

MOOON-agent 系统设计与使用说明

易剑 2012/6/16

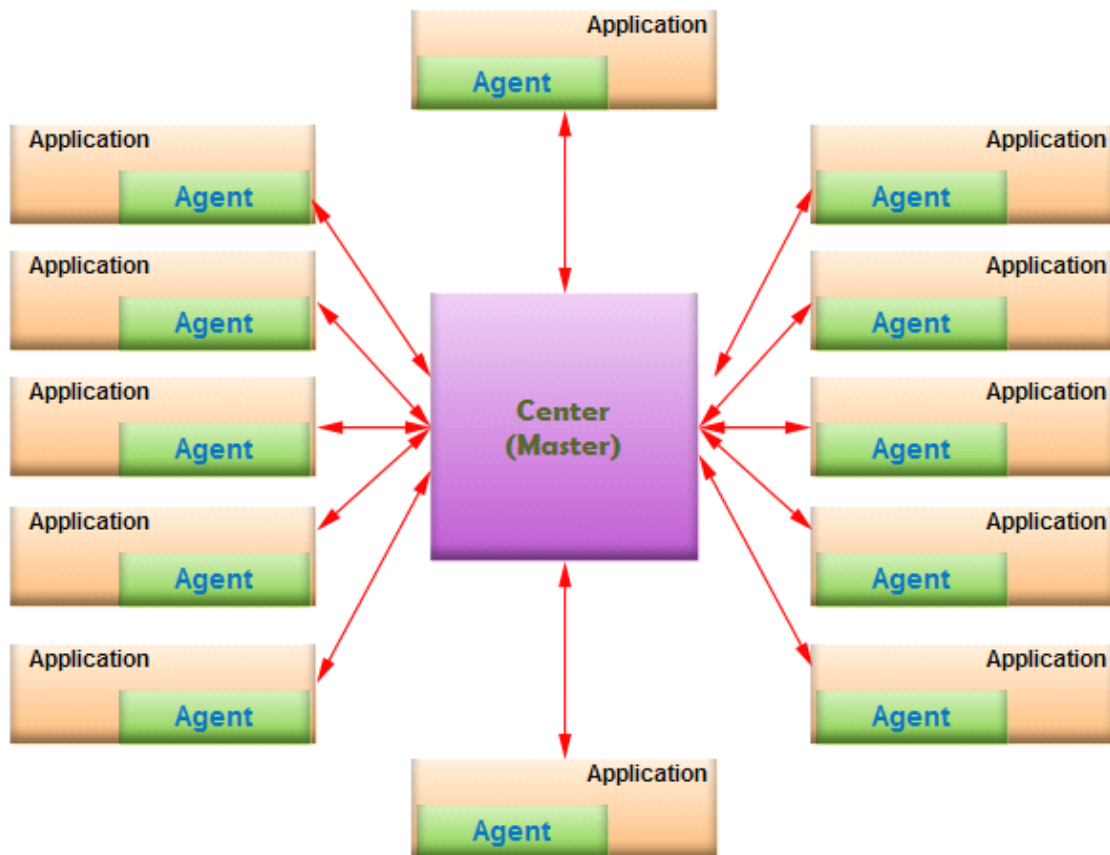
目录

1. 设计目标.....	1
2. 应用场景.....	2
3. 主要功能.....	2
4. 系统骨架.....	3
5. 资源接口.....	3
6. 内置 CommandProcessor.....	3
7. 编程接口.....	3
7.1. agent.h.....	4
7.2. message.h.....	5
7.3. message_command.h.....	5
7.4. command_processor.h.....	6
8. 编程示例.....	7

