C++ 服务器案例学习笔记

郑华

2018 年 2 月 6 日

目录

第一章 开源-muduo-1.1.2 **5**

1.1 架构图 . . . 5
1.2 参考 . . . 5
1.3 基础类设计 . . . 5
1.3.1 Thread . . . 5
1.3.2 MutexLock . . . 8
1.3.3 Condition . . . 8
1.3.4 CountDownLatch . . . 8
1.3.5 BlockingQueue 与 BoundedBlockingQueue . . . 9
1.3.6 ThreadPool . . . 11
1.3.7 Singleton . . . 11
1.3.8 线程特定数据 . . . 12
1.3.9 日志 . . . 12
1.4 网络类设计 . . . 13

第二章 简单 http 服务器 **15**

2.1 架构图 . . . 15
2.2 http 协议 . . . 15
2.2.1 请求服务器数据 . . . 15
2.2.2 服务器应答浏览器 . . . 16
2.3 HTML . . . 16
2.4 xinetd . . . 16
2.4.1 守护进程 . . . 16
2.4.2 配置 xinetd . . . 16

第三章 开源-libevent-1.4 **19**

3.1 参考 . . . 19

第四章 开源-lighthttpd-1.4.15 **21**

4.1 架构图 . . . 21

第五章 开源-nginx-0.5.38 **23**

5.1 架构 . . . 23
5.2 参考 . . . 23

第六章 游戏服务器 - 戴汉水 **25**

6.1 经验学习 . . . 25
6.1.1 #ifdef .. 编译选择 . . . 25
6.1.2 string 和 wstring 相互转换 . . . 25
6.1.3 GBK 和 UTF8 的相互变换 . . . 25

6.1.4 boost 智能指针 . . . 25
6.1.5 游戏服务器种类 . . . 25
6.1.6 enum 枚举的使用 . . . 25
6.1.7 按模块编码，而不是头文件与 cpp 文件 . . . 25
6.2 错误记录 . . . 26
6.2.1 错误 204 error LNK2001: 无法解析的外部符号 . . . 26

第七章 游戏服务器 2 - 戴汉水 **27**
7.1 经验 . . . 27
7.1.1 设置自己的解决方案目录结构 . . . 27
7.1.2 学会程序跟踪，与 Debug . . . 27
7.2 错误记录 . . . 27
7.2.1 Error MSB3073 代码 9009 . . . 27
7.2.2 Error fatal error LNK1104: cannot open file 'atlsd.lib' . . . 27
7.2.3 Project : error PRJ0019: 工具从" 正在执行预生成事件..." . . . 27
7.2.4 subwcrev 不是内部或外部命令，也不是可运行的程序 . . . 27
7.2.5 resource 文件目录如果打不开 . . . 27

第八章 游戏解决方案的搭建 **29**
8.1 DLL 动态库 . . . 29
8.2 调用 DLL . . . 29
8.3 调用 DLL 中的头文件，但其实不是 . . . 29

第九章 Others **31**
9.1 线程池 . . . 31
9.1.1 核心思想 . . . 31
9.1.2 组成部分 . . . 31
9.1.3 实现 . . . 32

第一章　开源-muduo-1.1.2

1.1　架构图

```
muduo
|-- build.sh
|-- ChangeLog
|-- CMakeLists.txt
|-- License
|-- README
|-- muduo              muduo 库的主体
|   |-- base           与网络无关的基础代码，位于 ::muduo namespace，包括线程库
|   \-- net            网络库，位于 ::muduo::net namespace
|       |-- poller     poll(2) 和 epoll(4) 两种 IO multiplexing 后端
|       |-- http       一个简单的可嵌入的 web 服务器
|       |-- inspect    基于以上 web 服务器的“窥探器”，用于报告进程的状态
|       \-- protorpc   简单实现 Google Protobuf RPC，不推荐使用
|-- examples           丰富的示例
\-- TODO
```

