当析构函数遇到多线程

多线程中 C++ 对象的生与死

这是陈硕在 2009 年上海 C++ 技术大会演讲的投影片，可自由用于个人学习，其他使用需得到作者许可。

编写线程安全的类不是难事，用同步原语保护内部状态即可。但是对象的生与死不能由对象自身拥有的互斥器来保护。如何保证即将析构对象 x 的时候，不会有另一个线程正在调用 x 的成员函数？或者说，如何保证在执行 x 的成员函数期间，对象 x 不会在另一个线程被析构？如何避免这种 race condition 是 C++ 多线程编程面临的基本问题，可以借助 tr1 中的 shared_ptr 和 weak_ptr 完美解决。这也是实现线程安全的 Observer 模式的必备技术。

在此基础上，还将介绍一个线程安全的、contention-free 的 Signals/Slots 设计与实现，以及借助 shared_ptr 实现线程安全的 copy-on-write。

When destructors come across multi-threading

Birth and death of objects in threads

陈硕 Shuo Chen
blog.csdn.net/Solstice
giantchen@gmail.com

2009-12

Intended audience

- C++ programmers aware of
  - tr1::shared_ptr, tr1::weak_ptr
  - tr1::function, tr1::bind
  - deadlocks and race conditions
  - Mutex and MutexLock

Agenda

- Part I 45 min
  - Object lifetime management in multi-threads
- Mini QA 5 min
- Part II 30 min
  - Thread safe signals/slots
- QA 10 min

Object lifetime in multi-threads

- In C++, programmers manage lifetime of objects by themselves
- Things become much more complicated when multi-threading comes in to play
- When you are about to delete an object, how do you know that it is not being used in another thread?
- How can you tell if the object is still alive before you trying call its member function?

Mutex and MutexLock

- Mutex wraps creation and destruction of
  - A `CRITICAL_SECTION` on Windows
  - A `pthread_mutex_t` on Linux
- MutexLock wraps acquiring and releasing
  - enter or leave critical section on Windows
  - lock or unlock pthreads mutex on Linux
- Both are not copy-constructible or assignable

Thread safety

- A class is thread-safe
  - if it behaves correctly when accessed from multiple threads
  - regardless of the scheduling or interleaving of the execution of those threads by the OS
  - and with no additional synchronization or other coordination on the part of the calling code

Java Concurrency in Practice, by Brian Goetz et al.

A thread safe Counter class

- Write a thread safe class is not difficult
  - Protect internal state via synchronization
- How about its birth and death?

```
class Counter : noncopyable
{
 public:
  Counter(): value_(0) {}
  int64_t value() const;
  int64_t increase();
  int64_t decrease();
 private:
  int64_t value_;
  mutable Mutex mutex_;
}
```

```
int64_t Counter::value() const
{
  MutexLock lock(mutex_);
  return value_;
}
int64_t Counter::increase()
{
  MutexLock lock(mutex_);
  int64_t ret = value_++;
  return ret;
}
```


* In real world, atmoic operations are better for Counter.

Bear easily

- Do not allow `this` pointer to escape during construction
  - Don't register to any callback in ctor (*)
  - Even at last line of ctor is also bad

```
// Don't do this.
class Foo : public Observer
{
public:
  Foo(Observable* s) {
    s->register(this); // X
  }
  virtual void update();
};
```

```
// Do this.
class Foo : public Observer
{ // ...
  void observe(Observable* s) {
    s->register(this);
  }
};
Foo* pFoo = new Foo;
Observable* s = getIt();
pFoo->observe(s);
```

(*) unless you know it will not call you back in any other thread

Die hard

- You can't tell if a object is alive by looking at the pointer or reference
  - if it's dead, an invalid object won't tell anything
  - set the pointer to NULL after delete? helpless
- There must be some alive object help us telling other object's state
- But, pointer and reference are not objects, they are primitive types

Mutex isn't the solution

- It only guarantees functions run sequentially

```
Foo::~Foo()
{
  MutexLock lock(mutex_);
  // free internal state
}
// thread A
delete x; x = NULL;
```

```
void Foo::update()
{
  MutexLock lock(mutex_);
  // make use of internal state
}
// thread B
if (x) x->update();
```

- What if x->~Foo() runs before x->update() ?
- There must be something outside the object to rescue.