1. 设计目标

一个通用的 agent 框架，提供编程接口，并内置通用的功能。

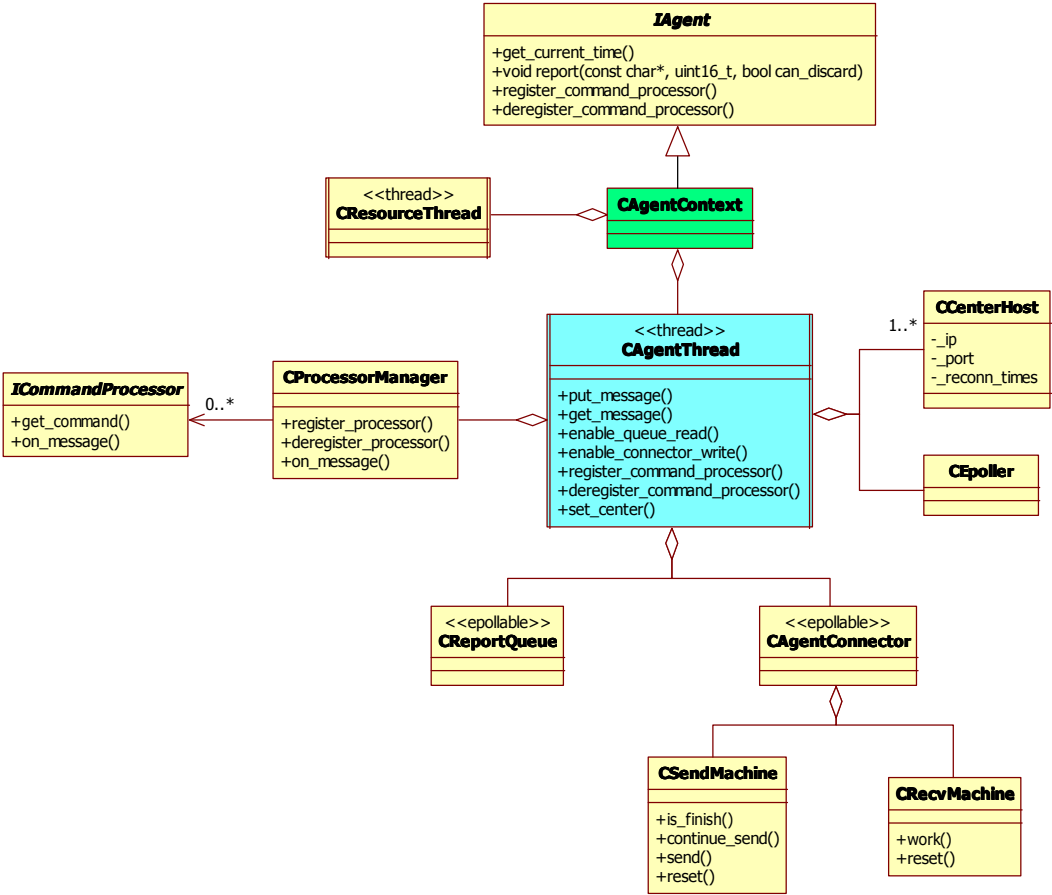
2. 应用场景



3. 主要功能

- 1) 自动上报心跳
- 2) 支持以域名方式指定 center 或者 IP 列表方式指定 center
- 3) 与 center 断开后自动重连接
- 4) 支持多种重连接 center 策略，如轮询方式
- 5) 自动上报数据到 center
- 6) 可选多种心跳方式，简单心跳不附带数据，富心跳可携带系统状态数据，如 CPU 使用率、内存使用情况等
- 7) 提供获取 CPU 使用率、内存使用情况和流量等接口
- 8) 内置配置等 CommandProcessor，常用需求不用编程直接使用
- 9) 非单例，单个进程可创建多个 agent 实例

4. 系统骨架



5. 资源接口

暂略。

6. 内置 CommandProcessor

暂略。

7. 编程接口

除宏外，所以内容均位于 agent 名字空间内。

7.1.agent.h

```

/**
 * 常量定义
 */
enum
{
    REPORT_MAX = 10240 /** 一次report的最大字节数 */
};

class IAgent
{
public:
    virtual ~IAgent() {}
    virtual void set_center(const std::string& domainname_or_iplist,
uint16_t port) = 0;
    virtual bool report(const char* data, size_t data_size, bool
can_discard=true) = 0;
    virtual bool report(const char* format, ...) = 0;
    virtual bool register_command_processor(ICommandProcessor*
processor) = 0;
    virtual void deregister_command_processor(ICommandProcessor*
processor) = 0;
};

/**
 * 日志器，所以分发器实例共享
 * 如需要记录日志，则在调用create之前，应当先设置好日志器
 */
extern sys::ILogger* logger;

/**
 * 用来创建agent实例，注意agent不是单例，允许一个进程内有多个实例
 * @queue_size 上报队列大小，如果队列满，会导致消息丢失或report调用阻塞
 * @connect_timeout_milliseconds 与center连接的超时毫秒数，如果在这个时
间内没有数据上报，
 *                                     则会自动发送心跳消息，否则不会发送心跳消息
 */
extern IAgent* create(uint32_t queue_size, uint32_t
connect_timeout_milliseconds);

/** 销毁一个agent实例 */
extern void destroy(IAgent* agent);

