# 设计模式

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 修订内容 | 修改人 |
| 2019/01/24 | V0.1 | 初稿 | 朱灿基 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

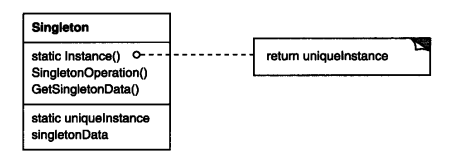
# 一、概述

# 二、创建型模式

## 2.1、单例模式

单例模式（Singleton）：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

单例模式结构：



Singleton定义一个Instance操作，允许客户访问它的唯一实例。（Instance操作：C++中的一个静态成员函数）

客户只能通过Singleton的Instance操作访问一个Singleton的实例。

Singleton的优点：

1、对唯一实体的受控访问。因为Singleton类封装它的唯一实例，所以它可以严格的控制客户怎样及何时访问它。

2、缩小名空间。Singleton模式是对全局变量的一种改进。它避免了那些存储唯一实例的全局变量污染名空间。

3、允许对操作和表示的精化。Singleton类可以有子类，而且用这个扩展类的实例来配置一个应用很容易的。你可以用你所需要的类的实例在运行时刻配置应用。

4、允许可变数目的实例。这个模式使得你易于改变你的想法，并允许Singleton类的多个实例。此外，你可以用相同的方法来控制应用所使用的实例的数目。只有允许访问Singleton实例的操作需要改变。

|  |
| --- |
| static Singleton\* Instance();  static Singleton\* Instance(int a);  static Singleton\* Instance(int a, int b); |

可以定义多个Instance操作，根据函数参数的变化来进行定义。

5、比类操作更灵活。**另一种**封装单例功能的方式是使用类操作（即C++中的静态成员函数）。但这种语言技术都难以改变设计以允许一个类有多个实例。此外，C++中的静态成员函数不是虚函数，因此子类不能多态的重定义它们。（注：此处 **另一种** 对比的是单例模式）

Singleton模式的实现问题

1、保证一个唯一的实例。Singleton模式使得这个唯一实例是类的一般实例，但该类被改写成只有一个实例能被创建。做到这一点的常用方法是将创建这个实例的操作隐藏在一个类操作（一个静态成员函数或一个类方法）后面，由它保证只有一个实例被创建。这个操作可以访问保存唯一实例的变量，而且它可以保证这个变量在返回值之前用这个唯一实例化初始化。这种方法保证了单例在它的首次使用前被创建和使用。

C++中，可以用Singleton类的静态成员函数Instance来定义这个类操作。Singleton还定义了一个静态成员变量\_instance，它包含了一个指向它的唯一实例的指针。

Singleton类定义：

|  |
| --- |
| class Singleton {  public:  static Singleton\* Instance();  protected:  Singleton();  private:  static Singleton\* \_instance;  }; |

Singleton实现：

|  |
| --- |
| Singleton\* Singleton::\_instance = nullptr;  Singleton\* Singleton::Instance() {  if (\_instance == nullptr) {  \_instance = new Singleton;  }  return \_instance;  } |

客户访问：

|  |
| --- |
| int main()  {  static Singleton\* ssina = Singleton::Instance();  } |

客户仅通过Instance成员函数访问这个单例。变量\_instance初始化为指针空值，而静态成员函数Instance返回该变量值。Instance使用惰性初始化，它的返回值直到第一次访问才创建和保存。

2、创建Singleton类的子类。主要问题与其说是定义子类不如说是建立它的唯一实例，这样客户就可以使用它。事实上，指向单例实例的变量必须用子类的实例化进行初始化。最简单的技术是在Singleton的Instance操作中决定你想使用的是哪一个单例。

另一个选择Singleton的子类的方法是将Instance的实现从父类中分离出来并将它放入子类。这就允许C++程序员在链接时刻决定单例的类（即通过链入一个包含不同实现的对象文件），但对单例的客户则隐藏这一点。

链接的方法在连接时刻确定了单例类的选择，这使得难以在运行时刻选择单例类。使用条件语句来决定子类更加灵活一些，但这硬性限定了可能的Singleton类的集合。

一个更灵活的方法是使用一个单例注册表（registry of singleton）。可能的Singleton类的集合不是由Instance定义的，Singleton类可以根据名字在一个众所周知的注册表中注册它们的单例实例。

这个注册表在字符串名字和单例之间建立映射。当Instance需要一个单例时，它参考注册表，根据名字请求单例。

注册表查询相应的单例（如果存在的话）并返回它。这个方法使得Instance不再需要直到所有可能的Singleton类或实例。它所需要的只是所有Singleton类的一个公共的接口，该接口包括了对注册表的操作

## 2.2、享元模式

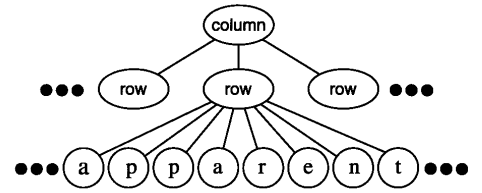
享元（flyweight）：运用共享技术有效支持大量细粒度的对象。

Flyweight模式描述了如何共享对象，使得可以细粒度地使用它们而无需高昂的代价。

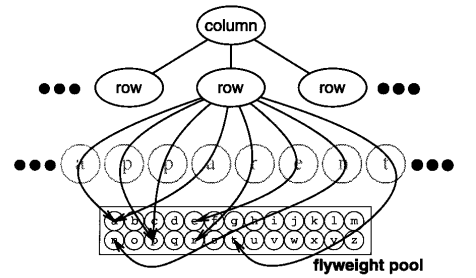
Flyweight是一个共享对象，它可以同时在多个场景（context）中使用，并且在每个场景中Flyweight都可以作为一个独立的对象。Flyweight不能对它所运行的场景做出任何假设，这里的关键概念是内部状态和外部状态之间的区别。内部状态存储于Flyweight中，它包含了独立于Flyweight场景的信息，这些信息使得Flyweight可以被共享。而外部状态取决于Flyweight场景，并根据场景而变化，因此不可共享。用户对象负责在必要的时候将外部状态传递给Flyweight。

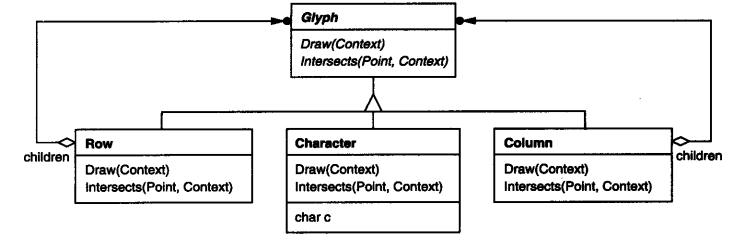
Flyweight模式对那些通常因为数量太大而难以用对象来表示的概念或实体进行建模。例如，文档编辑器可以为字母表中的每一个字母创建一个Flyweight。每个Flyweight存储一个字符代码，但它在文档中的位置和排版风格可以在字符出现时由正文排版算法和使用的格式化命令决定。字符代码是内部状态，而其他的信息则是内部状态。

