第8章 服务配置器(Service Configurator)：用于服务动态配置的模式

许多分布式系统都含有一组*全局*服务。应用开发者可以调用这些服务来帮助他满足分布式开发的需求。在构造分布式应用时，需要像名字服务、远程终端访问服务、登录和时间时间服务这样的全局服务。构造这些服务的一种办法是将每个服务编写成单独的程序。随后这些服务程序就在它们自己的私有进程中执行和运行。但是，这样的方法会导致配置的噩梦。管理员需要管理每一个节点，依据当前的用户需求和策略来执行服务程序。如果需要增加新服务，或是需要移除旧服务，管理员就必须将时间花费在每台机器上、重新进行配置。此外，这样的配置是*静态的*。要进行重配置，管理员需要人工地终止服务（通过杀掉服务进程），随后重启一个替换服务。同样，在机器上运行的服务也可能不被*任何*应用所使用。显然，这样的方法是低效的和不合需要的。

如果服务可以被*动态地*启动、移除、挂起和恢复，那将会方便得多。这样，服务开发者就不必再担心配置的服务。他所需关心的是服务如何完成工作。管理员就可以在应用中增加或替换新服务，而*不用*重新编译或关闭服务进程。

服务配置器模式可以完成所有这些任务。它使服务的*实现*与*配置*去耦合。无需关闭服务器，就可以在应用中增加新服务和移除旧服务。在大多数情况下，提供服务的服务器都被实现为看守（daemon）进程。