*Worse, mutex_ itself is a member of x, which has already been destroyed.

Mutex con't

- What's wrong of those function/operator ?

```
void swap(Counter& a, Counter& b)
{
  MutexLock aLock(a.mutex_);
  MutexLock bLock(b.mutex_);
  int64_t value = a.value_;
  a.value_ = b.value_;
  b.value_ = value;
}

// thread 1     // thread 2
swap(a, b);     swap(b, a);
```

```
Counter& Counter::operator=(
                  const Counter& rhs)
{
  if (this == &rhs)
    return;

  MutexLock myLock(mutex_);
  MutexLock itsLock(rhs.mutex_);
  value_ = rhs.value_;
  return *this;
}
```

Member mutex

- Can help reading and writing of sibling members
- Can not help destructing
- Can not help reading then writing two objects, because of potential deadlocks
  - assignment operator
  - swap

How could this happen?

- Association relation non-permanent
  - a member pointer/reference to an object whose lifetime is not controlled by me
  - any kind of object callback
- Observer pattern
  - when the observable notifies observers, how does it know the observer is still alive?
- Observer unregisters itself in dtor?
  - not working, race condition exists

Race condition

- Observer got deleted in one thread
- Calling its update() in another thread

```
struct Observable
{
  void register(Observer* x);
  void unregister(Observer* x);
  void notifyObservers() {
    for each Observer* x:
      x->update(); // (1)
  }
  // ...
}

reach (2), switch to (1)
Worse: Observer is a base class.
```

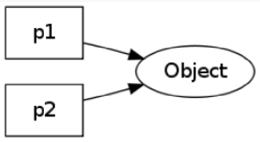
```
struct Observer
{
  virtual void update() = 0;
  void observe(Observable* s) {
    s->register(this);
    subject_ = s;
  }
  virtual ~Observer() {
    // (2)
    subject_->unregister(this);
  }
  Observable* subject_;
};
```

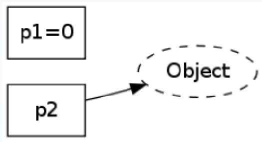

Heuristics

- Raw pointers are bad, esp. when visible in other threads.
- Observable should store some thing other than raw Observer* pointer to tell if the observer still lives.
- So does Observer
- A smart pointer could do this
  - Not that simple, trick things happens
  - No worry, we will study them carefully

The problem

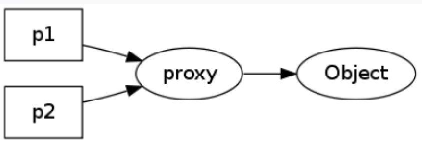
- two raw pointers (possible in threads)
- to the same objects on heap
- object is deleted via p1
- although p1 is set to 0
- p2 is a dangling pointer

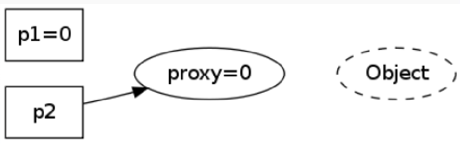




A "solution"

- two pointers
- to a proxy on heap
- the proxy points to the object
- delete the object by p1
- set proxy to 0
- now p2 knows it's gone
- when to delete proxy?



