1. 线程安全的对象生命期管理
2. Observer（观察者）设计模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主体对象。这个主体对象在发生状态变化时，会通知所有观察者对象，使他们能够自动更新自己。  
一个软件系统常常要求某一个对象的状态发生变化的时候，某些其他的对象作出相应的改变。做到这一点的设计方案有很多，但是为了使系统能够易于复用，应该选择低耦合度的设计方案。减少对象之间的耦合有利于系统的复用，但是同时设计师需要使这些低耦合度的对象之间能够维持行动的协调一致，保证高度的协作（Collaboration）。观察者模式是满足这一要求的各种设计方案中最重要的一种。



从上面的解释可以看出，观察者模式是非常适合客户-服务器模式编程的。

1. 对象线程安全的定义：

多个线程同时访问时，都表现出正确的行为，无论操作系统以何种顺序调度线程，调用段无需额外的同步或其他协调操作。

1. 对象构造线程安全的要求只有一点：不要在创建对象的时候泄露this指针——不要在构造函数中注册任何回调，不要在构造函数的任何地方把this指针传给跨线程的对象。
2. 析构函数遇到多线程会存在的问题：

在一个对象被多个线程使用时，对象的销毁会出现会存在问题：产生多种竞态条件，有的线程可能在使用对象，有些线程可能却在销毁对象。即出现线程不安全问题。重点应关注析构函数的线程安全。

1. 作为对象成员的mutex并不能完全保证线程安全，如下代码：

Foo::~Foo(){ //释放对象状态

MutextLockGuard lock(mutex\_); //对Foo对象的临界资源加锁过程

//free internal state

}

void Foo::updata(){ //使用对象状态

MutextLockGuard lock(mutex\_);

//make use of internal state

}

有下面代码可能产生竞态条件：

//线程1 //线程2

extern Foo \*x; extern Foo \*x;

delete x; if (x){

x = NULL; x->update();

}

当线程1执行到析构函数并获得锁准备继续执行，线程2执行if(x)判断x不为空后阻塞准备或得x，当线程1执行完，线程2的结果是不可预见的。

另外如果同时读写同一个类的两个对象时也可能发生死锁。解决的办法有，保持相同的加锁顺序，即先加锁地址较小的mutex。

1. 面向对象程序设计中，对象的关系主要有三种：组合（composition）、聚合（aggregation）、关联（association）。（其实还有个依赖（dependency）），其中，聚合和关联都会存在线程不安全问题。
2. 上面的问题很大一部分原因是因为使用了原始指针，因为原始指针不能确保它指向的对象是否还存活着。
3. 使用shared\_ptr（强引用）和weak\_ptr（弱引用）来解决上面提到的问题。weak\_ptr虽然是弱引用，并不能增加引用计数，但是他可以知道一个对象是否还是活着的：如果还活着，那么他可以提升(使用weak\_ptr的lock()函数，该函数是线程安全的)成一个shared\_ptr，如果对象已经死了，提升会失败，返回一个空的shared\_ptr。
4. C++可能的内存问题有以下几大类：

a.缓存区溢出 b.空悬指针 c.重复释放 d.内存泄露 e.不配对的new[]/delete f.内存碎片

其中可以使用智能指针解决上面前五个问题。

1. shared\_ptr本身不是100%线程安全的，所以要在多线程中使用同一个shared\_ptr，正确的做法用mutex保护。
2. shared\_ptr存在的一些问题：
3. 有可能延长对象的生命期，如由于shared\_ptr支持拷贝构造和复制，那么如果不小心遗留了一个拷贝，那么对象可能就会永世长存。
4. shared\_ptr的拷贝要比原始指针要高，有可能在调用函数传值时有性能上的问题（这个问题一般可以通过将函数参数指定为const reference（常引用）来解决）。
5. RAII：Resource Acquisition Is Initialization，资源获取就是初始化

每一个明确的资源配置动作都应该在单一语句中执行，并在该语句中立刻将配置获得的资源交给handle对象（如shared\_ptr），程序中一般不出现delete。

1. 如何获得一个指向当前对象的shared\_ptr<T>对象呢？使用enable\_shared\_from\_this，这是一个以其派生类为模板类型实参的基类模板，集成它，this指针就能变身为shared\_ptr。如:

class A : public enable\_shared\_from\_this < A > {

public:

A(int v1, int v2):a(v1),b(v2){}

shared\_ptr<A> f()

{

return shared\_from\_this();

}

void p(){ cout << a << " " << b; }

private:

int a;

int b;

};

int \_tmain(int argc, char \*argv[])

{

shared\_ptr<A> a(new A(1,2));

shared\_ptr<A> pa = a->f(); //pa也为指向上面new出来的A对象的shared\_ptr

pa->p();

cout<<pa.use\_count(); //输出为2可以证明pa和a都指向能一个内存，

当然a的寿命可能会被延长

return 0;