图 1.1: 架构图

1.2　参考

代码架构：见 muduo 参考手册，net 文件夹下

博客：http://www.cppblog.com/Solstice/default.html?page=3

https://www.oschina.net/question/28_61182

1.3　基础类设计

1.3.1　Thread

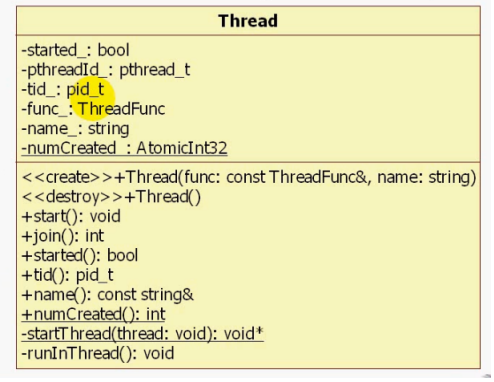

图 1.2: Thread 类图

Thread 头文件

```
namespace muduo
{

    class Thread : boost::noncopyable
    {
      public:
        typedef boost::function<void ()> ThreadFunc;

        explicit Thread(const ThreadFunc&, const string& name = string());
        #ifdef __GXX_EXPERIMENTAL_CXX0X__
        explicit Thread(ThreadFunc&&, const string& name = string());
        #endif
        ~Thread();

        void start();
        int join(); // return pthread_join()

        bool started() const { return started_; }

        pid_t tid() const { return *tid_; }
        const string& name() const { return name_; }

        static int numCreated() { return numCreated_.get(); }

      private:
        void setDefaultName();

        bool started_;
        bool joined_;
        pthread_t pthreadId_;
        boost::shared_ptr<pid_t> tid_;
        ThreadFunc func_;
        string name_;

        static AtomicInt32 numCreated_;
    };

}
```

要点

- 该线程类使用基于对象的方式实现，即注册回调函数来实现各对象不同的功能。
- 静态方法不能调用非静态，如 this
- 参数类型匹配，例如传递对象指针先得转化为 void*, 进去后然后再转回来。
- pthread_create 在线程创建以后，就开始运行相关的线程函数
- pthread_join 如果没有加pthread_join() 方法，main 线程里面直接就执行起走了，加了之后是等待线程执行了之后才执行的后面的代码。