逻辑上，文档中的给定字符每次出现都有一个对象与其对应。



然而，物理上每个字符共享一个Flyweight对象，而这个对象出现在文档结构中的不同地方。一个特定字符对象的每次出现都指向同一个实例，这个实例位于Flyweight对象的共享池中。





适用性

Flyweight模式的有效性很大程度上取决于如何使用它以及在何处使用它。当以下情况都成立时使用Flyweight模式：

1、一个应用程序使用了大量的对象。

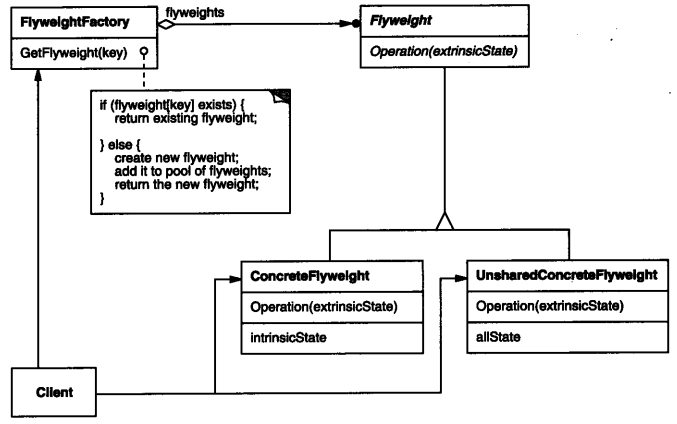
2、完全由于使用大量的对象，造成很大的存储开销。

3、对象的大多数状态都可变为外部状态。

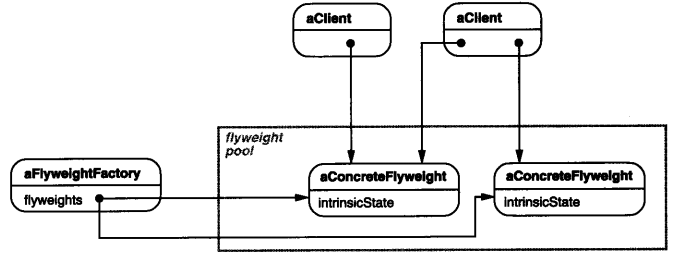
4、如果删除对象的外部状态，那么可以用相对比较少的共享对象取代很多组对象。

5、应用程序不依赖于对象标识。由于Flyweight对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

结构：



下面的对象图说明了如何共享Flyweight。



参与者：

**Flyweight(Glyph)** 描述一个接口，通过这个接口Flyweight可以接受并作用于外部状态。

**ConcreateFlyweight(Character)** 实现Flyweight接口，并为内部状态（如果有的话）增加存储空间。ConcreateFlyweight对象必须是可共享地。它所存储的状态必须是内部的；即，它必须独立于ConcreateFlyweight对象的场景。

**UnsharedConcreateFlyweight(Row, Column)** 并非所有的Flyweight子类都需要被共享。Flyweight接口使共享成为可能，但它并不强制共享。在Flyweight对象结构的某些层次，UnsharedConcreateFlyweight对象通常将ConcreateFlyweight对象作为子节点（Row和Column就是这样）。

**FlyweightFactory** 创建并管理Flyweight对象。确保合理地共享Flyweight。当用户请求一个Flyweight时，FlyweightFactory对象提供一个已创建的实例或者创建一个（如果不存在的话）。

**Client** 维持一个对Flyweight的引用。计算或存储一个（多个）Flyweight的外部状态。

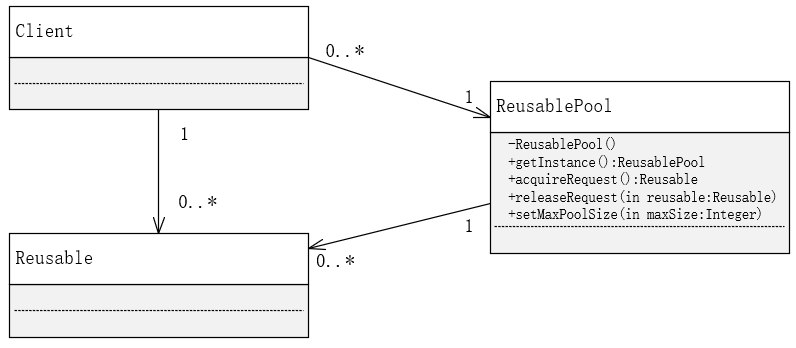
## 2.3、对象池模式

Object Pool[1、2]

对象池属于创建型模式，可以说是一个特殊的工厂模式：它预先创建好若干个可用的对象，用户向池申请对象，使用完毕后归还池而不是直接销毁，所以归还的对象可以被之后的请求复用。通过这种方式，对象池模式平摊了昂贵的构造成本，消除了对象的销毁成本，可以提高系统的整体运行效率。

**对象池模式（The** **Object Pool Pattern）**

对象池是一种对象复用技术。Mark Grand在《Patterns in Java》描述了对象池的设计模式，它通过管理和复用有限对象来共享某些稀少或必须付出昂贵代价的资源，该模式的UML结构分为3部分。



Reusable：被复用的对象或资源。

Client：复用对象的使用者，1个Client对象可以使用多个复用对象，一个复用对象只能同时被1个Client使用。

ReusablePool：对象池即复用对象的容器或集合，用来管理和存储可复用的对象。一般来说为保证对象复用的安全，对象池将按照Singleton的模式设计为全局唯一实例。

该模式被广泛使用，出现了连接池、线程池等许多应用。

理解对象池模式

对象池模式管理一个可替代对象的集合。组件从池中借走对象，用它来完成一些任务并当任务完成时归还该对象。被归还的对象接着满足请求，不管是同一个组件还是其他组件的请求。对象池模式可以管理那些代表现实资源或者通过重用来分摊昂贵初始化代价的对象。

**使用对象池**

对象池技术可以避免在程序的生命周期中创建和删除大量对象。如果知道程序需要同一类型的大量对象，而且对象的生命期都很短，就可以为这些对象创建一个池（pool）进行缓存。只要代码中需要一个对象，就可以向对象池请求。用完此对象时，要把它放回到池中。对象池只创建一次对象，因此他们的构造函数只调用一次，而不是每次使用都调用。因此，当构造函数要完成一些设置动作，而且这些设置可以应用于该对象的多次使用时，对象池就很合适。另外在非构造函数方法调用中要在对象上设置特定于实例的参数时，也很适用于对象池。

**一个对象池实现**

对象池类模板的实现。这个池在构造时会分配一大块（chunk）指定类的对象（块可理解为包括许多对象，即一堆对象），并通过acquireObject()方法交出对象。当客户用完这个对象时，会通过releaseObject()方法将其返回。如果调用了acquireObject，但是没有空闲的对象，池会分配另一块对象。