}

1. 对于上面提到的使用enable\_shared\_from\_this可能会延长对象的生命期，可以考虑使用“弱回调”方式解决，弱回调就是先判断对象是不是还活着，如果活着，就调用它的成员函数，否则忽略掉。其实就是在上面的代码中，将a->f()返回的shared\_ptr强制转换为weak\_ptr，这样就不会延长a的寿命了。修改上面的代码如下：

int \_tmain(int argc, char \*argv[])

{

shared\_ptr<A> a(new A(1,2));

weak\_ptr<A> pa = weak\_ptr<A>(a->f());

pa.lock()->p();

pa.lock().use\_count();

cout<<a.use\_count(); //会看到上面的两个输出都是1，weak\_ptr没有增加引

用计数

return 0;

}

这样就可以做到线程安全的对象回调与析构，还有解决了多个类之间的生命周期影响。

总结

**在使用C++编写多线程程序的时候，应该注意出现的竞态问题（分析可能的竞态条件，不仅是多线程编程的基本功，也是分布式系统的基本功）。因为C++没有垃圾回收机制，所以都要开发人员回收内存，一般就在析构函数回收自己开辟的内存空间，但是将原始指针暴露给多线程往往会造成竞态，如当有些线程准备回收该对象内存，但是另外的线程可能还在使用该对象，而且单纯的检查指针是否为空是不能检查该对象是否还活着，这样就会引起未知的错误。统一用shared\_ptr来管理对象的生命期就变得尤为重要，可以将它理解为C++的自动内存回收。然而在使用时应该主要有可能引起的延长对象的生命期（总结11），解决办法就是总结14，使用shared\_ptr的好搭档weak\_ptr，可用于弱回调。当然，shared\_ptr不是100%线程安全的，所以必须加锁，使用互斥量就可以了。使用以上总结的内容，给出示例代码如下：**

class MutexLock{ //互斥量锁类

public:

MutexLock(){pthread\_mutex\_init(&mutex\_, NULL);}

~MutexLock(){pthread\_mutex\_destroy(&mutex\_);}

void lock(){pthread\_mutex\_lock(&mutex\_);}

void unlock(){pthread\_mutex\_unlock(&mutex\_);}

private:

pthread\_mutex\_t mutex\_;

};

class MutexLockGuard{ //互斥量加锁类

public:

explicit MutexLockGuard(MutexLock& mutex):mutex\_(mutex){mutex\_.lock();}

~MutexLockGuard(){mutex\_.unlock();}

private:

MutexLock& mutex\_;

};

class stock{ //举例股票类

public:

stock():ID(""),price(0.0){}

stock(const string &id, const float p):ID(id),price(p){}

stock(const string &id):ID(id),price(0.0){}

string key(){return ID;}

private:

string ID;

float price;

};

class stockfactory:public enable\_shared\_from\_this<stockfactory>{ //举例，股票对象集合

public:

shared\_ptr<stock> get(const string &key){

shared\_ptr<stock> pstock;

MutexLockGuard lock(mutex\_); //对资源进行加锁

weak\_ptr<stock>& wstock = stocks\_[key];

pstock = wstock.lock();

if (!pstock){

//使用weak\_ptr和shared\_ptr结合，达到弱回调，不会延长生命期

pstock.reset(new stock(key), bind(stockfactory::weakDeleteCallBack,

weak\_ptr<stockfactory>(shared\_from\_this()), \_1));

wstock = pstock;

}

return pstock;

}

private:

mutable MutexLock mutex\_;

map<string, weak\_ptr<stock>> stocks\_;

static void weakDeleteCallBack(const weak\_ptr<stockfactory> &wkfactory,

stock \*s){

//使用weak\_ptr就可以检查目前对象是否还活着

shared\_ptr<stockfactory> factory(wkfactory.lock());

if (factory){

factory->removeStock(s);

}

delete s;

}

void removeStock(stock \*s){

if(s){

MutexLockGuard lock(mutex\_);

stocks\_.erase(s->key());

}

}

};