http://blog.csdn.net/dinghqalex/article/details/42921931

回调图如下1.3.1：

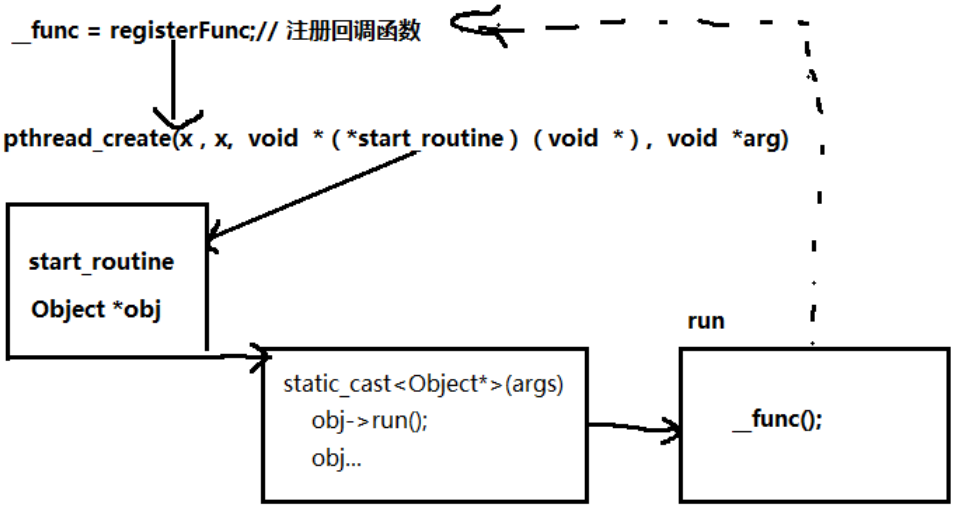

图 1.3: 回调演示

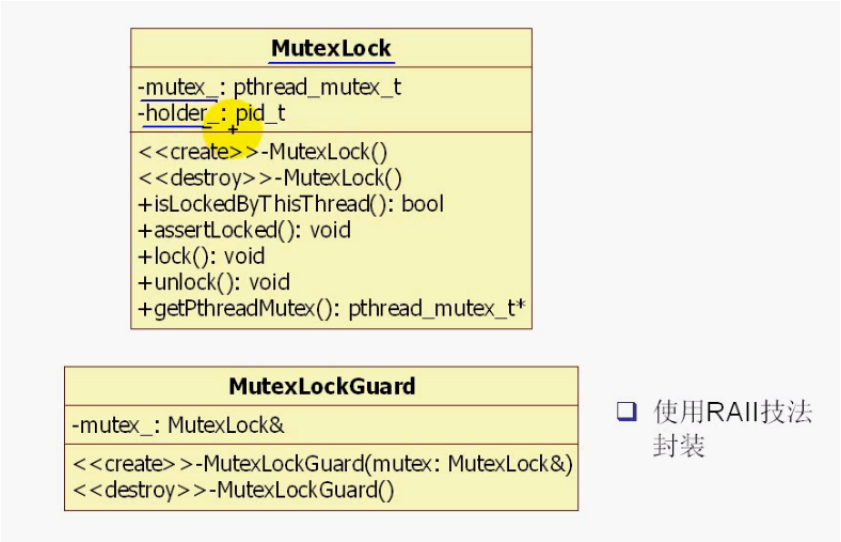

图 1.4: MutexLock 类图

1.3.2 MutexLock

要点

- 存在锁竞争
-

1.3.3 Condition

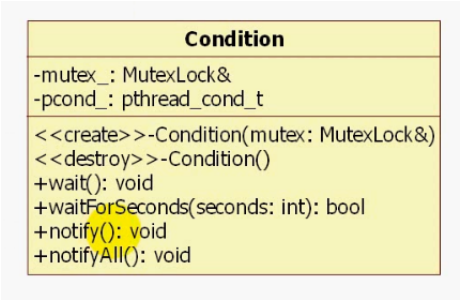

图 1.5: Condition 类图

具体流程

- 锁住 `mutex_lock`
- 等待条件满足

```
while()
{
    mutex_unlock
    等待条件
    mutex_lock
}
```

