

Juggle 说明文档

吴铭

2015/2/2

Juggle 说明文档

JUGGLE 说明文档.....	2
JUGGLE 简介.....	2
JUGGLE 使用说明.....	2
JUGGLE 设计文档.....	5
使用 UUID 确保消息的唯一性.....	6
使用内存池管理 SERVICE.....	6
使用无锁队列优化多线程中的消息传递.....	6
使用协程实现同步非阻塞接口.....	7
使用 CODEGEN 提高工作效率.....	7

Juggle 简介

Juggle 是一个基于 dsl 语言的可配置网络数据封包协议的 rpc 框架，基于 codegen 提供了 c++ 的服务器构建工具。

同类产品：

Protobuf: <https://code.google.com/p/protobuf/>

Thrift: <http://thrift.apache.org/>

Juggle 使用说明

Juggle dsl 是强类型的中间语言，通过模板，支持了仅原生类型的泛型。

Juggle dsl 关键字：

`module` 对应 c++ 中的 `class`，用于定义一个供客户端访问的服务

`array` 对应 c++ 中的 `std::vector`

`struct` 对应 c++ 中的 `class`，用于用户自定义数据

`string` 字符串，对应 c++ 中的 `std::string`

`float` 浮点数，对应 c++ 中的 `double`

`int` 整数，对应 c++ 中的 `int64_t`

`bool` 布尔值，对应 c++ 中的 `bool`

juggle 的一个简单例子：

```
module juggle{  
    string login(string argv3);  
    string test(string argv3);  
  
}
```

使用 `juggle rpcmake` 生成代码

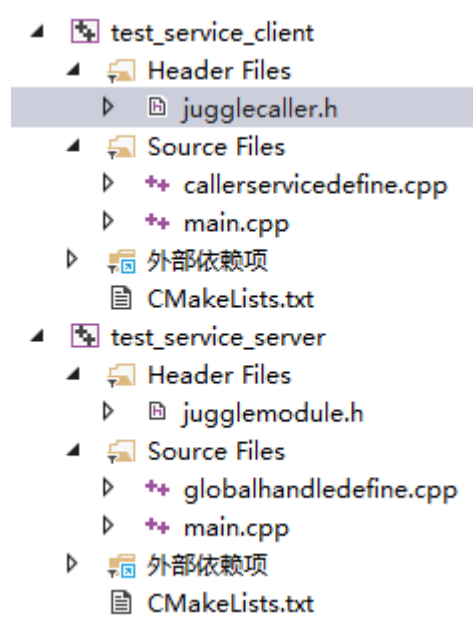
```
F:\workspace\fossilizid\test\test_service\juggle_scprit>python ../../juggle\rpcmake\rpcmake.py ./ ./ Fossilizid::jsonplugin::object json_plugin.h
```

参数分别为juggle 代码目录，生成代码目标目录，协议封包对象，协议封包对象头文件

生成代码如下：

名称	修改日期	类型	大小
client	2015/2/2 11:09	文件夹	
server	2015/1/30 15:03	文件夹	
juggle.juggle	2015/1/29 20:42	JUGGLE 文件	1 KB

之后即可基于 juggle 生成的代码，创建服务器工程



客户端直接使用生成的远程调用对象即可

```
sync::juggle j(ch.get());
```

```
std::cout << j.login("i am login").c_str() << std::endl;
```

```
std::cout << j.test("i am test").c_str() << std::endl;
```