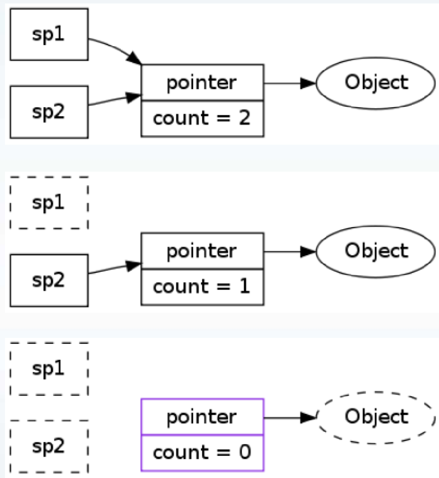



```
Type* volatile proxy;
```

```
// to safely delete and set proxy to 0
Type* tmp = proxy;
proxy = NULL;
memory_barrier(); // flush the CPU cache
delete tmp;
```

A better solutoin

- sp1, sp2 are objects, not raw pointers
- proxy lives on heap, with a count member
- count=2 initially
- count=1 when sp1 dies
- count=0 when sp2 dies as well
- it is time to delete the object and the proxy
- reference counting, huh?



Universal solution

- Another layer of indirection
- Calling through a proxy object, which always exists
  – either a value stored locally, or a global facade
- Handle/Body Idiom once more
- Wait! the standard library has exact what we need

shared_ptr/weak_ptr basics

- shared_ptr<T> is a reference counting smart pointer, available in boost and tr1
- It destroy the underlying object when the count goes down to zero
- weak_ptr is a rcsp without increasing the reference count, it can be promoted to a shared_ptr
- No explicit `delete` any more

shared_ptr/weak_ptr con't

- shared_ptr controls the lifetime
  - The object is guaranteed to be deleted when the last shared_ptr pointing to it is destroyed or reset
- weak_ptr doesn't control
  - but it know whether the object is dead ♥
- Counting is atomic, thread safe and lock-free
  - `_Interlocked(In|De)crement` on Windows
  - "`lock xadd`" instruction on Linux/x86
- Use them as values
  - can be stored in std containers

Common pointer pitfalls

- Buffer overrun
  - vector or other standard containers
- Dangling pointer
  - shared_ptr/weak_ptr
- Double delete
  - scoped_ptr
- Memory leak
  - scoped_ptr
- Unpaired new[]/delete
  - vector, shared_array, scoped_array

Apply to observer

- Stores weak_ptr to Observer

```
struct Observable { // not thread safe!
  void register(weak_ptr<Observer> x);
  void unregister(weak_ptr<Observer> x);
  void notifyObservers() {
    Iterator it = observers_.begin();
    while (it != observers_.end()) {
      shared_ptr<Observer> obj(it->lock());
      if (obj) {
        obj->update(); ++it;
      } else {
        it = observers_.erase(it);
      }
    }
  }
  vector<weak_ptr<Observer> > observers_;
};
```

Outstanding issues

- Not flexible, should allow non-shared_ptr obj
- In dtor of Observer, it unregister itself from the Observable
  - What if Observable object goes out of scope?
- To make Observable thread safe, mutex is used in `(un)register` and `notifyObservers`
  - Will register/unregister be blocked for indeterminate period by notifyObjservers?
- We will address them in Part II

Pitfalls of shared_ptr

- Do not extend lifetime inadvertently
  - Why not store shared_ptr in Observable?
- Thread safety
  - same as built-ins, string and containers
- Used as function parameter
  - pass by const reference is Ok and efficient

```
void doit(const shared_ptr<Foo>& pFoo)
```

  - as long as most outer caller holds an instance

An object cache

- Create Foo object for a new key
- Return the same Foo object for the same key
- But Foo never gets a chance to be deleted
  - even no one else holds it, except the cache

```
class FooCache : boost::noncopyable
{
 public:
  shared_ptr<Foo> get(const string& key);
 private:
  map<string, shared_ptr<Foo> > data_;
  mutable Mutex mutex_;
};
```


weak_ptr to rescue (?)