1. 线程同步精要
2. 线程同步的四个原则：
3. 首要原则是尽量最低限度地共享对象，减少需要的同步场合。一个对象最好不要暴露给其他线程，如果必须，优先使用immutable变量，其次是可修改变量。
4. 使用高级并发编程构件，如TaskQueue、Producer-Consumer Queue、CountDownLatch等等。
5. 最后不得己必须使用底层同步原语时，只用非递归(不可重入/不可中断)的互斥器和条件变量，慎用读写锁，不要用信号量。
6. 除了使用atomic整数之外，不自己编写lock-free代码（无锁编程），也不要用内核级同步原语。
7. 互斥器的使用原则：
8. 建议使用非递归的（不可重入）的互斥器
9. 用RAII手打封装mutex的创建、销毁、加锁、解锁这四个部分（第一章的最后代码已经给出示例代码，MutexLock和MutexLockGuard）。
10. 不手工调用lock()和unlock()，一切交给Guard对象的构造和析构函数，Guard对象的生命期就是整个临界区，这样正好保证在一个函数或者一个作用范围内加锁和解锁。另外，分析对象什么时候析构是C++程序员的基本功。这样的方法叫做Scoped Locking。
11. 在每次构造Guard对象时，思考一路上已经持有的锁，防止因加锁顺序不同而导致死锁。
12. 不使用跨进程的mutex，进程间使用TCP sockets。
13. RAII的基本含义是：C++中的资源，如内存、文件句柄等，应该由对象来管理，资源在对象的构造过程中初始化，并在对象的析构函数中被释放。（STL的shared\_ptr就是RAII的一个具体应用）
14. 使用可RAII后会带来如下的好处：加解锁在一个线程、不会忘记解锁、不会重复解锁。
15. 编写高性能多线程程序只是还要知道cache line（高速缓存行）和false sharing（错误共享）。

cache line:在多处理器（SMP）中每个CPU都会有一个本地高速缓存，一般CPU从内存读取数据到cache中的时候，每次会有一个读取单元，这个读取单元就是一个cache line。

false sharing：我们都知道cache是要保持和内存的数据一致的。一般可能会有多个处理器读取同一个cache line单位的内存数据到自己的cache中。在多线程的情况下会有以下问题出现：位于两个处理器上的两个线程（完全无关），如thread1和thread2，此时thread1和thread2的部分数据位于同一块cache line的上，并被两个处理器读到了各自的cache中，恰恰这时thread1的修改了cache line上的变量，然后thread2上的CPU则会被告知自己的一个cache line（和thread1是相同内存）是数据不一致的。这样thread2上的CPU就要到内存中重新读取该块内存到cache中，但是此时thread2并不关心被修改的那个变量的值（thread1和thread2无关），我们把这个现象叫做false sharing。对于false sharing的避免，主要方法是通过代码检查（intel 官方文档介绍）。

1. 条件变量的使用原则：

条件变量是线程可用的另一种同步机制。条件变量给多个线程提供了一个汇合的场所。条件变量和互斥量一起使用，允许线程以无竞争的方式等待特定的条件发生（UNIX环境高级编程）。

条件变量的正确使用方式：

1. wait端：
2. 必须与mutex一起使用，条件的读写须受此mutex的保护
3. 在mutex上锁的时候才能调用wait()
4. 把条件的判断和wait()放到循环中，不然会造成虚假唤醒（spurious wakeup）
5. signal/broadcast端：
6. 不一定要在mutex已上锁的时候调用signal
7. 在signal之前一般要修改条件
8. 修改条件要在锁的保护下进行
9. 注意区分signal和broadcast
10. 具体的封装演示代码如下：

class condition{ //条件变量的封装

public:

condition(MutexLock &mutex):mutex\_(mutex){pthread\_cond\_init(&cond\_, NULL);}

void wait(){pthread\_cond\_wait(&cond\_, mutex\_.getMutex());}

void notify(){ pthread\_cond\_signal(&cond\_);}

void notifyAll(){pthread\_cond\_broadcast(&cond\_);}

~condition(){pthread\_cond\_destroy(&cond\_);}

private:

MutexLock &mutex\_;

pthread\_cond\_t cond\_;

};

class countDownLatch{

public:

explicit countDownLatch(int c):cond\_(mutex\_),count(c){}

void wait(){

MutexLockGuard lock(mutex\_); //wait端要求1

while(count > 0) //wait端要求2、3

cond\_.wait();

}

void countDown(){

MutexLockGuard lock(mutex\_);// signal/broadcast端要求3

--count; // signal/broadcast端要求2

if (count == 0){cond\_.notifyAll();}

}

private:

mutable MutexLock mutex\_;

condition cond\_;

int count;

};

1. 注意，在验证上面代码正确性的时候，封装了一个创建线程的类，代码如下面，注意在定义线程函数的时候，将其定义成了static函数，这是因为在C语言中线程函数也是全局函数的，如果在C++中不这样定义，编译器会认为类型不匹配，编译不过。

int i=1;

countDownLatch count(5);

class ThreadCreate{

public:

ThreadCreate(){

pthread\_create(&tid, NULL, fun, NULL);

}

private:

pthread\_t tid;

static void\* fun(void\*); //线程函数要定义为static

};

void \*ThreadCreate::fun(void\*){

printf("%lu------>%s\n",(unsigned long)pthread\_self(), "wait condtition count == 0");

count.wait();

sleep(i++);

printf("%lu------>%s\n",(unsigned long)pthread\_self(), "count == 0 exit!");