- 解锁 `mutex_unlock`

要点

- 观察者模式

1.3.4 CountDownLatch

可以用于所有子线程等待主线程发起“起跑”
可以用于主线程等待子线程初始化完毕才开始工作。

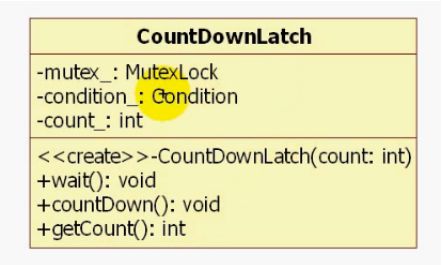

图 1.6: Condition 类图

1.3.5 BlockingQueue 与 BoundedBlockingQueue

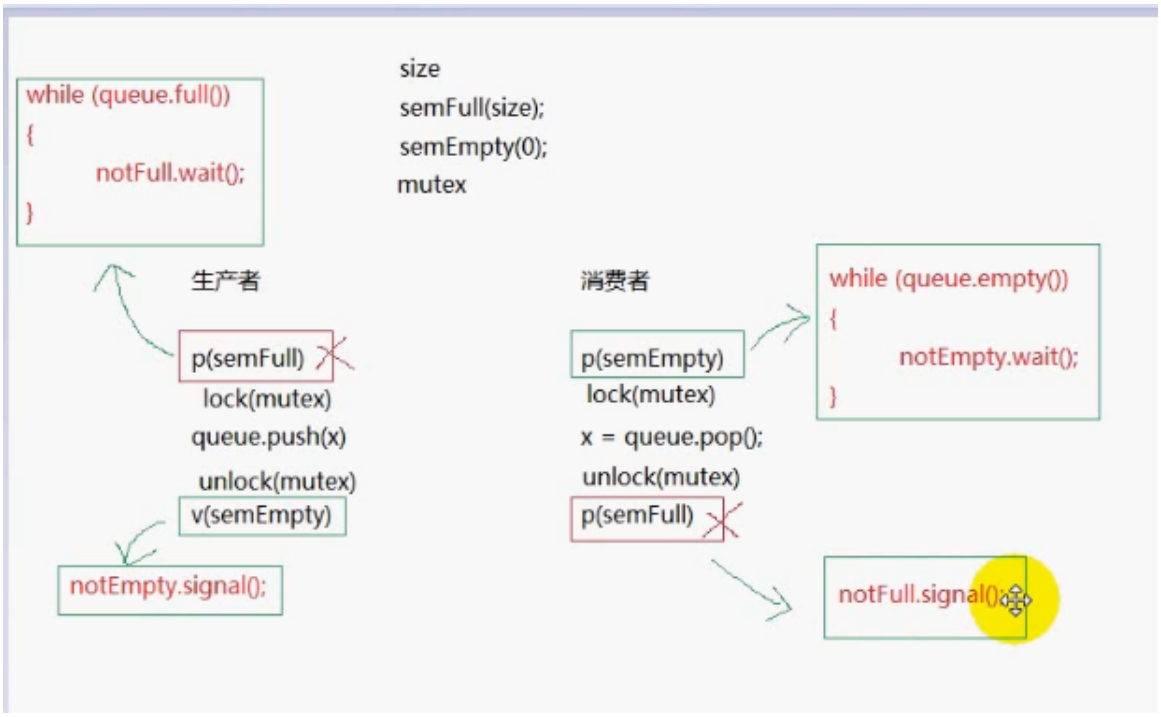

图 1.7: 有界队列

BlockingQueue<T>

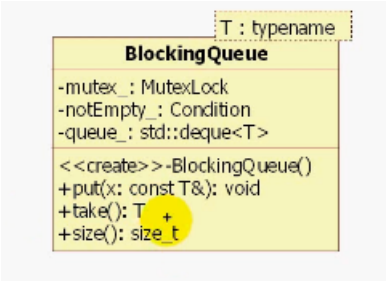

图 1.8: BlockingQueue 类图

BoundedBlockingQueue<T>

实现示例 ->

```
namespace muduo
{

    template<typename T>
    class BlockingQueue : boost::noncopyable
    {
        public:
        BlockingQueue()
        : mutex_(),
        notEmpty_(mutex_),
        queue_()
        {
```

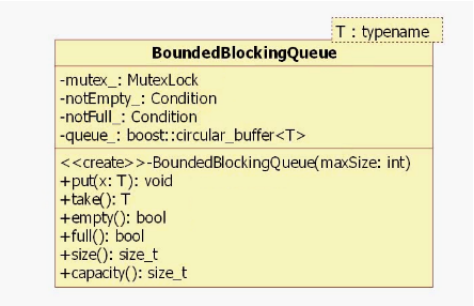

图 1.9: BlockingQueue

```
}

void put(const T& x)
{
    MutexLockGuard lock(mutex_);
    queue_.push_back(x);
    notEmpty_.notify(); // wait morphing saves us
    // http://www.domaigne.com/blog/computing/condvars-signal-with-mutex-locked-or-not/
}

#ifdef __GXX_EXPERIMENTAL_CXX0X__
void put(T&& x)
{
    MutexLockGuard lock(mutex_);
    queue_.push_back(std::move(x));
    notEmpty_.notify();
}
// FIXME: emplace()
#endif

T take()
{
    MutexLockGuard lock(mutex_);
    // always use a while-loop, due to spurious wakeup
    while (queue_.empty())
    {
        notEmpty_.wait();
    }
    assert(!queue_.empty());
    #ifdef __GXX_EXPERIMENTAL_CXX0X__
    T front(std::move(queue_.front()));
    #else
    T front(queue_.front());
    #endif
    queue_.pop_front();
    return front;
}

size_t size() const
{
    MutexLockGuard lock(mutex_);
    return queue_.size();
}

private:
mutable MutexLock mutex_;
```

```
        Condition notEmpty_;
        std::deque<T> queue_;
    };

}
```

核心要点

- 对于无界队列只需要将 notFull 条件变量去掉即可

1.3.6 ThreadPool

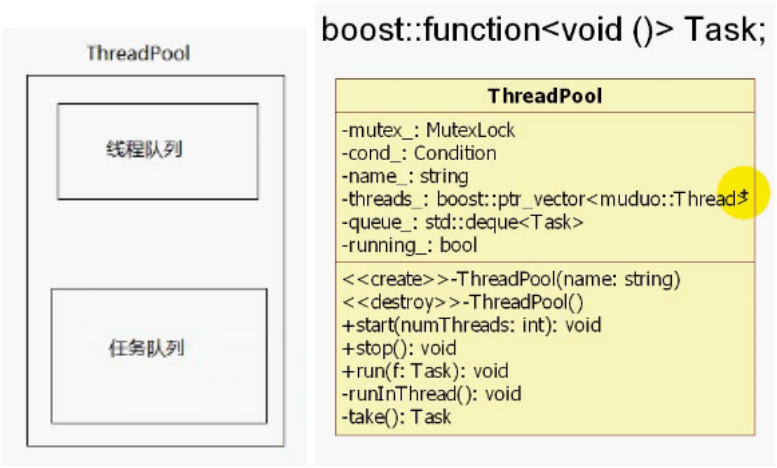

图 1.10: ThreadPool 模型

示例代码 https://github.com/ctzhenghua/muduo/blob/master/muduo/base/ThreadPool.cc
http://blog.csdn.net/u013507368/article/details/48130151