- Memory leak! `data_` keeps growing

```
class FooCache : boost::noncopyable
{
 public:
  shared_ptr<Foo> get(const string& key) {
    shared_ptr<Foo> pFoo;
    MutexLock lock(mutex_);
    weak_ptr<Foo>& wkFoo = data_[key];
    pFoo = wkFoo.lock();
    if (!pFoo) { pFoo.reset(new Foo(key)); wkFoo = pFoo; }
    return pFoo;
  }
 private:
  map<string, weak_ptr<Foo> > data_;
  mutable Mutex mutex_;
};
```

Custom deallocator

- d(p) will be called for delecting the object

```
shared_ptr(Y* p, D d); void reset(Y* p, D d);
```

```
class FooCache : boost::noncopyable
{
  // in get(), change pFoo.reset(new Foo(key)); to
  // pFoo.reset(new Foo(key),
  //            boost::bind(&FooCache::deleteFoo, this, _1));

  void deleteFoo(Foo* object) {
    if(object) {
      MutexLock lock(mutex_);
      data_.erase(object->key());
    }
    delete object;  // sorry, I lied
  }
  // assuming FooCache lives longer than all Foo's ...
```


Is shared_ptr 100% thread safe?

- Not possible, it has two pointer members
  - A shared_ptr instance can be "read" (accessed using only const operations) simultaneously by multiple threads
  - Different shared_ptr instances can be "written to" (accessed using mutable operations such as `operator=` or `reset`) simultaneously by multiple threads, dtor is considered a "write access"
    - even when these instances are copies, and share the same reference count underneath
  - If a shared_ptr is to be read and written from multiple threads, protect both with mutex

* http://www.boost.org/doc/libs/1_41_0/libs/smart_ptr/shared_ptr.htm

Safely read / write shared_ptr

```
shared_ptr<Foo> globalPtr;
Mutex mutex; // No need for ReaderWriterLock
void doit(const shared_ptr<Foo>& pFoo);
```

```
void read()
{
  shared_ptr<Foo> ptr;
  {
    MutexLock lock(mutex);
    ptr = globalPtr;
  }

  // use ptr since here
  doit(ptr);
}
```

```
void write()
{
  shared_ptr<Foo> newptr(new Foo);
  {
    MutexLock lock(mutex);
    globalPtr = newptr;
  }

  // use newptr since here
  doit(newptr);
}
```


shared_ptr gotchas

- The destruction is captured at creation
  - no virtual destructor is needed for T
  - a shared_ptr<void> holds everything
  - safely pass boundary of modules on Windows
  - binary compatible even if object size changes
    - Given that all accessors must be out-lined
- Deletion happens in the same thread as the last shared_ptr goes out of scope, not necessary same thread of creation.
- An off-the-shelf RAII handle
  - be cautious of cycle referencing, lead to leaks
  - usually "*A* holds a shared_ptr to *B*, *B* holds a weak_ptr back to *A*, *B* is shared by *A*s or *A*/*C*"

enable_shared_from_this

- Get a shared_ptr from `this`
- Only works if this object is held by `shared_ptr`
- Will not work in ctors

```
class FooCache : public boost::enable_shared_from_this<FooCache>,
                 boost::noncopyable
{ /* ... */ };

shared_ptr<FooCache> pCache(new FooCache);

shared_ptr<Foo> FooCache::get(const string& key) {
  // ...
  pFoo.reset(new Foo(key),
             boost::bind(&FooCache::deleteFoo,
                         shared_from_this(),
                         _1));
```


Alternatives

- A centralized object facade, all callbacks go through it.
  - calling two distinct objects from two threads shouldn't involve any lock. it's hard to implement within this model
- Roll your own proxy class
  - which simply does the same thing as shared_ptr

Other languages?

- Java, C#
- Concurreny is hard without garbage collection
- GC languages will not face this problem, although they have their own.

Take aways

- Manage lifecycle of objects with shared_ptr, especially in multi-thread programs.
- Otherwise you are reinventing wheel, similar but worse
- Use weak_ptr for notifications, caches, etc.
- Define your deallocator, for object pools

Thread safe signals/slots

Part II

What's wrong with Observer?