对象池实现中最难的一方面是要记录那些对象是空闲的，那些对象正在使用，这个实现采用了以下做法，即把空闲的对象保存在一个队列中。每次客户请求一个对象时，池就会把队列中的第一个对象交给该客户。这个池不会显式地跟踪正在使用的对象。它相信客户在用完对象后会正确地将对象交还到池中。另外，这个池会在一个像两种记录所有已分配的对象。这个向量尽在撤销池时才会用到，以便释放所有对象的内存，从而避免内存泄漏。

代码示例：

|  |
| --- |
| #include <queue>  #include <vector>  #include <memory>  #include <stdexcept>  #include <algorithm>  using namespace std;  //#define kDefaultChunkSize 10  template <typename T>  class ObjectPool  {  public:  ObjectPool(int chunkSize = kDefaultChunkSize)  throw(invalid\_argument, bad\_alloc);    ~ObjectPool();    T& acquireObject();    void releaseObject(T& obj);    protected:  //m\_FreeList stores the objects that are not currently in use by clients  queue<T\*> m\_FreeList;  //m\_AllObjects stores pointers to all the objects, in use or not.  //This vector is needed in order to ensure that all objects are freed properly in the destructor  vector<T\*> m\_AllObjects;  int m\_ChunkSize;  static const int kDefaultChunkSize = 10;    //Allocates m\_ChunkSize new objects and adds them to the m\_FreeList  void allocateChunk();  static void arrayDeleteObject(T\* obj);    private:  ObjectPool(const ObjectPool<T>& src);  ObjectPool<T>& operator=(const ObjectPool<T>& rhs);    };  template <typename T>  const int ObjectPool<T>::kDefaultChunkSize;  template <typename T>  ObjectPool<T>::ObjectPool(int chunkSize)  throw(invalid\_argument, bad\_alloc) : m\_ChunkSize(chunkSize)  {  if (m\_ChunkSize <= 0)  {  throw invalid\_argument("chunk size must be positive");  }  //create m\_ChunkSize objects to start.  allocateChunk();  }  //在连续的存储空间中分配m\_ChunkSize个元素，  //它会在vector中存储对象数组的指针，并把每个对象压至queue中。  template <typename T>  void ObjectPool<T>::allocateChunk()  {  T\* newObjects = new T[m\_ChunkSize];  m\_AllObjects.push\_back(newObjects);  for (int i = 0; i < m\_ChunkSize; i++)  {  m\_FreeList.push(&newObjects[i]);  }  }  //Freeing function for use in the for\_each algorithm in the destructor  template <typename T>  void ObjectPool<T>::arrayDeleteObject(T\* obj)  {  delete [] obj;  }  template <typename T>  ObjectPool<T>::~ObjectPool()  {  for\_each(m\_AllObjects.begin(), m\_AllObjects.end(), arrayDeleteObject);  }  //返回空闲列表中的队头对象，如果没有空闲对象则首先调用allocateChunk  template <typename T>  T& ObjectPool<T>::acquireObject()  {  if (m\_FreeList.empty())  {  allocateChunk();  }  T\* obj = m\_FreeList.front();  m\_FreeList.pop();  return (\*obj);  }  //将对象返回到空闲列表的队尾  template <typename T>  void ObjectPool<T>::releaseObject(T& obj)  {  m\_FreeList.push(&obj);  } |

**使用对象池**

考虑一个要从用户得到请求并处理这些请求的应用。这个应用很可能是图形前端和后端数据库之间的一个中间件。可以把每个用户请求编码到一个对象中。

代码示例：

|  |
| --- |
| //使用对象池，用户请求  class UserRequest  {  public:  UserRequest() {}  ~UserRequest() {}    //Methods to populate the request with specific information  //Methods to retrive the request data    private:  //data members    };  //客户  class Client  {  public:  Client() {}  ~Client() {}    UserRequest& obtainUserRequest(ObjectPool<UserRequest>& pool);  void processUserRequest(ObjectPool<UserRequest>& pool, UserRequest& req);  private:  };  UserRequest& Client::obtainUserRequest(ObjectPool<UserRequest>& pool)  {  cout << "Client::obtainUserRequest: " << pool.getCount() << endl;  UserRequest& request = pool.acquireObject();  sleep(10);  //Populate the request with user input  return request;  }  void Client::processUserRequest(ObjectPool<UserRequest>& pool, UserRequest& req)  {  //Process the request  cout << "Client::processUserRequest: " << pool.getCount() << endl;  pool.releaseObject(req);  }  int main()  {  ObjectPool<UserRequest> requestPool(1000);    while (/\*program is running\*/)  {  UserRequest& req = obtainUserRequest(requestPool);  processUserRequest(requestPool, req);  }  } |

在上述例子中，对比标准类图：

ReusablePool --- ObjectPool

Reusable --- UserRequest

Client --- Client

## 2.4、前摄器模式

Proactor

前摄器模式是应用于异步调用的设计模式，它的核心是前摄器、异步的操作处理器、异步的事件多路分离器和完成事件队列，可以不适用线程实现异步操作。

（与前摄器模式对应的另外一种异步模式是反应器模式Reactor，它处理的不是完成事件（complete），而是就绪事件（ready））

前摄器模式的基本流程可以简述如下：

前摄器发起创建一个完成处理器，用于在异步调用完成后的回调，然后发起一个异步操作，交给操作处理器异步执行，当异步操作完成时操作处理器将把事件放入完成事件队列。前摄器调用多路分离器从完成事件队列中获得并分派事件，回调完成处理器执行后续操作。

前摄器模式用于异步调用有很多的好处，它封装了并发机制，简化了功能代码的编写，将并发机制与线程的执行解耦，不需要考虑多线程的同步问题，能够提供高性能的异步操作。但它也有缺点，模式比较复杂，处理流程难以理解和调试。

boost的asio库基于操作系统的异步调用机制实现了可移植的前摄器模式，解耦了应用程序与操作系统，可以高效的实现异步I/O操作。

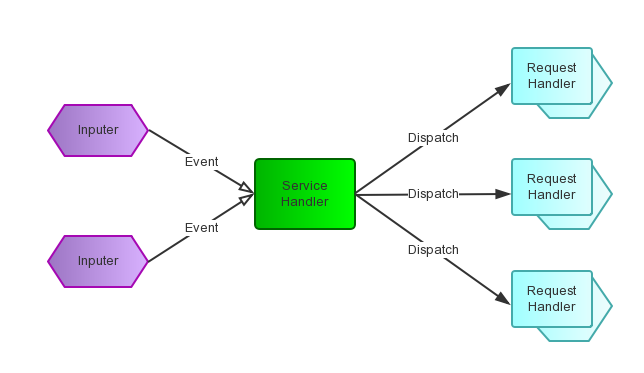
## 2.5、反应器模式

### 1、概念

Reactor[3、4]

The reactor design pattern is an event handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.