```

7.2.message.h

```
#pragma pack(4) // 网络消息按4字节对齐

/**
 * Agent消息头
 */
typedef struct TAgentMessageHeader
{
    NUInt32 size;    /** 消息包字节数 */
    NUInt32 command; /** 消息的命令字 */
}agent_message_header_t;

/**
 * 简单的心跳消息，仅一个消息头
 */
typedef struct TSimpleHeartbeatMessage
{
    agent_message_header_t header;
}simple_heartbeat_message_t;

/**
 * 上报消息
 */
typedef struct TReportMessage
{
    agent_message_header_t header;
    char data[0]; /** 需要上报的内容 */
}report_message_t;

#pragma pack()
```

7.3.message_command.h

```
/**
 * 上行消息命令字
 */
typedef enum TUplinkMessageCommand
{
    U_SIMPLE_HEARTBEAT_MESSAGE = 1, /** 简单心跳消息 */
    U_REPORT_MESSAGE           = 2  /** 上报消息 */
}
```

```

}uplink_message_command_t;

/**
 * 下行消息命令字，由ICommandProcessor处理
 */
typedef enum TDownlinkMessageCommand
{

}downlink_message_command_t;

```

7.4.command_processor.h

```

/**
 * 消息上下文结构
 * 由于是异步接收消息的，所以需要有一个上下文结构来保存最新状态
 */
typedef struct TMessageContext
{
    size_t total_size;    /** 消息体的字节数 */
    size_t finished_size; /** 已经收到的消息体字节数 */

    TMessageContext(size_t total_size_, size_t finished_size_)
        :total_size(total_size_)
        ,finished_size(finished_size_)
    {
    }
}message_context_t;

class ICommandProcessor
{
public:
    virtual ~ICommandProcessor() {}

    /**
     * 返回该CommandProcessor处理的命令字
     */
    virtual uint32_t get_command() const = 0;

    /**
     * 有消息需要处理时的回调函数
     * 请注意消息的接收是异步的，每收到一点消息数据，都会回调on_message
     * 整个消息包接收完成的条件是 msg_ctx.total_size 和
     msg_ctx.finished_size+buffer_size两者相等

```

```

    * @buffer 当前收到的消息体数据
    * @buffer_size 当前收到的消息体数据字节数
    * @return 如果消息处理成功，则返回true，否则返回false，当返回false
    时，会导致连接被断开进行重连接
    */
    virtual bool on_message(const TMessageContext& msg_ctx, const char*
buffer, size_t buffer_size) = 0;
};

```

8. 编程示例

```

// 命令字1的CommandProcessor
class CCommandProcessor1: public ICommandProcessor
{
private:
    virtual uint32_t get_command() const
    {
        return 1;
    }

    virtual bool on_message(const TMessageContext& msg_ctx, const char*
buffer, size_t buffer_size)
    {
        fprintf(stdout, "[%zu:%zu] %.s\n", msg_ctx.total_size,
msg_ctx.finished_size, (int)buffer_size, buffer);
        return true;
    }
};

```

```

// 命令字2的CommandProcessor
class CCommandProcessor2: public CCommandProcessor1
{
private:
    virtual uint32_t get_command() const
    {
        return 2;
    }
};

```

```

// 命令字3的CommandProcessor
class CCommandProcessor3: public CCommandProcessor1
{
private:

```

```

    virtual uint32_t get_command() const
    {
        return 3;
    }
};

class CMainHelper: public sys::IMainHelper
{
public:
    CMainHelper()
        : _agent(NULL)
    {
    }

private:
    virtual bool init(int argc, char* argv[])
    {
        uint32_t queue_size = 100;
        uint32_t connect_timeout_milliseconds = 2000;

        _agent = agent::create(queue_size,
connect_timeout_milliseconds);
        if (NULL == _agent)
        {
            return false;
        }

        _agent->register_command_processor(&_command_processor1);
        _agent->register_command_processor(&_command_processor2);
        _agent->register_command_processor(&_command_processor3);

        _agent->set_center(ArgsParser::center_ip->get_value(),
                        ArgsParser::center_port->get_value());

        std::string report("test");
        while (true)
        {
            sys::CUtil::millisleep(3000);
            _agent->report(report.data(), report.size());
        }

        return true;
    }
}

```



```

virtual void fini()
{
    agent::destroy(_agent);
    _agent = NULL;
}

virtual int get_exit_signal() const
{
    return SIGTERM;
}

private:
    agent::IAgent* _agent;
    CCommandProcessor1 _command_processor1;
    CCommandProcessor2 _command_processor2;
    CCommandProcessor3 _command_processor3;
};

// 入口函数
extern "C" int main(int argc, char* argv[])
{
    if (!ArgsParser::parse(argc, argv))
    {
        fprintf(stderr, "Args error: %s.\n",
ArgsParser::g_error_message.c_str());
        exit(1);
    }

    CMainHelper main_helper;
    return sys::main_template(&main_helper, argc, argv);
}

```