缓冲区设计--环形队列

在程序的两个模块间进行通讯的时候，缓冲区成为一个经常使用的机制。

如上图，写入模块将信息写入缓冲区中，读出模块将信息读出缓冲区。
这样使得：

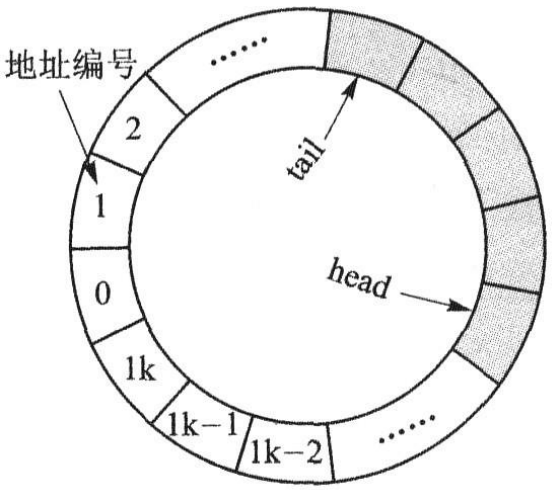
- 将程序清晰地划分模块，建立良好的模块化架构，使得写入和读出成为高聚合，低耦合的模块。
- 对于写入和读出的处理可能产生的快慢不均匀的情况进行平衡，使得整个处理的速度趋于平滑的均匀状态，避免出现读出模块处理的慢速使得写入模块等待使得响应速度下降的状况；同样，也避免写入模块的快慢不均匀，使得读出模块忙闲不一的情况。
- 可以增加处理的并发性。由于写入和读出模块的良好设计和划分，可以使得它们彼此隔离和独立，从而，使用线程和进程产生不同的并发处理，并通过缓冲区大小的调节，使得这个处理达到良好的匹配和运行状态。例如，写入模块可以有 N 个线程或进程，读出模块可以有 M 个线程和进程，缓存冲区可以配置 L 的大小。N、M、L 可以通过模拟试验设定适应具体应用的值。也可以建立一套自动调节的机制，当然，这样会造成设计的复杂性。

缓冲区显然不适合下面的情况：

- 数据的接收处理原本就是密切相关，难以划分模块。
- 处理中的模块间明显不存在处理不均匀的情况，或者不是主要问题。
- 需要同步响应的情况。显然，写入端只是将信息 push 到队列中，并不能得到读出端的处理响应信息，只能适合于异步的信息传递的情况。

缓冲区的设计：

- 缓冲区是一个先进先出队列。写入模块将信息插入队列；读出模块将信息弹出队列。
- 写入模块与读出模块需要进行信息的协调和同步。
- 对于多线程和多进程的写入或读出模块，写入模块间以及读出模块间需要进行临界区处理。



队列使用环形队列，如上图。环形队列的特点是，不需要进行动态的内存释放和分配，使用固定大小的内存空间反复使用。在实际的队列插入和弹出操作中，是不断交叉进行的，当 push 操作时，head 会增加；而 pop 操作时，tail 会增加。push 的速度快的时候，有可能追

上 tail，这个时候说明队列已经满了，不能再进行 push 的操作了，需要等待 pop 操作腾出队列的空间。当 pop 的操作快，使得 tail 追上 head，这个时候说明队列已空了，不能再进行 pop 操作了，需要等待 push 进来数据。

下面列出了一个环形队列类的的数据结构的源程序 。

```
/* LoopQue.h
  Author: ZhangTao
  Date: July 26, 2009
*/

# ifndef LoopQue_h
# define LoopQue_h

# include

namespace xtl  {

template<typename _Tp>
class LoopQue_impl  {
 public:
  static int addsize(int max_size) {
   return max_size * sizeof(_Tp);
  }

  LoopQue_impl(int msize) : max_size(msize), _front(0), _rear(0), _size(0) {}

  _Tp& front() { return data[_front]; }

  void push(const _Tp& value)  {
   data[_rear] = value;
   _rear = (_rear + 1) % max_size;
   _size++;
  }

  void pop()  {
   _front = (_front + 1) % max_size;
   _size--;
  }

  int check_pop(_Tp& tv)  {
   if ( empty() )
    return -1;

   tv = front();
   pop();
  }

  int check_push(const _Tp& value)  {
   if ( full() )
    return -1;

   push(value);
  }

  bool full() const  { return _size == max_size; }
  bool empty() const { return _size == 0; }
  int  size() const { return _size; }
  int  capacity() const { return max_size; }

 private:
  int32_t _front;  // front index
  int32_t _rear;   // rear index
  int32_t _size;   // queue data record number

  const int32_t max_size; // queue capacity

  _Tp  data[0];   // data record occupy symbol
};

template<typename _Tp>
struct LoopQue_allocate  {
 LoopQue_impl<_Tp>& allocate(int msize)   {
  char *p = new char[sizeof(LoopQue_impl<_Tp>) +
          LoopQue_impl< Tp>::addsize(msize)];
```

```
    return *(new (p) LoopQue_impl< Tp>(msize));
  }

  void deallocate(void *p)  {
    delete [] (char *)p;
  }
};

template<typename _Tp, typename Alloc = LoopQue_allocate<_Tp> >
class LoopQue  {
  public:
    typedef _Tp  value_type;

    LoopQue(int msize) : impl(alloc.allocate(msize)) {}
    ~LoopQue()  { alloc.deallocate((void *)&impl); }

    value_type& front() { return impl.front(); }
    const value_type& front() const { return impl.front; }

    void push(const value_type& value)  { impl.push(value); }
    void pop()  { impl.pop(); }

    int check_pop(value_type& tv)  { return impl.check_pop(tv); }
    int check_push(const value_type& value)  { return impl.check_push(value); }

    bool full() const  { return impl.full(); }
    bool empty() const { return impl.empty(); }
    int  size() const { return impl.size(); }

  private:
    Alloc alloc;
    LoopQue_impl<_Tp>& impl;
};

} // end of < namespacestl>

# endif  // end of
```