1.3.7 Singleton

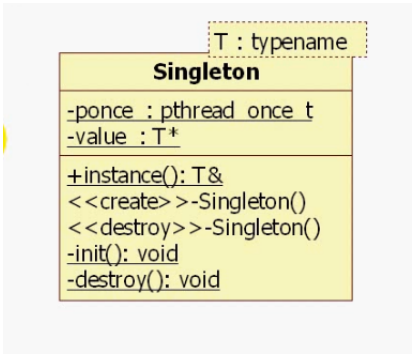

图 1.11: ThreadPool 模型

核心要点

- `pthread_once` 确定某个函数只调用一次
- `atexit` 确定对象销毁时执行的函数
- `typedef char T_must_be_complete_type[sizeof(T) == 0?-1,1]`

示例代码 https://github.com/ctzhenghua/muduo/blob/master/muduo/base/Singleton.h

1.3.8 线程特定数据

- 在单线程程序中，我们经常要用到”全局变量”以实现多个函数间共享数据
- 在多线程环境下，由于数据空间是共享的，因此全局变量也为所有线程所共有。
- 但有时应用程序设计中有必要提供线程私有的全局变量，仅在某个线程中有效，但却可以跨多个函数访问。
- POSIX 线程库通过维护一定的数据结构来解决这个问题.(如共享内存一般，创建 key, 然后使用，如下图)
- 线程特定数据也称为线程本地存储 Thread-local storage. 对于 POD 类型的线程本地存储，可以用`__thread`关键字

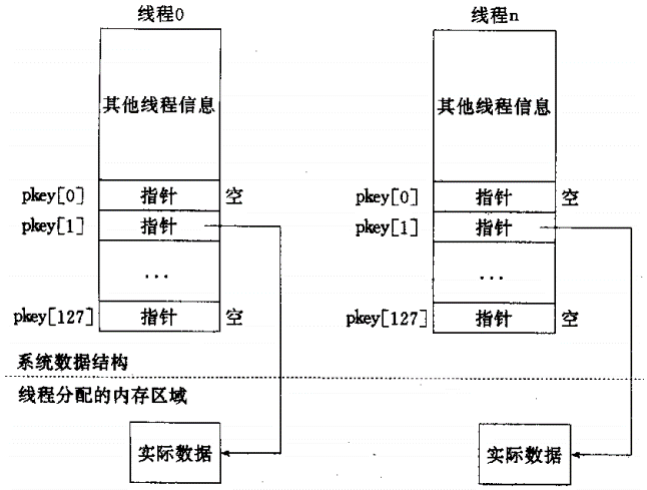

图 1.12: ThreadLocal 用法及模型

1.3.9 日志

作用

- 开发时：
  - 调试错误
  - 更加的理解程序
- 运行时：
  - 诊断系统故障并处理
  - 记录系统运行状态

日志级别

- `TRACE` 指出比`DEBUG` 粒度更细的一些信息事件（开发过程中使用）
- `DEBUG` 指出细粒度信息事件对调试应用程序是非常有帮助的。（开发过程中使用）
- `INFO` 表明消息在粗粒度级别上突出强调应用程序的运行过程
- `WARN` 系统能正常运行，但可能会出现潜在错误的情形。
- `ERROR` 指出虽然发生错误事件，但仍然不影响系统的继续运行。
- `FATAL` 指出每个严重的错误事件将会导致应用程序的退出