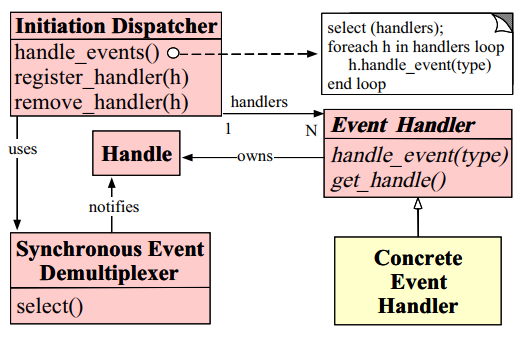
Reactor模式首先是事件驱动的，有一个或多个并发输入源，有一个Service Handler，有多个Request Handlers；这个Service Handler会同步的将输入的请求（Event）多路复用的分发给相应的Request Handler。



从结构上，这有点类似生产者消费者模式，即有一个或多个生产者将事件放入一个Queue中，而一个或多个消费者主动的从这个Queue中Poll事件来处理；而Reactor模式则并没有Queue来做缓冲，每当一个Event输入到Service Handler之后，该Service Handler会主动的根据不同的Event类型将其分发给对应的Request Handler来处理。

### 2、Reactor模式的类图结构

下面的这个类图是Reactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events 这篇论文中的图，但是类之间的关系不是特别准确。



类图对比

（缺一张类图，后续补充）

**Handle**：即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer等。由于Reactor模式一般使用在网络编程中，因而这里一般指Socket Handle，即一个网络连接（Connection，在Java NIO中的Channel）。这个Channel注册到Synchronous Event Demultiplexer中，以监听Handle中发生的事件，对ServerSocketChannnel可以是CONNECT事件，对SocketChannel可以是READ、WRITE、CLOSE事件等。

**Synchronous Event Demultiplexer（同步事件分离）**：阻塞等待一系列的Handle中的事件到来，如果阻塞等待返回，即表示在返回的Handle中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的select来实现。在Java NIO中用Selector来封装，当Selector.select()返回时，可以调用Selector的selectedKeys()方法获取Set<SelectionKey>，一个SelectionKey表达一个有事件发生的Channel以及该Channel上的事件类型。上图的“Synchronous Event Demultiplexer ---notifies--> Handle”的流程如果是对的，那内部实现应该是select()方法在事件到来后会先设置Handle的状态，然后返回。不了解内部实现机制，因而保留原图。

**Initiation Dispatcher**：用于管理Event Handler，即EventHandler的容器，用以注册、移除EventHandler等；另外，它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event()方法。

**Event Handler**：定义事件处理方法：handle\_event()，以供InitiationDispatcher回调使用。

**Concrete Event Handler（具体事件句柄）**：事件EventHandler接口，实现特定事件处理逻辑。

**Demultiplexing（分离）**：一种将到来的数据从输入端口路由到接收方的机制。输入端口和接收方之间为一对多的关系。经常需要对到来的事件和数据流进行分离。相反的操作成为多路复用（Multiplexing）。

### 3、代码实现

## 2.6、半同步半异步I/O模式

Half Sync/Half Async (HS-HA)

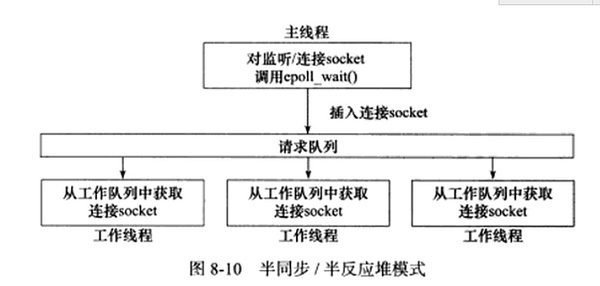
### 2.6.1、背景介绍

半同步半异步

其实半同步半异步的一种变体，其中一个异步线程，充当主线程，用来监听IO事件或者链接请求，当有链接请求到来或者IO事件到来时，可以该异步线程负责将该就绪socket放入到请求队列中，根据算法或者事件来选择一个工作线程来处理该socket事件

半同步半反应堆：

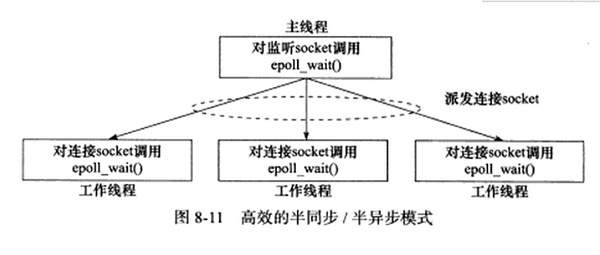
其和半同步和半异步的唯一却别就是，工作线程的选择，半同步半异步是根据算法和时间来选择由那个工作线程负责处理到达请求队列的socket事件，而半同步半反应堆则是由许多工作线程共同来竞争，胜者负责处理该socket事件，这是典型的Reactor模式，因为数据的读写是由工作线程也就事件处理器自己来完成，如我们在设置为，由异步主线程自己负责读取数据，然后将读完成事件socket和读取完成的数据一起封装后发送到请求队列中，由工作线程负责处理已经被读取过来的数据，这就是proator模式，这也更充分的说明了Reactor和Proactor的最本质的区别就是究竟由谁来负责数据的读写。



半同步/半反应堆模式因为也是主线程和工作线程共享任务队列，所以也会存在与半同步/半异步模式一样的问题，就是每次对队列进行操作，都需要进行加锁，从而消耗cpu的时间。而且一个线程同一时间只能处理一个客户请求，如果队列中积累了很多任务，增加工作线程的话，工作线程的切换也会耗费大量cpu时间

高效模式：

在每个工作线程中利用epollO复用技术，可以同时监听多个事件，异步主线程负责监听链接事件，当事件到来后，传入到工作线程，工作线程将该socket注册到自己的epoll内核注册表中，这样，这样每个工作线程都能处理多个客户连接了



领导者追随者模式：

若干个线程组成线程池，其中一个线程作为领导者用于监听IO事件，其他线程睡眠，当有IO事件到来是，当前领导者从线程池中唤醒一个线程，作为新的领导者，继续监听新的IO时间到来，而原来的领导者，转而去处理自己监听到的IO事件，当处理完毕后，加入到追随者中，等待再次被选中成为领导者。

领导者/追随者模式最大的优点在于，它是自己监听I/O事件并处理客户请求，也就是说从接收到处理都是在同一线程中完成，所以不需要在线程之间传递任何额外的数据，也不用在线程间同步对请求队列的访问。但是它也有明显的缺点，就是只支持一种事件源集合，所以导致它不能像上述那样让每个线程独立的管理多个客户连接