8.1 构架组件

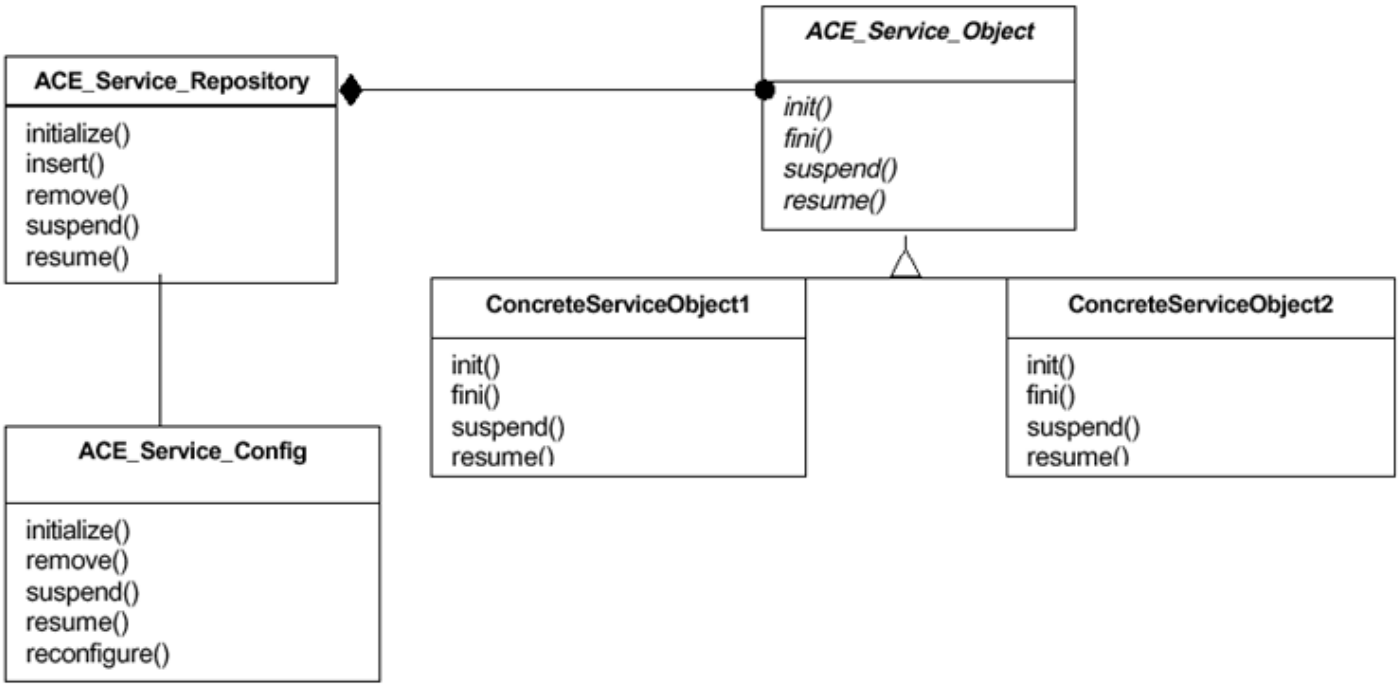

图8-1 ACE服务配置器中的组件及其关系

ACE中的服务配置器由以下组件组成：

- 名为ACE_Service_Object的抽象类。应用开发者必须从它派生出子类，以创建他自己的应用特有的具体服务对象（Service Object）。
- 应用特有的具体服务对象。
- 服务仓库ACE_Service_Repository。它记录服务器所运行的和所知道的服务。
- ACE_Service_Config。它是整个服务配置器框架的应用开发接口。
- 服务配置文件。该文件含有所有服务对象的配置信息。其缺省的名字是svc.conf。当你的应用对ACE_Service_Config发出open()调用时，服务配置器框架会读取并处理你写在此文件中的所有配置信息，随后相应地配置应用。

ACE_Service_Object包括了一些由框架调用的方法，用于服务要启动（init()）、停止（fini()）、挂起（suspend()）或是恢复（resume()）时。ACE_Service_Object派生自ACE_Shared_Object和ACE_Event_Handler。ACE_Shared_Object在应用想要使用操作系统的动态链接机制来进行加载时被用作抽象基类。ACE_Event_Handler已在对反应堆的讨论中进行了介绍。当开发者想要他的类响应来自反应堆的事件时，他就从ACE_Event_Handler派生他的子类。

为什么服务对象要从ACE_Event_Handler继承？用户发起重配置的一种方法是生成一个信号；当这样的信号事件发生时，反应堆被用于处理信号，并向ACE_Service_Config发出重配置请求。除此而外，软件的重配置也可能在某事件产生后发生。因而所有的服务对象都被构造为能对事件进行处理。

服务配置文件有它自己的简单脚本，用于描述你想要服务怎样启动和运行。你可以定义你是想要增加新服务，还是挂起、恢复或移除应用中现有的服务。另外还可以给服务发送参数。服务配置器还允许进行基于ACE的*流*（stream）的重配置。我们将在讨论了ACE流构架之后再来更多地讨论这一点。

8.2 定义配置文件

服务配置文件指定在应用中哪些服务要被加载和启动。此外，你可以指定哪些服务要被停止、挂起或恢复。还可以发送参数给你的服务对象的init()方法。

8.2.1 启动服务

服务可以被静态或动态地启动。如果服务要动态启动，服务配置器实际上会从共享对象库（也就是，动态链接库）中加载服务对象。为此，服务配置器需要知道哪个库含有此对象，并且还需要知道对象在该库中的名字。因而，在你的代码文件中你必须通过你需要记住的名字来实例化服务对象。于是动态服务会这样被配置：

```
dynamic service_name type_of_service * location_of_shared_lib:name_of_object “parameters”
```

而静态服务这样被初始化：

```
static service_name "parameters_send_to_service_object"
```

8.2.2 挂起或恢复服务

如刚才所提到的，你在启动服务时分配给它一个名字。这个名字随后被用于挂起或恢复该服务。于是要挂起服务，你所要做的就是在svc.conf文件中指定：

```
suspend service_name
```