服务器端生成了一组协议响应代码，并且定义了 rpc 接口的虚函数以及创建 module 的

```
class juggle: public Fossilizid::juggle::module{
public:
    + juggle() -: module("juggle", -Fossilizid::uuid::UUID()){
    +     Fossilizid::juggle::_service_handle->register_module_method("juggle_login", -boost::bind(&juggle::call_login, -this, _1, _2));
    +     Fossilizid::juggle::_service_handle->register_module_method("juggle_test", -boost::bind(&juggle::call_test, -this, _1, _2));
    + }
    + ~juggle() {
    + }

    + virtual std::string login(std::string argv3) -: 0;
    + void call_login(Fossilizid::juggle::channel *ch, -boost::shared_ptr<Fossilizid::juggle::object> v) {
    +     + auto argv3 = - (*v) ["argv3"].asstring();
    +     + auto ret = - login(argv3);
    +     + boost::shared_ptr<Fossilizid::juggle::object> r = - boost::make_shared<Fossilizid::jsonplugin::object>();
    +     + (*r) ["suuid"] = - (*v) ["suuid"].asstring();
    +     + (*r) ["method"] = - (*v) ["method"].asstring();
    +     + (*r) ["rpcevent"] = - "reply_rpc_method";

    +     + (*r) ["ret"] = - ret;
    +     + ch->push(r);
    + }

    + virtual std::string test(std::string argv3) -: 0;
    + void call_test(Fossilizid::juggle::channel *ch, -boost::shared_ptr<Fossilizid::juggle::object> v) {
    +     + auto argv3 = - (*v) ["argv3"].asstring();
    +     + auto ret = - test(argv3);
    +     + boost::shared_ptr<Fossilizid::juggle::object> r = - boost::make_shared<Fossilizid::jsonplugin::object>();
    +     + (*r) ["suuid"] = - (*v) ["suuid"].asstring();
    +     + (*r) ["method"] = - (*v) ["method"].asstring();
    +     + (*r) ["rpcevent"] = - "reply_rpc_method";

    +     + (*r) ["ret"] = - ret;
    +     + ch->push(r);
    + }
};
juggle* create_juggle();
```

用户需要实现对应的接口即可

```
class juggleimpl: public juggle{
```

```
public:
```

```
virtual std::string login(std::string argv3){
```

```
    printf("juggleimpl login %s\n", argv3.c_str());
```

```
    return "login sucess";
```

```
}
```

```
virtual std::string test(std::string argv3) {
```

```
    printf("juggleimpl test %s\n", argv3.c_str());
```

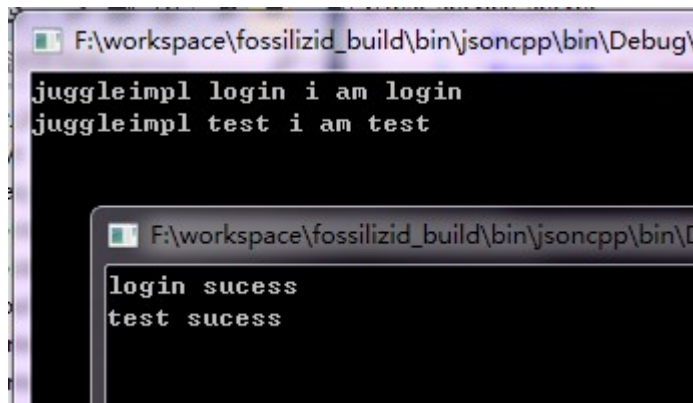
```
    return "test sucess";
```

```
}
```

```
};
```

```
juggle * create_juggle() {  
  
    return new juggleimpl();  
  
}
```

测试用例执行如下:



The image shows two overlapping command prompt windows. The top window has a title bar with the path 'F:\workspace\foosilizid_build\bin\jsoncpp\bin\Debug\' and contains the following text: 'juggleimpl login i am login' and 'juggleimpl test i am test'. The bottom window has a title bar with the path 'F:\workspace\foosilizid_build\bin\jsoncpp\bin\' and contains the following text: 'login sucess' and 'test sucess'.

```
F:\workspace\foosilizid_build\bin\jsoncpp\bin\Debug\  
juggleimpl login i am login  
juggleimpl test i am test  
  
F:\workspace\foosilizid_build\bin\jsoncpp\bin\  
login sucess  
test sucess
```

Juggle 设计文档

Juggle 基于协程和函数对象设计了清晰的同步非阻塞访问接口，并且在设计上实现了通信层和协议层、消息处理的解耦合。

网络层部分 juggle 定义了一组最简的通信接口：

```
class channel::public base::channel{
public:
    /*
    * push a object to channel
    */
    virtual void push(boost::shared_ptr<object> v) = 0;

    /*
    * get a object from channel
    */
    virtual boost::shared_ptr<object> pop() = 0;
```