- Its Object-Oriented design
  - Observer is a base class
    - strong restriction/couple on type
  - To observe two events => multiple inheritance
    - call my create() when an order comes in
    - call my destroy() when the cafe closes
  - To observe same event twice => helper classes
    - call my start() member function 1 second later
    - call my stop() member function 10 seconds later

Signals/Slots

- A one-to-many callback, similar to Observer
  - a signal is an event
  - a slot is a callback executed when the signal is raised
  - one signal, many slots
- No restriction on types of slots
  - Any class can subscribe to any signal
- Can be done with standard libraries
  - Unlike QT, no preprocessing

A Timer, call me back later

```
class Timer // an easy way to do things later
{
 public:
  typedef boost::function<void ()> Callback;

  void runAfter(int time_in_ms, const Callback& cb);
  static Timer& instance();
  // ...
};

class Request
{
  void send() {
    Timer::instance().runAfter( // cancel me 10s later
        10*1000, boost::bind(&Request::cancel, this));
  }
  void cancel();
};
```

tr1::function and tr1::bind basics

- tr1::function is a genetic functor which references to any *callable* things.
- `bind` makes a `function` pointing to member functions of any class

```
boost::function<void()> f;
Foo* pFoo = new Foo;
f = bind(&Foo::doit, pFoo);
f(); // calls pFoo->doit()

Bar bar; // possible bar is noncopyable
f = bind(&Bar::dothat, boost::ref(bar));
f(); // calls bar.dothat()
```


A new methodology of interface/library design

- function+bind = delegate in C#, Observer in Java
- Get rid of inheritance, base class, hierarchy
- Partial restriction on function signature, not on type or function name
- A replacement of Strategy/Command pattern
  - And many OO behavioural design patterns
- As fast as calling through member function pointer

A Thread class

```
class Thread : boost::noncopyable
{
 public:
  typedef boost::function<void()> Callback;
  Thread(Callback cb) : cb_(cb)
  { }
  void start() {
    /* some magic to call run() in new created thread */
  }
 private:
  void run() {
    cb_();
  }
  Callback cb_;
  // ...
};
```

```
struct Foo
{
  void run1();
  void run2();
};

Foo foo;
Thread thread1(boost::bind(&Foo::run1, &foo));
thread1.start();
Thread thread2(boost::bind(&Foo::run2, &foo));
thread2.start();
```


Caveats

- bind makes copy(-ies) of argument(s)
  - use boost::ref() when the object is noncopyable
  - shared_ptr

```
tr1::function<void()> f;
shared_ptr<Foo> pFoo(new Foo);
f = bind(&Foo::doit, pFoo); // long life foo
```

- bind also takes parameter placeholders

```
void Foo::xyz(int);
function<void(int)> f = bind(&Foo::xyz, pFoo, _1);
f(42); // pFoo->xyz(42);

vector<Foo> foos;
for_each(foos.begin(), foos.end(), bind(&Foo::xyz, _1, 42));
// call Foo::xyz(42) on every objects in foos
```

Is Timer thread safe?

- No, same issue as Observer
- Unregisier in dtor does not work
- To make it safe, either
  - the timer guarantee the object not deleted
  - check its aliveness before calling cancel()

Weak callback

```
class Request : public enable_shared_from_this<Request>
{
  static void timeoutCallback(boost::weak_ptr<Request> wkReq) {
    shared_ptr<Request> req(wkReq.lock());
    if (req) {
      req->cancel();
    }
  }

  void send() {
    Timer::instance().runAfter( // cancel me 10s later
        10*1000,
        boost::bind(&Request::timeoutCallback,
                    boost::weak_ptr<Request>(shared_from_this())));
  }

  // ...
```

Boost signals?

- Cons
  - Not thread safe
  - Not header only
  - auto unregister by inheriting `trackable` class
    - makes connecting multiple signals difficult
- Update 2010 Jan:
  - Boost.signals2 is thread safe and header only
  - The rest of slides can be ignored

Our Signal

- Synchronous, callback in the same thread
- Auto disconnecting at destructing Slot
  - No base class requirement
- Thread safe and race condition free
- Contention free
  - None of the methods will be blocked for uncertain time

Thanks!

Q&A

blog.csdn.net/Solstice
giantchen@gmail.com