这使得服务对象中的suspend()方法被调用。随后你的服务对象就应该挂起它自己（基于特定服务不同的“挂起”含义）。

如果你想要恢复这个服务，你所要做的就是在svc.conf文件中指定：

```
resume service_name
```

这使得服务对象中的resume()方法被调用。随后你的服务对象就应该恢复它自己（基于特定服务不同的“恢复”含义。）

8.2.3 停止服务

停止并移除服务（如果服务是动态加载的）同样是很简单的操作，可以通过在你的配置文件中指定以下指令来完成：

```
remove service_name
```

这使得服务配置器调用你的应用的fini()方法。该方法应该使此服务停止。服务配置器自己会负责将动态对象从服务器的地址空间里解除链接。

8.3 编写服务

为服务配置器编写你自己的服务相对比较简单。你可以让这个服务做任何你想做的事情。唯一的约束是它应该是ACE_Service_Object的子类。所以它必须实现init()和fini()方法。在ACE_Service_Config被打开（open()）时，它读取配置文件（也就是svc.conf）并根据这个文件来对服务进行初始化。一旦服务被加载，它会调用该服务对象的init()方法。类似地，如果配置文件要求移除服务，fini()方法就会被调用。这些方法负责分配和销毁服务所需的任何资源，比如内存、连接、线程等等。在svc.conf文件中指定的参数通过服务对象的init()方法来传入。

下面的例子演示一个派生自ACE_Task_Base的服务。ACE_Task_Base类含有activate()方法，用于在对象里创建线程。（在“任务和主动对象”一章中讨论过的ACE_Task派生自ACE_Task_Base，并包括了用于通信目的的消息队列。因为我们不需要我们的服务与其它任务通信，我们仅仅使用ACE_Task_Base来帮助我们完成工作。）更多详细信息，请阅读“任务和主动对象”一章。该服务是一个“无为”（do-nothing）的服务，一旦启动，它只是周期性地广播当天的时间。

例8-1a

```
//The Services Header File.
#if !defined(MY_SERVICE_H)
#define MY_SERVICE_H

#include "ace/OS.h"
#include "ace/Task.h"
#include "ace/Synch_T.h"

// A Time Service class. ACE_Task_Base already derives from
//ACE_Service_Object and thus  we don’t have to subclass from
//ACE_Service_Object in this case.
class TimeService: public ACE_Task_Base
{
public:
virtual int init(int argc, ASYS_TCHAR *argv[]);
virtual int fini(void);
virtual int suspend(void);
virtual int resume(void);
virtual int svc(void);

private:
int canceled_;
ACE_Condition<ACE_Thread_Mutex> *cancel_cond_;
ACE_Thread_Mutex *mutex_;
};

#endif
```

相应的实现如下所述：在时间服务接收到init()调用时，它在任务中启用（activate()）一个线程。这将会创建一个新线程，其入口为svc()方法。在svc()方法中，该线程将会进行循环，直到它看到canceled_标志被设置为止。此标志在服务配置构架调用fini()时设置。但是，在fini()方法返回底层的服务配置框架**之前**，它必须**确定**在底层的线程已经终止。因为服务配置器将要实际地卸载含有TimeService的共享库，从而将TimeService对象从应用进程中删除。如果在此**之前**线程并未终止，它将会对已经被服务配置器“蒸发”的代码发出调用！我们当然不需要这个。为了确保线程在服务配置器“蒸发”TimeService对象**之前终止**，程序使用了条件变量。（要更多地了解怎样使用条件变量，请阅读有关线程的章节）。

