叶节点的 actor 都是监察者 actor，专门用于监视它的叶节点 actor 的情况，如果某个叶节点 actor 因为各种原因而退出（例如程序错误而挂掉），它会收到 gce::exit 消息及其退出原因等描述，用户可以根据情况而对应做不同的处理（例如重启）。

目前 gce 这部分只有基本的 actor 之间的链接处理，类似 erlang otp（erlang 的库资源平台）中的监察者等还未实现；但这并不妨碍用户自己使用 actor 的链接关系自己处理这种树形层级监视管理的方式。具体的示例代码可以见：libs/actor/example/cluster 中 conn、gate 等 actor 的关系处理。

与 **Boost.Asio** 的无缝结合

gce 的 actor 可以和 Boost.Asio 的一切 io 对象（socket、timer、signal_set、serial_port 等）无缝结合使用，无论哪种 actor。

这里主要分为两类：

基于线程的 actor 和无阻塞 actor，这两种 actor 适合和 Boost.Asio 的 io 对象的同步方式结合使用；

基于有栈和无栈协程的 actor，这两种 actor 适合和 Boost.Asio 的 io 对象的异步+协程的方式结合使用；

第一种在实际应用中很少见，因为 Boost.Asio 的同步使用方式从性能到扩展性都有较大限制（这不是 Boost.Asio 的问题，而是基于线程的同步方式的问题），因次我们重点介绍基于有栈和无栈协程的 actor 和 Boost.Asio 的结合使用。

基于有栈协程的 ***actor*** 和 ***Boost.Asio*** 的结合

由于基于有栈协程的 actor 的内部直接使用的 Boost.Asio 封装的 Boost.Coroutine，因此结合起来非常方便：

```
void func(gce::actor<gce::stackful>& self)
{
  try
  {
    /// 取得 Boost.Asio 的有栈协程的 yield_context，用于上下文切换
    gce::yield_t yield = self.get_yield();
    /// 取得所在的 gce::context 内部的唯一的 boost::asio::io_serivce
    gce::context* ctx = self.get_context();
    boost::asio::io_service& ios = ctx->get_io_service();

    /// 构建一个 boost::asio::system_timer，并计时 5 秒
    boost::asio::system_timer tmr(ios);
    tmr.expires_from_now(boost::chrono::seconds(5));
    tmr.async_wait(yield);
    /// 过去了 5 秒，和 self.wait(boost::chrono::seconds(5));的作用完全一样
  }
  catch (std::exception& ex)
  {
    /// tmr 计时过程中发生错误
  }
}
```

更加详细的代码示例，请见 libs/actor/example/asio。

基于无栈协程的 ***actor*** 和 ***Boost.Asio*** 的结合

基于无栈协程的 actor 使用 gce::adaptor 来处理和 Boost.Asio 的结合。

```
class my_stackless_actor
{
public:
  explicit my_stackless_actor(gce::context& ctx)
    : tmr_(ctx.get_io_service())
  {
  }
public:
  void run(gce::actor<gce::stackless>& self)
  {
    GCE_REENTER (self)
    {
      tmr_.expires_from_now(boost::chrono::seconds(5));
      GCE_YIELD tmr_.async_wait(gce::adaptor(self, ec_));
      if (!ec_)
      {
        /// 过去了5秒，
        /// 和GCE_YIELD self.wait(boost::chrono::seconds(5));的作用完全一样
      }
      else
      {
        /// tmr计时过程中发生错误
      }
    }
  }
private:
   boost::asio::system_timer tmr_;
   gce::errcode_t ec_;
};
```

更加详细的代码示例，请见 libs/actor/example/cluster/conn.cpp

附录

编译 ***Boost***

- 下载 boost；
- 解压缩到本地，比如 Boost 1.55 解压后产生 boost_1_55_0 目录；
- 进入根目录下的 tools\build\v2 目录下；

- 运行 bootstrap.sh/bat；
- windows 下你可以将生成的 b2.exe 和 bjam.exe 拷贝到你想要的地方，然后把路径加入环境变量 Path 中；linux 下你可以用 b2 install --prefix=安装目录 来安装到你需要的地方，如果有 root 权限，推荐安装到/usr 下，这样无需加入环境变量 Path 中了；
- 回到 Boost 根目录（比如 boost_1_55_0）；
- 使用命令编译 boost：b2 toolset=msvc-11.0 link=static variant=debug,release threading=multi runtime-link=shared --layout=versioned --with-atomic --with-chrono --with-coroutine --with-context --with-date_time --with-exception --with-program_options --with-regex --with-system --with-test --with-thread --with-timer stage
- 上述命令中， toolset=msvc-11.0 可以替换为对应的编译器，比如 gcc、msvc-9.0 等；
- 等待编译完成；
- 完成后，编译好的库，在根目录下的 stage/lib 下