和一组注册 channel 到事务层的接口

```
class channel{
public:
    /*
    * create a new channel
    */
    void handle_new_channel();

    /*
    * release the channel
    */
    void handle_disconnect_channel();
};
```


协议层部分 **juggle** 定义了一个虚基类的打包接口定义,用户需要基于 **juggle** 给出的接口实现自己的打包协议:

```

class object{
public:
→   /* ... */
→   virtual bool empty() const = 0;

→   /* ... */
→   virtual void clear() const = 0;

→   /* ... */
→   virtual bool asbool() const = 0;
→   virtual int64_t asint() const = 0;
→   virtual double asfloat() const = 0;
→   virtual std::string asstring() const = 0;

→   /* ... */
→   virtual bool isnull() const = 0;
→   virtual bool isbool() const = 0;
→   virtual bool isint() const = 0;
→   virtual bool isfloat() const = 0;
→   virtual bool isstring() const = 0;
→   virtual bool isarray() const = 0;
→   virtual bool ismap() const = 0;

→   /* ... */
→   virtual size_t size() const = 0;
→
→   /* ... */
→   virtual object &operator[] (int index) = 0;
→   virtual const object &operator[] (int index) const = 0;
→   virtual object &append(const bool &other) = 0;
→   virtual object &append(const int64_t &other) = 0;
→   virtual object &append(const double &other) = 0;
→   virtual object &append(const std::string &other) = 0;
→   virtual bool erase(const int index) = 0;

→   /* ... */
→   virtual bool hasfield(const std::string &key) const = 0;
→   virtual object &operator[] (const std::string &key) = 0;
→   virtual const object &operator[] (const std::string &key) const = 0;
→   virtual bool erase(const std::string &key) = 0;

→   /* ... */
→   virtual void operator=(const bool &other) = 0;
→   virtual void operator=(const int64_t &other) = 0;
→   virtual void operator=(const double &other) = 0;
→   virtual void operator=(const std::string &other) = 0;
→   virtual void operator=(const char *other) = 0;

};

```

事务层 `juggle` 完成了一个单线程的整个事务处理的全部流程，玩家只需要定义 `juggle service` 的接口，并且调用对应的初始化接口和驱动接口，完成整个事件

循环，然后在此基础上编写消息响应代码即可。

```
class service{
public:
|→  /*
|→   *--initialise service
|→   */
|→  virtual void init() = 0;

|→  /*
|→   *--drive service work
|→   */
|→  virtual void poll() = 0;

};

boost::shared_ptr<service> create_service() {

class juggleimpl::public juggle{
public:
|→  virtual std::string login(std::string argv3) {
|→   → printf("juggleimpl-login-%s\n", argv3.c_str());

|→   → return "login-sucess";
|→  }

|→  virtual std::string test(std::string argv3) {
|→   → printf("juggleimpl-test-%s\n", argv3.c_str());

|→   → return "test-sucess";
|→  }

};

juggle* create_juggle() {
|→  return new juggleimpl();
}

int main() {
|→  Fossilizid::reduce::acceptor::channelserver::server(Fossilizid::juggle::create_service());

|→  server.init("127.0.0.1", -1234);

|→  while (1) {
|→   → server.poll();
|→  }

|→  return 1;
}
```

使用 uuid 确保消息的唯一性

UUID 是一种通过算法来确保在一个有限的范围内分布式唯一的技术。

在 juggle 中，为每条消息生成了 uuid，并且通过 uuid 为消息注册自己的回调函数，以此来实现了对同一类事件即可同步访问亦可异步访问。

```
boost::shared_ptr<object> caller::call_module_method_sync(  
→ std::string methodname, boost::shared_ptr<object> value)  
{  
→ semaphore s;  
→ call_module_method_async(methodname, value, boost::bind(&semaphore::post, &s, _1));  
→ return s.wait();  
}  
  
void caller::call_module_method_async(std::string methodname,   
→ boost::shared_ptr<object> value, boost::function<void(boost::shared_ptr<object>)> callback)  
{  
→ (*value)["method"] = methodname;  
→ (*value)["suuid"] = uuid::UUID();  
→ (*value)["rpcevent"] = "call_rpc_method";  
  
→ boost::static_pointer_cast<juggleservice>(  
→ → _service_handle->register_rpc_callback((*value)["suuid"].asstring(), callback);  
→  
→ _ch->push(value);  
}
```