例8-1b

```
#include "Services.h"

int TimeService::init(int argc, char *argv[])
{
ACE_DEBUG((LM_DEBUG,"(%t)Starting up the time Service\n"));
mutex_ = new ACE_Thread_Mutex;
cancel_cond_ = new ACE_Condition<ACE_Thread_Mutex>(*mutex_);
activate(THR_NEW_LWP|THR_DETACHED);
return 0;
}

int TimeService::fini(void)
{
ACE_DEBUG((LM_DEBUG,
      "(%t)FINISH!Closing down the Time Service\n"));

//All of the following code is here to make sure that the
//thread in the task is destroyed before the service configurator
//deletes this object.
canceled_=1;
mutex_->acquire();
while(canceled_)
      cancel_cond_->wait();
mutex_->release();
ACE_DEBUG((LM_DEBUG,"(%t)Time Service is exiting \n"));

return 0;
}

//Suspend the Time Service.
int TimeService::suspend(void)
{
ACE_DEBUG((LM_DEBUG,"(%t)Time Service has been suspended\n"));
int result=ACE_Task_Base::suspend();
return result;
}

//Resume the Time Service.
int TimeService::resume(void)
{
```

```
ACE_DEBUG((LM_DEBUG,"(%t)Resuming Time Service\n"));
int result=ACE_Task_Base::resume();
return result;
}

//The entry function for the thread. The tasks underlying thread
//starts here and keeps sending out messages. It stops when:
// a) it is suspeneded
// b) it is removed by fini(). This happens when the fini() method
// sets the cancelled_ flag to true. Thus causes the TimeService
// thread to fall through the while loop and die. Before dying it
// informs the main thread of its imminent death. The main task
// that was previously blocked in fini() can then continue into the
// framework and destroy the TimeService object.
int TimeService::svc(void)
{
char *time = new char[36];
while(!canceled_)
{
      ACE::timestamp(time,36);
      ACE_DEBUG((LM_DEBUG,"(%t)Current time is %s\n",time));
      ACE_OS::fflush(stdout);
      ACE_OS::sleep(1);
}

//Signal the Service Configurator informing it that the task is now
//exiting so it can delete it.
canceled_=0;
cancel_cond_->signal();
ACE_DEBUG((LM_DEBUG,
      "Signalled main task that Time Service is exiting \n"));
return 0;
}

//Define the object here
TimeService time_service;
```

下面是一个简单的、只是用于启用时间服务的配置文件。可以去掉注释#号来挂起、恢复和移除服务。

例8-1c

```
# To configure different services, simply uncomment the appropriate
#lines in this file!
#resume TimeService
#suspend TimeService
#remove TimeService
#set to dynamically configure the TimeService object and do so without
#sending any parameters to its init method
dynamic TimeService Service_Object * ./Server:time_service ""
```

最后，下面是启动服务配置器的代码段。这些代码还设置了一个信号处理器对象，用于发起重配置。该信号处理器已被设置成响应SIGWINCH信号（在窗口发生变化时产生的信号）。在启动服务配置器之后，应用进入一个反应式循环，等待SIGWINCH信号事件发生。一旦事件发生，就会回调事件处理器，由它调用ACE_Service_Config的reconfigure()方法。如先前所讲述的，在此调用发生时，服务配置器重新读取配置文件，并处理用户放在其中的任何新指令。例如，在动态启动TimeService后，在这个例子中你可以改变svc.conf文件，只留下一个挂起命令在里面。当配置器读取它时，它将调用TimeService的挂起方法，从而使它挂起它的底层线程。类似地，如果稍后你又改变了svc.conf，要求恢复服务，配置器就会调用TimeService::resume()方法，从而恢复先前被挂起的线程。

例8-1d

```
#include "ace/OS.h"
#include "ace/Service_Config.h"
#include "ace/Event_Handler.h"
#include <signal.h>
//The Signal Handler which is used to issue the reconfigure()
//call on the service configurator.
class Signal_Handler: public ACE_Event_Handler
{
public:
int open()
{
      //register the Signal Handler with the Reactor to handle
      //re-configuration signals
      ACE_Reactor::instance()->register_handler(SIGWINCH,this);
      return 0;
}

int handle_signal(int signum, siginfo*,ucontext_t *)
{
      if(signum==SIGWINCH)
            ACE_Service_Config::reconfigure();
      return 0;
}
```

```
};

int main(int argc, char *argv[])
{
//Instantiate and start up the Signal Handler. This is uses to
//handle re-configuration events.
Signal_Handler sh;
sh.open();

if (ACE_Service_Config::open (argc, argv) == -1)
      ACE_ERROR_RETURN ((LM_ERROR,
            "%p\n","ACE_Service_Config::open"),-1);

      while(1)
      ACE_Reactor::instance()->handle_events();
}
```

8.4 使用服务管理器

ACE_Service_Manager是可用于对服务配置器进行远程管理的服务。它目前可以接受两种类型的请求。其一，你可以向它发送“help”消息，列出当前被加载进应用的所有服务。其二，你可以向服务管理器发送“reconfigure”消息，从而使得服务配置器重新配置它自己。