### 2.6.2、半同步半异步线程池

线程池的好处：

1、对于多核处理器，由于线程会被分配到多个CPU，会提高并行处理的效率。

2、每个线程独立阻塞，可以防止主线程被阻塞而使主流程被阻塞，导致其他的请求得不到响应的问题。

线程池分为半同步半异步线程池和领导者追随线程池。半同步半异步线程池实现上更简单，使用也比较多，也比较方便。

半同步半异步线程池分成三层。第一层是同步服务层，它处理来自上层的任务请求，上层的请求可能是并发的，这些请求不是马上就会被处理，而是将这些任务放到一个同步排队层中，等待处理。第二层是同步排队层，来自上层的任务请求都会加到排队层中等待处理。第三层是异步服务层，这一层会有多个线程同时处理排队层中的任务，异步服务层从同步排队层中取出任务并行的处理。

这三层的结构可以最大程度上处理上层的并发请求。对于上层来说只要将任务丢到同步队列中就行了。至于任务具体怎么处理，都是靠异步服务层的多线程异步并行来完成的。

三层结构中，排队层居于核心地位，因为上层会将任务加到排队层中，异步服务层同时也会取出任务，这里有一个同步的过程。在实现时，排队层就是一个同步队列，允许多个线程同时去添加或取出任务，并且要保证操作过程是安全的。线程池有两个活动过程，一个是往同步队列中添加任务的过程，一个是从同步队列中取任务的过程。

同步队列的实现

同步队列的主要作用：

1、保证队列中共享数据的线程安全。

2、为上一层同步服务层提供添加新任务的接口。

3、为下一层异步服务层提供取任务的接口。

4、队列中任务数的控制（任务数的上限）。

线程池的实现

类似于生产者-消费者模式，同步层是生产者，不断将新任务丢到排队层中，因此线程池需要提供一个添加新任务的接口供生产者使用；消费者是异步层，具体是由线程池中预先创建的线程去处理排队层中的任务。

### 2.6.3、HS-HA.pdf

（自己翻译的HS-HA.pdf文档关键部分）

The Half-Sync/Half-Async architectural pattern decouples synchronous tasks from asynchronous tasks in complex concurrent systems.

**Solution**

半同步半异步模式分成三层：

同步层、异步层、同步队列

高层任务，比如数据库查询、文件传输，为了简化并行编程，同步任务在独立的线程或进程中运行。

底层任务，比如网络接口、服务中断，异步是为了实现高质量服务。

同步队列，是为了在同步和异步层进行任务同步。

**Structure**

半同步半异步模式中包含以下各部分：

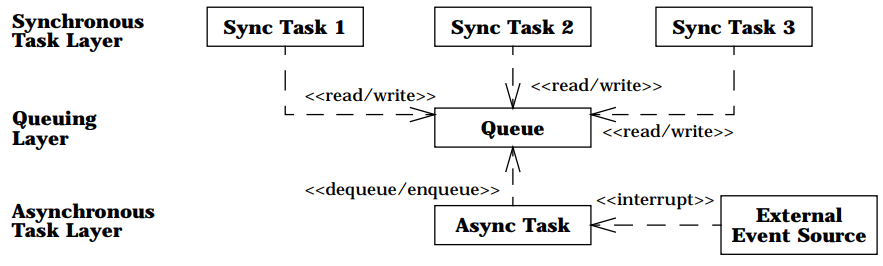
**同步任务层**执行高层处理任务。同步层的任务运行在独立的线程或进程中，它们有自己的运行时栈和寄存器。因此，当执行同步操作时，比如I/O，它们可以阻塞。举例说，为了高层任务，应用程序可以使用read和write系统调用去执行I/O同步。

**异步任务层**执行底层处理任务，特别是来自多个外部的I/O。异步层的任务没有一个专用的运行时栈或寄存器。因此，当它们执行异步操作时不能无限期的阻塞。举例说，I/O硬盘和操作系统内核的进程协议，是典型的异步任务，通常在中断处理中。

**队列层**提供一个缓冲区在同步任务层和异步任务层之间。异步任务生产的消息缓存在同步队列中，同步任务顺序的取出消息，反之亦然。另外，同步队列层负责通知这一层中的任务开始执行，当消息从另一层中传送给它们时。

**外部I/O源**产生被异步任务层接收和处理的事件。例如，网络接口，磁盘控制器，终端对操作系统来说都是通用的外部I/O源。

类图描述：



### 2.6.3、C++11实现

SyncQueue示例

|  |
| --- |
| #ifndef SYNCQUEUE\_HPP  #define SYNCQUEUE\_HPP  #include <list>  #include <mutex>  #include <thread>  #include <condition\_variable>  #include <iostream>  using namespace std;  template <typename T>  class SyncQueue  {  public:  SyncQueue(int maxSize):m\_maxSize(maxSize), m\_needStop(false)  {}    void Put(const T& x)  {  Add(x);  }    void Put(T&& x)  {  Add(forward<T>(x));  }    void Take(list<T>& list)  {  unique\_lock<mutex> locker(m\_mutex);  m\_notEmpty.wait(locker, [this]{return m\_needStop||NotEmpty();});  if(m\_needStop)  {  return;  }  list = move(m\_queue);  m\_notFull.notify\_one();  }    void Take(T& t)  {  unique\_lock<mutex> locker(m\_mutex);  m\_notEmpty.wait(locker, [this]{return m\_needStop || NotEmpty();});  if(m\_needStop)  {  return;  }  t = m\_queue.front();  m\_queue.pop\_front();  m\_notFull.notify\_one();  }    void Stop()  {  lock\_guard<mutex> locker(m\_mutex);  m\_needStop = true;    m\_notFull.notify\_all();  m\_notEmpty.notify\_all();  }    bool Empty()  {  lock\_guard<mutex> locker(m\_mutex);  return m\_queue.empty();  }    bool Full()  {  lock\_guard<mutex> locker(m\_mutex);  return m\_queue.size()== m\_maxSize;  }    size\_t Size()  {  lock\_guard<mutex> locker(m\_mutex);  return m\_queue.size();  }    int Count()  {  return m\_queue.size();  }    private:  bool NotFull() const  {  bool full = m\_queue.size() >= m\_maxSize;  if(full)  {  cout << "buffer full ,waiting..." << endl;  }  return !full;  }    bool NotEmpty() const  {  bool empty = m\_queue.empty() ;  if(empty)  {  cout << "buffer empty ,waiting... AsSync thread id: " << this\_thread::get\_id() << endl;  }  return !empty;  }    template<typename F>  void Add(F&& x)  {  unique\_lock<mutex> locker(m\_mutex);  m\_notFull.wait(locker, [this]{return m\_needStop || NotFull();});  if(m\_needStop)  return;  m\_queue.push\_back(forward<F>(x));  m\_notEmpty.notify\_one();  }    private:  list<T> m\_queue;  mutex m\_mutex;  condition\_variable m\_notEmpty;  condition\_variable m\_notFull;  int m\_maxSize;  bool m\_needStop;    };  #endif |