};

总结

**多线程是一项重要的个人技能，不能因为它难就本能地排斥，掌握同步原语和他们的使用场合是多线程编程的基本功。上面的内容可归纳如下：**

1. **线程同步的四项原则，尽量用高层同步设施（线程池、队列、倒计时）**
2. **使用普通互斥量和条件变量完成剩余的同步任务（**书中还介绍了使用shared\_ptr和mutex代替读写锁(且是read-copy-update模式)的例子，值得学习，此处没有总结出**），采用RAII惯用手法和Scoped Locking。**
3. 多线程服务器的使用场合与常用编程模型
4. 设计分布式系统时需要思考的问题：容错、扩容、负载均衡和任务切换（退休）。
5. 多线程中线程的分类：
6. IO线程，这类线程的主循环是IO multiplexing，阻塞地等在select/poll/epoll\_wait系统的调用上。这类线程也处理定时事件。
7. 计算线程，这类线程的主循环是blocking queue，阻塞地等在condition variable上，这类变量一般位于thread pool中。
8. 第三方库所用的线程，比如logging等。
9. 单线程服务器的编程模型：

non-blocking IO + IO multiplexing(非阻塞IO+IO多路转接)，在这种模型中的结构是一个事件循环（event loop），以事件驱动和事件回调的方式实现业务逻辑。

示例代码：

while(!done)

{

int timeout\_ms = max(1000, getNextTimedCallback());

int retval = ::poll(fds, nfds, timeout\_ms); //这里的poll伸缩性不足，Linux下

可改为epoll

if (retval < 0){

//处理错误，回调用户的error handler

}

else{

//处理到期的timers，回调用户的time handler

if(retval > 0){

//处理IO事件，回调用户的IO event handler

}

}

}

1. 多线程服务器的常用编程模型
2. 每个请求创建一个线程，使用阻塞式IO操作
3. 使用线程池，同样适用阻塞式IO操作。比a提高性能
4. 使用non-blocking IO + IO multiplexing
5. Leader/follower等高级模式
6. 作者推荐的C++多线程服务端编程模型：one event loop per thread + thread pool。

* one event loop per thread：这种模型下，程序里的每个IO线程有一个event loop（事件循环，循环处理某一事件，如处理来自网络中连接到本地TCP 22端口的请求连接事件），代码结构如上面代码。Eventloop代表了线程的主循环，需要让哪个线程干活，就把timer或IO channel注册到哪个线程的loop里即可（即向epoll的某个时间处理线程注册一次）。
* thread pool：用来计算，具体可以是任务队列或生产者消费者队列。

具体的代码实现待以后学习后再补充，目前只提供了理论。

1. 如果要在一台多核机器上提供一种服务或执行一个任务，流行的可用模式有：

* 运行多个单线程的进程
* 主进程+worker进程（master+slaves）

1. 多线程服务器适用的场合：
2. 单线程使用的场合

程序可能会fork；

限制程序的CPU占用率（可能系统上正在运行更重要的进程）。

1. 多线程适用的场合

多线程可以提高响应速度，让IO和计算相互重叠，降低延时。比如分布式集群管理软件。

总结

**本书中作者提倡的多线程编程模型为：one event loop per thread + thread pool。这也是muduo库的模型，这样的编程模型兼顾了系统IO的相应延时和计算的高效率。具体的介绍可查看上面的介绍。学习多线程编程还有一个巨大的好处，即训练异步思维，提高分析并发事件的能力。这对设计分布式系统帮助巨大，因为运行在多台机器上的服务进程本质上是异步的。熟悉多线程编程的话，很容易就能发现分布式系统在消息和事件处理方面的race condition。**

1. C++多线程Linux系统编程精要
2. 多线程系统编程的难点不在于学习线程原语，而在于理解多线程与现有的C/C++库函数和系统调用的交互关系，以进一步学习如何设计并实现线程安全且高效的程序。
3. Linux上的线程标识

虽然Linux的threads库提供了pthread\_self函数用于返回当前线程的标示符，其类型为pthread\_t，但是这是一个未知类型，不能进行一些比较、打印、定义非法值等操作，且在本进程内有效，在/proc下无法查找对应的task，所以pthread\_t不适合用于程序中对线程的标识符。

在Linux上建议使用gettid()系统调用的返回值作为现场id，且它是pid\_t类型的，通常是一个小整数。它可以在/proc中找到对应项、用top等命令查看任务、且是全局唯一的、0是非法的。通常将其存储在一个\_\_thread变量中，防止每次使用的时候都要调用系统调用，提高效率。

第五章 高效的多线程日志