下面的例子演示了一个客户，它向服务管理器发送这两种类型的命令。

例8-2

```
#include "ace/OS.h"
#include "ace/SOCK_Stream.h"
#include "ace/SOCK_Connector.h"
#include "ace/Event_Handler.h"
#include "ace/Get_Opt.h"
#include "ace/Reactor.h"
#include "ace/Thread_Manager.h"

#define BUFSIZE 128

class Client: public ACE_Event_Handler
{
public:
~Client()
{
      ACE_DEBUG((LM_DEBUG,"Destructor \n"));
```

```
}

//Constructor
Client(int argc, char *argv[]): connector_(), stream_()
{
      //The user must specify address and port number
      ACE_Get_Opt get_opt(argc,argv,"a:p:");
      for(int c;(c=get_opt())!=-1;)
      {
            switch(c)
            {
            case 'a':
                        addr_=get_opt.optarg;
                        break;
                        case 'p':
                        port_= ((u_short)ACE_OS::atoi(get_opt.optarg));
                        break;
                        default:
                        break;
            }
      }

      address_.set(port_,addr_);
}

//Connect to the remote machine
      int connect()
      {
      connector_.connect(stream_,address_);
      ACE_Reactor::instance()->
            register_handler(this,ACE_Event_Handler::READ_MASK);
      return 0;
}

//Send a list_services command
int list_services()
{
      stream_.send_n("help",5);
      return 0;
}

//Send the reconfiguration command
      int reconfigure()
```

```
      {
      stream_.send_n("reconfigure",12);
      return 0;
}

//Handle both standard input and remote data from the
//ACE_Service_Manager
int handle_input(ACE_HANDLE h)
{
      char buf[BUFSIZE];

      //Got command from the user
      if(h== ACE_STDIN)
      {
            int result = ACE_OS::read (h, buf, BUFSIZ);
            if (result == -1)
                        ACE_ERROR((LM_ERROR,"can't read from STDIN"));
            else if (result > 0)
            {
                        //Connect to the Service Manager
                        this->connect();
                        if(ACE_OS::strncmp(buf,"list",4)==0)
                              this->list_services();
                        else if(ACE_OS::strncmp(buf,"reconfigure",11)==0)
                              this->reconfigure();
            }

            return 0;
      }
      //We got input from remote
            else
            {
            switch(stream_.recv(buf,BUFSIZE))
            {
            case -1:
                        //ACE_ERROR((LM_ERROR,  "Error in receiving from
                        remote\n"));
                        ACE_Reactor::instance()->remove_handler(this,
                                    ACE_Event_Handler::READ_MASK);
                        return 0;
            case 0:
                              return 0;
```

```
            default:
                        ACE_OS::printf("%s",buf);
                        return 0;
            }
      }
}

//Used by the Reactor Framework
ACE_HANDLE get_handle() const
{
      return stream_.get_handle();
}

//Close down the underlying stream
int handle_close(ACE_HANDLE,ACE_Reactor_Mask)
{
      return stream_.close();
}

private:
ACE_SOCK_Connector connector_;
ACE_SOCK_Stream stream_;
ACE_INET_Addr address_;
char *addr_;
u_short port_;
};

int main(int argc, char *argv[])
{
Client client(argc,argv);

//Register the the client event handler as the standard
//input handler
ACE::register_stdin_handler(&client,
ACE_Reactor::instance(),
ACE_Thread_Manager::instance());

ACE_Reactor::run_event_loop();
}
```

在此例中，Client类是一个事件处理器，它处理两种类型的事件：来自用户的标准输入事件和来自ACE_Service_Manager的回复。如果用户输入“list”或“reconfigure”命令，相应的消息就被发送到远地的ACE_Service_manager。服务管理器随之回复以当前配置的服务的列表或是“done”（指示服务的重配置已经完成）。因为ACE_Service_Manager是一个服务，你可以在svc.conf文件中指定你是想要静态还是动态地加载此服务，并使用服务配置构架来将它加载入应用中。

例如，下面的命令指定在9876端口静态地启动服务管理器：

```
static ACE_Service_Manager “-p 9876”
```