ThreadPool

|  |
| --- |
| #ifndef THREADPOOL\_HPP  #define THREADPOOL\_HPP  #include <list>  #include <thread>  #include <functional>  #include <iostream>  #include <memory>  #include <atomic>  #include "SyncQueue.hpp"  const int MaxTaskCount = 100;  class ThreadPool  {  public:  using Task = function<void()>;  ThreadPool(int numThread = thread::hardware\_concurrency()):m\_queue(MaxTaskCount)  {  Start(numThread);  }  ~ThreadPool()  {  Stop();  }    void Stop()  {  call\_once(m\_flag, [this]{StopThreadGroup();});  }    void AddTask(Task&& task)  {  m\_queue.Put(forward<Task>(task));  }    void AddTask(const Task& task)  {  m\_queue.Put(task);  }  private:  void Start(int numThread)  {  m\_running = true;  for (int i = 0; i < numThread; i++)  {  m\_threadgroup.push\_back(make\_shared<thread>(&ThreadPool::RunInThread, this));  }  }    void RunInThread()  {  while(m\_running)  {  list<Task>list;  m\_queue.Take(list);    for(auto& task : list)  {  if(!m\_running)  {  return;  }  task();  }  }  }    void StopThreadGroup()  {  m\_queue.Stop();  m\_running = false;    for(auto thread : m\_threadgroup)  {  if(thread)  {  thread->join();  }  }  m\_threadgroup.clear();  }    private:  list<shared\_ptr<thread>> m\_threadgroup;  SyncQueue<Task> m\_queue;  atomic\_bool m\_running;  once\_flag m\_flag;    };  #endif |

main

|  |
| --- |
| #include "SyncQueue.hpp"  #include "ThreadPool.hpp"  #include <iostream>  using namespace std;  void TestThreadPool()  {  ThreadPool pool;    thread thd1([&pool]{  for(int i = 0; i < 10; i++)  {  auto thdId = this\_thread::get\_id();  pool.AddTask([thdId]{  cout << "Sync thread 1 Id: " << thdId << endl;  });  }  });    thread thd2([&pool]{  for(int i = 0; i < 10; i++)  {  auto thdId = this\_thread::get\_id();  pool.AddTask([thdId]{  cout << "Sync thread 1 Id: " << thdId << endl;  });  }  });    this\_thread::sleep\_for(std::chrono::seconds(2));  getchar();  pool.Stop();  thd1.join();  thd2.join();  }  int main()  {  TestThreadPool();  } |

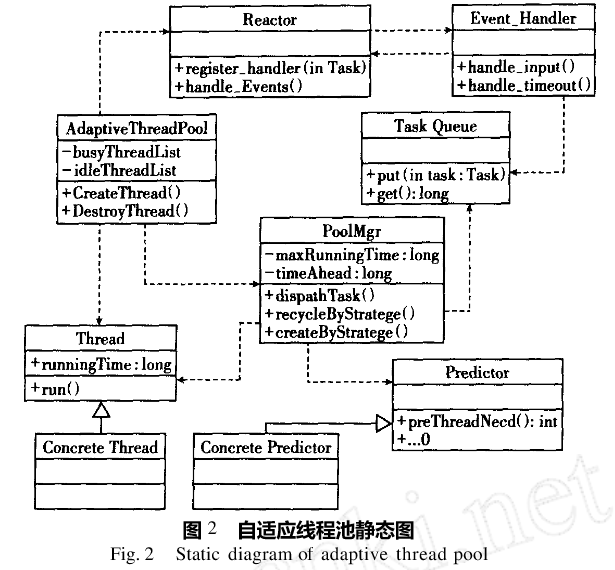
2.6.4、C++98实现

# 2.7、对象池模式的自适应线程池实现

## 1、自适应线程池的结构设计

线程池可以看作是对象池的一个实例，用来复用线程。自适应线程池与其他线程池相比多出了一个Predictor，负责对某一时间所需的线程数量进行预测，根据预测数量对线程池中实际存在的线程数量进行调整，从而达到自适应的效果。

线程池类图：



自适应线程池运行在服务器上，通过使用反应器（Reactor）结构化模式来多路分解并分配从一个或多个客户机发送给应用程序的服务请求。反应器将接收的请求存储在任务队列（TaskQueue）中。例如，反应器可以通过select检测socket句柄集合上的IO事件（如Connect），一旦检测到该事件，就将accept函数返回的与客户端连接的socket句柄存放在TaskQueue中。线程池管理器（PoolMgr）从TaskQueue中取出任务（包含socket句柄）分配给某一个线程来为客户提供服务。从而形成一个半同步/半异步的并发模式。需要指出的是，任务队列中存储的是任务（Task）的派生类，比如可以是包含了socket句柄的Task派生类。Task抽象类提供了统一的任务接口。

## 2、线程池的管理

线程池是由PoolMgr进行管理的。在线程池中，有2个队列，分别是空闲的线程队列和在运行的线程队列。由于不同的空闲线程都是一个计算机上的资源，不需要考虑任务均衡问题，所以PoolMgr在分配任务时，从空闲队列中取出第一个线程分配任务，无需遍历队列，从而使得分配任务的时间复杂度为O(1)，与队列大小无关。当线程执行完任务后，自己加入到空闲队列的队尾，无需PoolMgr的参与。如果空闲线程队列为空，任务队列不空，而且当前线程的总数量小于系统运行的最大线程数量，则创建新的线程，对任务进行处理；如果当前线程的总数量达到系统运行的最大线程数量，则不再创建新的线程，而是等待，等到空闲队列不为空时再进行任务分配。

PoolMgr分配线程时将线程从空闲队列中取出添加到已用队列中。每个线程都有一个计时器，记录已经运行的时间。如果某一个线程的运行时间超过了最大允许运行时间，则可以认为该线程出现了死锁或者因其他异常情况被挂起，PoolMgr则将它强制结束，以防止该线程被永远挂起，不能重新利用。

## 3、线程的创建与销毁策略

与其他的线程池不同，自适应线程池中线程的创建并不仅仅依赖于新的任务已经到达而且空闲线程队列为空这种情况，在另一种情况下PoolMgr也要创建线程，即：Predictor预测在t2时刻系统将需要n1个线程才能满足需要，PoolMgr会在t1（t1<t2）时刻将系统存在的线程数量n2与系统将需要的线程数量n1进行比较，如果n1>n2，则创建n1-n2个线程，之所以要t1<t2是因为如果到t2时刻再创建，就退化成为传统的线程池，失去预测的意义。

销毁线程的策略是：Predictor预测再t1时刻系统将需要n1个线程才能满足需要，PoolMgr会在t1时刻将系统存在的线程数量n2与系统将需要的线程数n1进行比较，如果n1<n2，则销毁n2-n1个空闲线程，如果空闲的线程小于n2-n1，则只是将空闲的线程销毁。基于上述线程的创建于销毁策略，线程池中不再设有最小线程数量的限制，而只有最大线程的限制，线程的数量再最大数量限制的范围内按需创建和销毁。