使用内存池管理 service

Fossilizid 提供了接口优雅的内存池，juggle 以此配合 `shared_ptr` 来管理 service 中的对象。

```
class factory{
public:
    factory();
    ~factory();

public:
    /* ... */
    template<class T, typename... Tlist>
    static T* create(int count, Tlist&&... var) {
        T* p = (T*)mempool::allocator(count * sizeof(T));

        for (int i = 0; i < count; i++) {
            new (&p[i]) T(std::forward<Tlist>(var)...);
        }

        return p;
    }

    /* ... */
    template<class T, typename... Tlist>
    static T* create(Tlist&&... var) {
        T* p = (T*)mempool::allocator(sizeof(T));

        new (p) T(std::forward<Tlist>(var)...);

        return p;
    }

    /* ... */
    template<class T>
    static void release(T* p, int count) {
        for (int i = 0; i < count; i++) {
            p[i].~T();
        }

        mempool::dealloc(p, count * sizeof(T));
    }
};
```

使用无锁队列优化多线程中的消息传递

无锁数据结构主要用于为多线程环境提供一个可以高效访问的容器。

代表性的有 CDS: <http://libcds.sourceforge.net/>

michael 在这个领域的论文:<http://www.research.ibm.com/people/m/michael/>

Fossilizid 提供了几组简单的无锁队列，并且用于网络层的消息传递

```
class channel : public juggle::channel {
public:
    channel(remoteq::CHANNEL _ch) {
        ch = _ch;
    }

    ~channel() {
    }

    /* ... */
    virtual void push(boost::shared_ptr<juggle::object> v) {
        remoteq::push(ch, *v, jsonplugin::object_to_buf);
    }

    /* ... */
    virtual boost::shared_ptr<juggle::object> pop() {
        boost::shared_ptr<juggle::object> v = 0;
        if (que.pop(v)) {
            return v;
        }
        return 0;
    }

public:
    void pushcmd(boost::shared_ptr<juggle::object> v) {
        que.push(v);
    }

private:
    remoteq::CHANNEL ch;
    container::msque<boost::shared_ptr<juggle::object>> que;
};
```

使用协程实现同步非阻塞接口

一般认识上远程访问一般是异步的，因为对一个远端的请求之后，客户端需要等待服务器的返回。有次引出了消息响应，状态保存等一系列策略。

但是基于协程有用户层控制切换的特性，我们可以在提出一个请求后将此协程挂起，并且将此消息的 `uuid` 注册到协程管理器，然后等待远端服务器的返回，收到返回消息后调度执行此协程。

```
boost::shared_ptr<object> caller::call_module_method_sync(std::string methodname, boost::shared_ptr<object> value) {  
    + semaphore s;  
    + call_module_method_async(methodname, value, boost::bind(&semaphore::post, &s, _1));  
    + return s.wait();  
}
```

得益于 C++ 提供的函数对象等机制。

我们只需为此消息注册特殊的回调函数即可。

见用例，可以看见我们对远端的访问是同步的。

```
+ Fossilizid::reduce::connect::channelserver::server(Fossilizid::juggle::create_service());  
+  
+ server.init();  
  
+ boost::shared_ptr<Fossilizid::juggle::channel> ch = server.connect("127.0.0.1", -1234);  
  
+ sync::juggle j(ch.get());  
  
+ std::cout << j.login("i-am-login").c_str() << std::endl;  
+ std::cout << j.test("i-am-test").c_str() << std::endl;  
  
+ while(1) {  
+     server.poll();  
+ }
```

使用 codegen 提高工作效率

一般意义上的消息响应代码都是趋同的：

注册消息响应函数->

消息收发->

消息 unpack->

调用消息响应函数

由此我们可以编写一些代码生成工具，以此完成这些重复的工作。这亦是 **juggle** 中最核心的部分。

通过对 **juggle** 脚本的分析，**rpcmake** 就会自动生成对应的代码

关于代码生成的说明：http://km.netease.com/kp_blog/view?article_id=165054