程序里定义了两个类 LoopQue_impl 及 LoopQueue。前者定义了环形队列的基本数据结构和实现，后者又进行了一次内存分配包装。

21 行的 LoopQue_impl 的构造函数是以队列的空间大小作为参数创建这个类的。也就是说，在类创建的时候，就决定了队列的大小。

63 行中定义的队列的数组空间为 _Tp data[0]。这似乎是一个奇怪的事情。事实上，这个空间大小应该是 max_size 个数组空间。但由于 max_size 是在类创建的时候确定的，在这里，data[0]只起到一个占位符的作用。所以，LoopQue_impl 这个类是不能直接使用的，需要正确的分配好内存大小，才能使用，这也是需要里另外设计一个类 LoopQueue 的重要原因之一。也许您会奇怪，为什么要这样使用呢？如果定义一个指针，例如：_Tp *data，然后在构造函数里面使用 data = new _Tp[max_size]，不是很容易吗？但是，不要忘记了，我们这个环形队列类有可能会是一个进程间共享类。例如，一个进程 push 操作，另一个进程 pop 操作。这样，这个类是需要建立在共享内存中的。而共享内存中的类的成员，如果包含有指针或者引用这样的类型，将给内存分配带来很大的麻烦。而我们这样以这个占位符的方式设计这个类，将减少这种麻烦和复杂性。

17 行的 addsize 的类函数确定了 LoopQue_impl 需要的另外的内存空间的大小。

LoopQueue 显示了怎样使用 LoopQue_impl，解决内存分配问题的。从 79 行的模版参数中，我们看到，除了缓冲区数据存放类型_Tp 的参数外，还有一个 Alloc 类型。这便是用于分配 LoopQue_impl 内存空间使用的模版类。

在 LoopQueue 的成员中，定义了 LoopQue_impl 的一个引用 impl（102 行）。这个引用便是指向使用 Alloc 分配空间得来的 LoopQue_impl 的空间。

Alloc 模版参数有一个缺省的定义值 LoopQue_allocate。从这个缺省的分配内存的类里，我们可以看到一个分配 LoopQue_impl 的实现样例，见 69 行：

```
char *p = new char[sizeof(LoopQue_impl<_Tp>) + LoopQue_impl<_Tp>::addsize(msize)];
return *(new (p) LoopQue_impl<_Tp>(msize));
```

这里，先根据 msize 分配好了靠虑到了 data[msize]的足够的内存，然后，再使用定位的 new 操作，将 LoopQue_impl 创建在这个内存区中。这样，LoopQue_impl 类就可以使用它其中的 _Tp data[0]的成员。实际上，这个成员已经有 _Tp data[msize]这样的空间了。这里，如果我们设计另外一个分配内存的类，例如，LoopQue_ShmAlloc。这个类是使用共享内存，并在其中建立 LoopQue_impl 类。这样我们就可以使用：

LoopQue<_Tp LoopQue_ShmAlloc>

来创建一个可以在进程间共享而进行通讯的环形队列类了。

至此，我们可以总结一下：

- 环形队列的数据结构和基本操作都是相当简单的。主要的操作就是 push 和 get。
- 环形队列可能需要线程或者进程间共享操作。尤其是进程间的共享，使得内存分配成为了一个相对复杂的问题。对于这个问题，我们将基础的数据结构定义为无指针和引用成员，适合于内存管理的类，然后又设计了一个代理的进行二次包装的类，将内存分配的功能作为模版可定制的参数。

这里，我们设计并实现了环形队列的数据结构，并也为这个结构能够定制存放到共享内存设计好了方针策略。但下面的工作，将更富于挑战性：

- 对于 push 的操作，需要操作前等待队列已经有了空间，也就是说队列没有满的状态。等到这个状态出现了，才继续进行 push 的操作，否则，push 操作挂起。
- 对于 get 的操作，需要操作前等待队列有了数据，也就是说队列不为空的状态。等到这个状态出现了，才继续进行 get 的操作，否则，get 操作挂起。
- 上面的 push 操作/get 操作一般是不同的进程和线程。同时，也有可能有多个 push 的操作和 get 操作的进程和线程。

这些工作我们将在随后的设计中进行。且听下回分解。

附件：LoopQue 的测试程序：

```
/* tst-loopque.cpp
 test program for  class
 Author: ZhangTao
 Date: July 27, 2009
*/

# include
# include            // for  function
# include      "xtl/LoopQue.h"

int
main(int argc, char **argv)
{
 int  qsize = 0;

 if ( argc > 1 )
   qsize = atol(argv[1]);

 if ( qsize < 1 )
   qsize = 5;

 xtl::LoopQue<int>  queue(qsize);
 for ( int i = 0; i < (qsize - 1); i++ ) {
   queue.push(i);
   std::cout << "Loop push:" << i << "\n";
 }

 queue.check_push(1000);
 std::cout << "Full:" << queue.full() << " Size:"
      << queue.size() << "\n\n";

 for ( int i = 0; i < qsize; i++ ) {
  int val = queue.front();
  std::cout << "Loop pop:" << val << "\n";
  queue.pop();
 }

 std::cout << "\nEmpty:" << queue.empty() << " Size:"
      << queue.size() << "\n";
 return 0;

}
```