## 实现目标

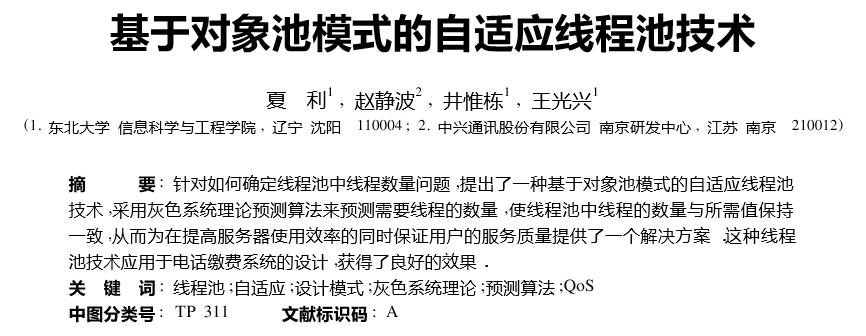
预计不会添加自适应功能，仅按传统线程池实现方式实现该论文。

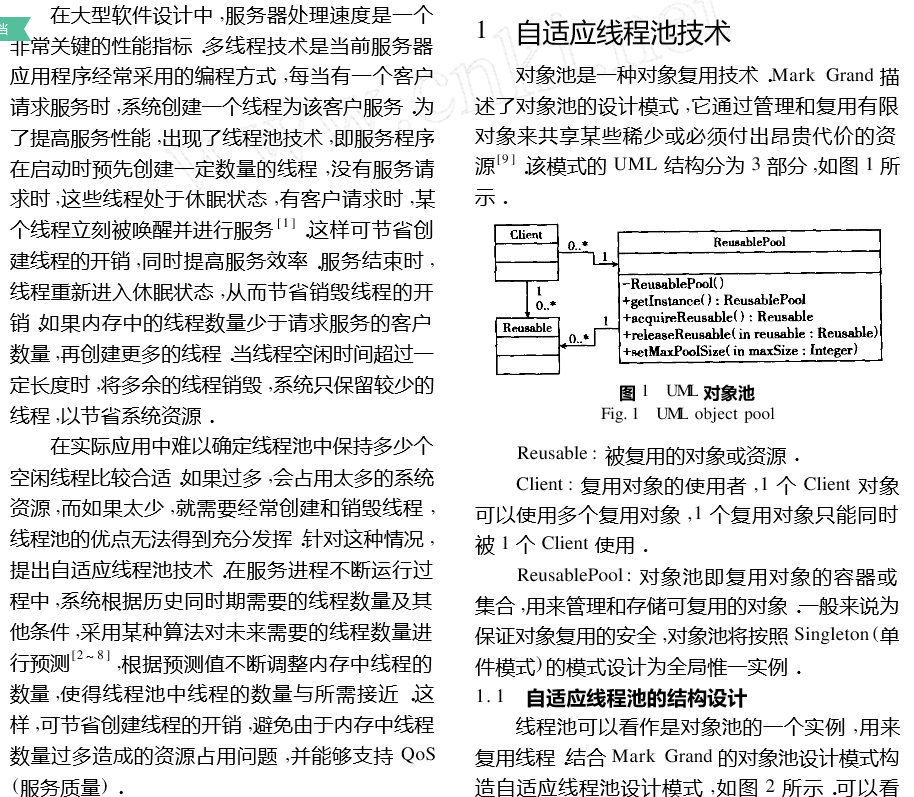
再Linux系统下使用C++98实现。

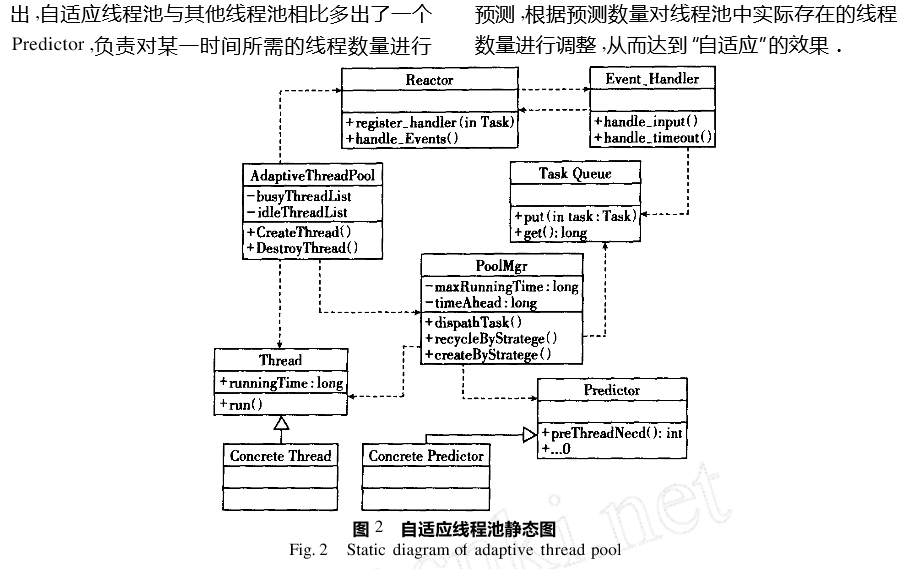
# 附件

## 附件A

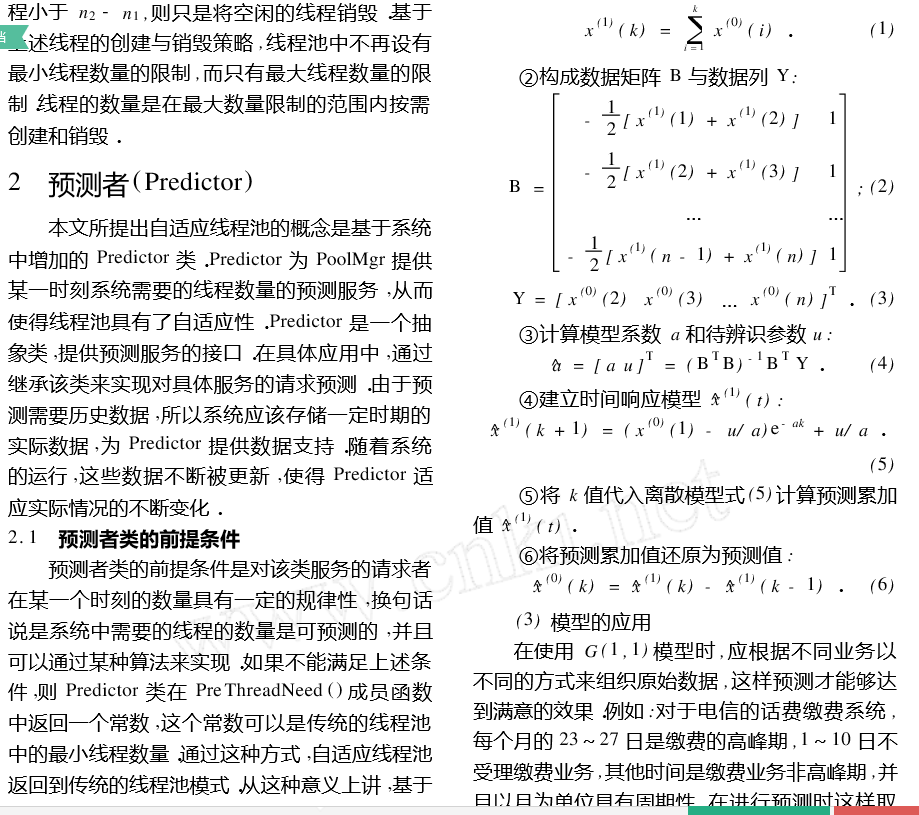
《基于对象池模式的自适应线程池技术》论文技术

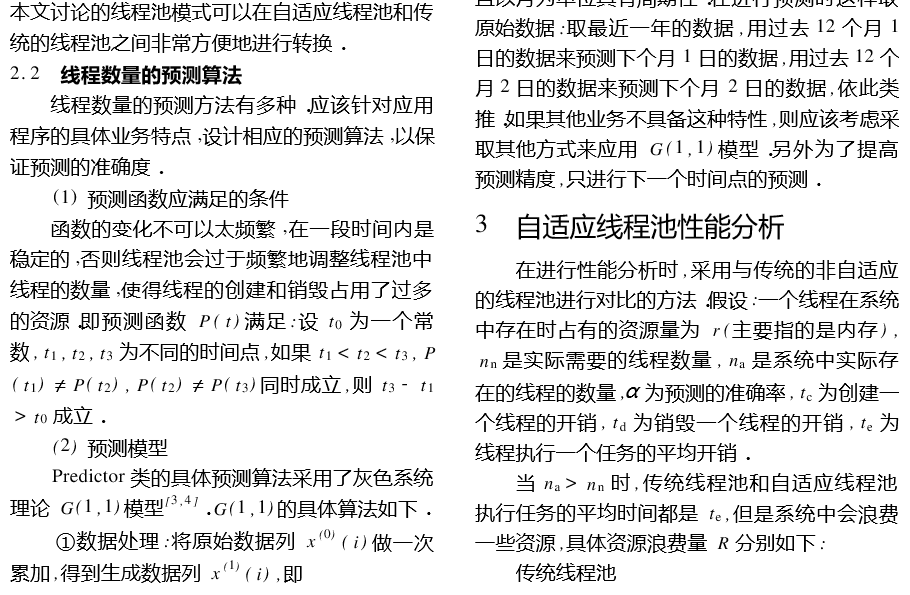


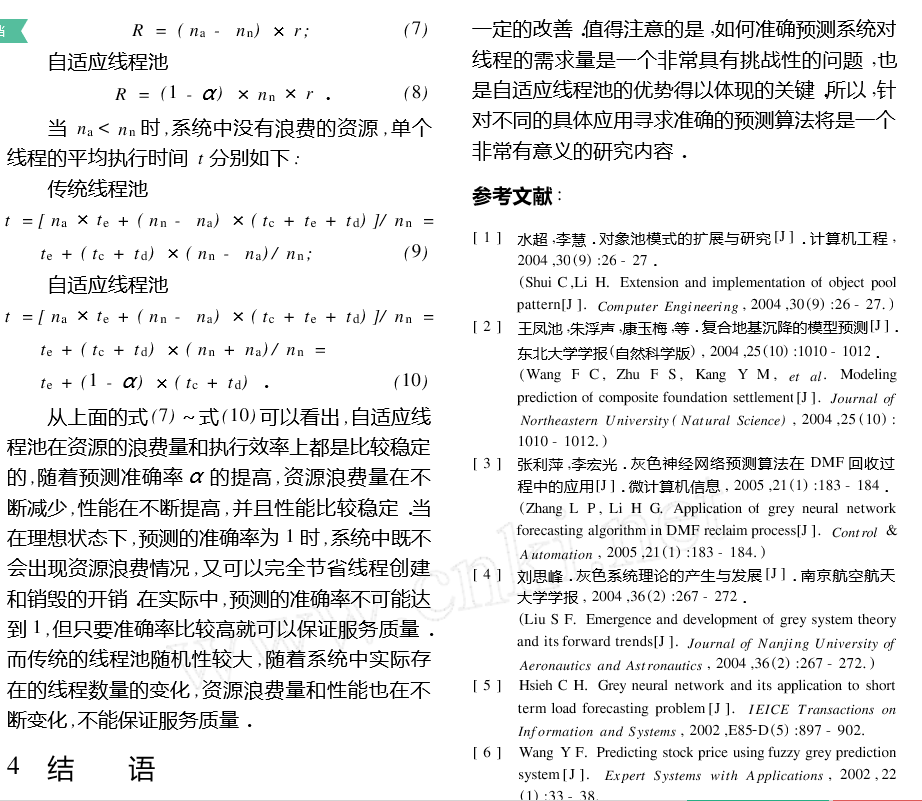


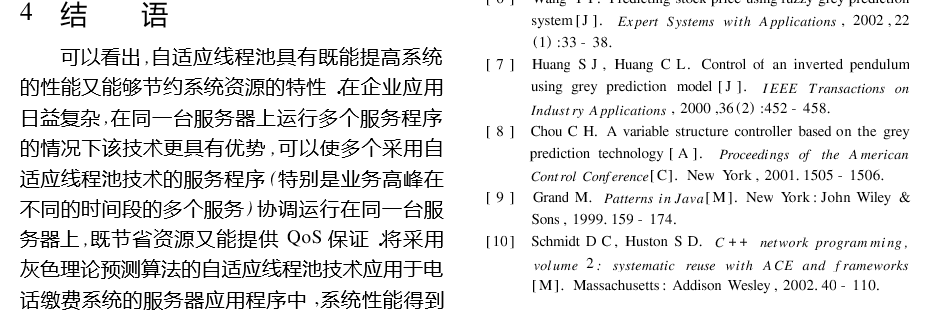












# 参考文献

1、https://wenku.baidu.com/link?url=d-Uvr2NIFpCRtfrH6fDashAvcuLZ3MOdbuWmBL\_oJ24Hm68SzOEK2n2X\_ry9Og8t-ZTKf262OOu9NEMx0BIuWrVBB15MN6ZzMKRXcuO0E0W

2、夏利. 基于对象池模式的自适应线程池技术[J].东北大学学报，2006年10月

3、http://www.blogjava.net/DLevin/archive/2015/09/02/427045.html

4、https://en.wikipedia.org/wiki/Reactor\_pattern

5、https://www.dre.vanderbilt.edu/~schmidt/PDF/HS-HA.pdf