1.4 网络类设计

第二章　简单 http 服务器

2.1　架构图

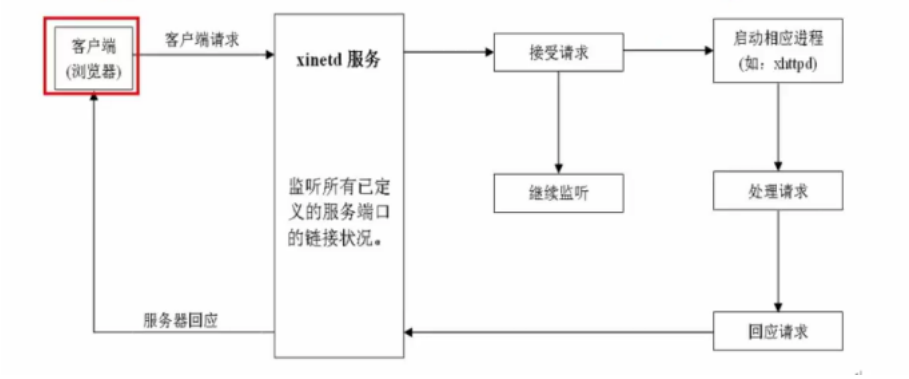

图 2.1: 架构图

2.2　http 协议

客户端请求的文件格式，https://www.cnblogs.com/lexiaofei/p/6943690.html

2.2.1　请求服务器数据

- HTTP请求方法
  - Http 协议定义了很多与服务器交互的方法，最基本的有 4 种，分别是GET,POST,PUT,DELETE.
  - 一个URL 地址用于描述一个网络上的资源，而 HTTP 中的GET, POST, PUT, DELETE 就对应着对这个资源的查，改，增，删 4 个操作。我们最常见的就是GET 和POST 了。GET 一般用于获取/查询资源信息，而POST 一般用于更新资源信息.
  - 协议版本

```
GET /3.txt  HTTP/1.1
```

- 服务器地址

```
Host: 192.168.0.3:80
```

- 协议头部分 (可选)
- 协议头结束 (空行)

```
\r\n
```

2.2.2 服务器应答浏览器

- HTTP应答方法

  HTTP/1.1 状态码 OK -> HTTP/1.1 200 OK

- 内容格式 (必选项)

  Content-Type: text/plain; charset=iso-8859-1

  - *.html :text/html; charset=iso-8859-1
  - *.jpg : image/jpeg
  - *.png : image/png

- 协议结束空行

  \r\n

- 内容

2.3 HTML

```
<html>
   <head><title> TestPage </title> </head>
   <body>
      <p> Test Ok </p>
      <img src='xx.jpg' /img>
   </body>
</html>
```

2.4 xinetd

2.4.1 守护进程

daemon, 系统中的一个后台进程，周期性的执行某个任务，或者等待某个事件的发生。不会随用户的注销而退出。

创建守护进程 不能是组长进程 (父进程)

1. fork 子进程，父进程结束
2. 子进程 setsid() 创建新会话，脱离终端控制
3. 修订权限、文件描述符重定向 dump2

2.4.2 配置 xinetd

举例 xinetd 接受到客户端请求后，启动 zhangsan 的可执行文件 ./zhangsan

1. 在/etc/xinetd.d 创建一个名为 zhangsan 的文件 (配置文件)
2. 写入配置内容，service zhangsan{}
3. server = /home/itcast/zhangsan (zhangsan 应具有可执行属性，并在该目录下)
4. vi /etc/services

加入zhangsan 2222/tcp #myhttpServer

5. 重启 xinetd 服务器`sudo service xinetd restart stop start restart`

6. `ps aux` grep xinetd|

7. 浏览器地址中输入`localhost:2222`

第三章　开源-libevent-1.4

3.1　参考

代码架构：见参考手册，net 文件夹下

第四章　开源-lighthttpd-1.4.15

4.1　架构图

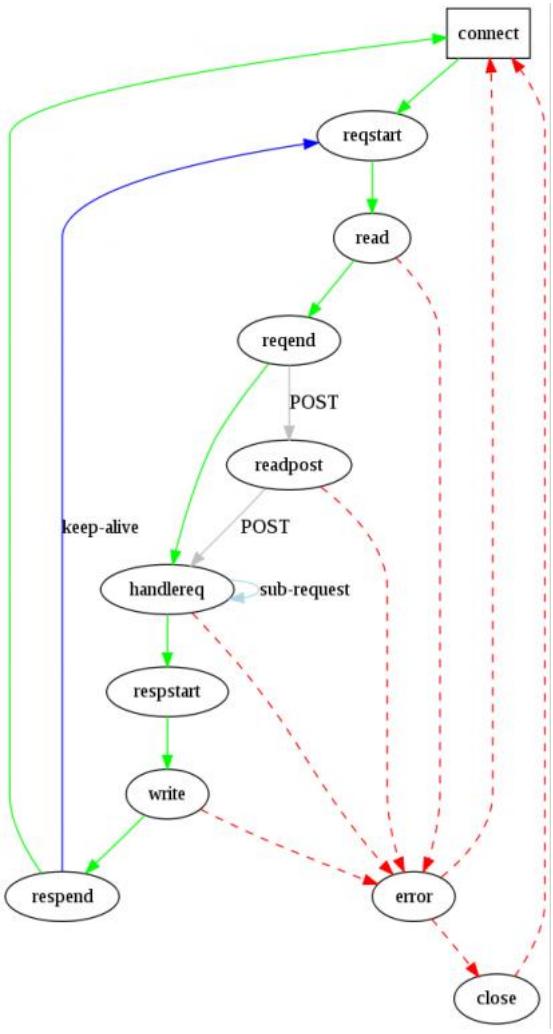

图 4.1: 架构图

第五章　开源-nginx-0.5.38

5.1　架构

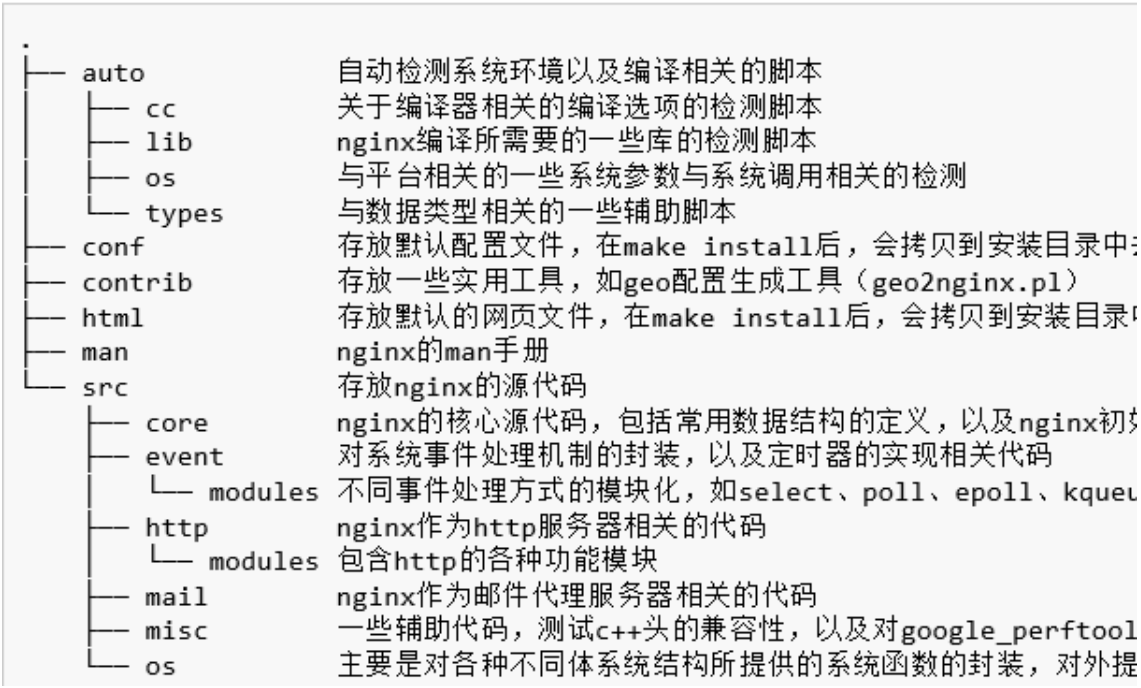

图 5.1: 架构图

5.2　参考

代码架构：http://tengine.taobao.org/book/chapter_09.html

第六章　游戏服务器 - 戴汉水

6.1　经验学习

6.1.1　#ifdef .. 编译选择

6.1.2　string 和 wstring 相互转换

6.1.3　GBK 和 UTF8 的相互变换

6.1.4　boost 智能指针

6.1.5　游戏服务器种类

数据库服务器，游戏服务器，登陆服务器，battle 服务器

6.1.6　enum 枚举的使用

6.1.7　按模块编码，而不是头文件与 cpp 文件

有点像 java 包的概念，像包一样的分解程序，分解成 lib, 然后调用，继承

6.2 错误记录

6.2.1 错误 204 error LNK2001: 无法解析的外部符号

```
"__declspec(dllimport)
class std::basic_ostream<char,struct std::char_traits<char> >
& __cdecl std::endl(class std::basic_ostream<char,struct std::char_traits<char> > &)"
```

解决办法 VS20xx:
项目、属性、链接器、常规、附加库目录：填写附加依赖库所在目录分号间隔多项
项目、属性、链接器、输入、附加依赖项：填写附加依赖库的名字.lib 空格或分号间隔多项

第七章　游戏服务器 2 - 戴汉水

7.1　经验

7.1.1　设置自己的解决方案目录结构

：http://www.360doc.com/content/13/0422/10/8251840_280065154.shtml

7.1.2　学会程序跟踪，与 Debug

7.2　错误记录

7.2.1　Error MSB3073 代码 9009

：

7.2.2　Error fatal error LNK1104: cannot open file 'atlsd.lib'

:http://blog.csdn.net/hsluoyc/article/details/46312293

貌似是 2008 以前版本用的库有个 atlsd.lib，由于 atlsd.lib 在高版本 VS 中也不存在，因此只好用其 Release 版本代替，经测试可以使用。没有安装高版本 VS 的同学可以留言索取 atls.lib。

7.2.3　Project：error PRJ0019: 工具从" 正在执行预生成事件..."

配置属性-生成事件-预编译事件或生成后事件-命令行，删除命令行内容即可

7.2.4　subwcrev 不是内部或外部命令，也不是可运行的程序

：svn 命令：该版本 visual studio 未安装 SVN 版本控制器，导致的错误

7.2.5　resource 文件目录如果打不开

请检查是不是不存在.rc 文件，或.rc 文件多了后缀如.rc.in

第八章　游戏解决方案的搭建

8.1　DLL 动态库

8.2　调用 DLL

8.3　调用 DLL 中的头文件，但其实不是

第九章　Others

9.1　线程池

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。

线程池的好处就在于线程复用，一个任务处理完成后，当前线程可以直接处理下一个任务，而不是销毁后再创建，非常适用于连续产生大量并发任务的场合。

http://wenku.baidu.com/view/f0bd6127ccbff121dd36831a.html

9.1.1　核心思想

假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。

如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

线程池技术正是关注如何缩短或调整 T1,T3 时间的技术，从而提高服务器程序性能的, 它把 T1，T3 分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有 T1，T3 的开销了。

除此之外，线程池能够减少创建的线程个数。通常线程池所允许的并发线程是有上界的，如果同时需要并发的线程数超过上界，那么一部分线程将会等待。而传统方案中，如果同时请求数目为 2000，那么最坏情况下，系统可能需要产生 2000 个线程。尽管这不是一个很大的数目，但是也有部分机器可能达不到这种要求。

因此线程池的出现正是着眼于减少线程池本身带来的开销。线程池采用预创建的技术，在应用程序启动之后，将立即创建一定数量的线程 (N1)，放入空闲队列中。这些线程都是处于阻塞 Suspended）状态，不消耗 CPU，但占用较小的内存空间。当任务到来后，缓冲池选择一个空闲线程，把任务传入此线程中运行。当 N1 个线程都在处理任务后，缓冲池自动创建一定数量的新线程，用于处理更多的任务。在任务执行完毕后线程也不退出，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程都一直处于暂停状态，线程池自动销毁一部分线程，回收系统资源。

基于这种预创建技术，线程池将线程创建和销毁本身所带来的开销分摊到了各个具体的任务上，执行次数越多，每个任务所分担到的线程本身开销则越小，不过我们另外可能需要考虑进去线程之间同步所带来的开销。

9.1.2　组成部分

1. 线程池管理器 ThreadPool　用于创建并管理线程池，包括创建线程池，销毁线程池，添加新任务

2. 工作线程 PoolWorker　线程池中线程，在没有任务时处于等待状态，可以循环的执行任务

3. 任务接口 Task　每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等

4. 任务队列 taskQueue　用于存放没有处理的任务。提供一种缓冲机制。

9.1.3 实现

C++ http://wenku.baidu.com/view/a4cc1093daef5ef7ba0d3c3a.html

Boost http://www.oschina.net/code/snippet_170948_18231