泛型编程与设计模式*)

陈叶旺　余金山
(华侨大学信息与工程学院　泉州 362011)

摘　要　现今面向对象已经成为软件业内的主流技术，然而它存在很多的弱点，使得它的通用性和抽象程度受到很大限制。设计模式作为面向对象领域内的高级软件复用技术，同样摆脱不了相同的困扰。泛型编程是面向对象的进一步发展，从更高的角度对世界进行抽象，为面向对象的不足之处提供了解决之道。将设计模式泛化带来更大规模的代码复用，使得设计模式的实现得以自动化，从而使开发人员能以面向设计的方式让使用者自行装配所需机能，产生能表达原始设计意图的代码，实现设计与编码之间的无缝过渡。

关键词　面向对象，设计模式，泛型编程，正交分解，Policy-Classes

Generic Programming and Design Patterns

CHENG Ye-Wang　YU Jin-Shan
(College of Info, Sci. & Eng., Huaqiao Univ., Quangzhou 362011)

Abstract　Now object-oriented has been the main technologies of software industry, However, there are many problems which limit its general purpose and the abstract degree. Design pattern is an advanced technologies of software reusing within object oriented, but it still can not break away the bugging of the similar problems. Generic programming develops object oriented. It abstracts the world with a higher view, thus many problems in object oriented can be handled by it. Using the ideal of generic programming to generalize design patterns brings programmers mass reusing and automated design patterns, gifting programmers the ability to assemble the components by requirements, and result in the source codes that can express the primitive purpose of design, archiving the natural transition from designing to coding.

Keywords　Object oriented, Design patterns, Generic programming, Orthotropic decomposition, Policy-classes

1　引言

面向对象(OO)是人们在软件工业里认识、模拟现实世界的一种方法，它模拟人类习惯的思维方式，使软件开发符合自然规律，适应了软件规模和数量以惊人的速度急剧膨胀，提高了软件系统的稳定性、可修改性和重用性。然而，随着OOP广泛应用，它的缺点也不断暴露出来，设计模式的应用并不能从根本上提供解决之道。

泛型编程(GP)进一步发展了面向对象，它从更高的角度来理解世界，具有更好的抽象性。它是继面向对象程序设计之后对程序设计领域的又一次大的冲击，已经影响到软件开发的思维模式。随着GP的流行，OO的主导地位有不断被淡化的趋势。

1.1　面向对象的不足

自从上世纪中后期至今，面向对象已经成为软件设计与开发的主流技术。它有很多的优点，然而也有自身难以克服的内在弱点：

1)OO核心原则：OO的核心原则——将抽象接口定义与具体实现分离，以抽象基类(或接口)作界面来构造系统的框架结构，以继承和多态为手段来扩展系统。图1是一个典型2D图形软件的架构。

这个简单的OO系统中，只以一个纯虚类CShape与外部交互，这样的设计有很多优点，但是它要求CShape这个接口要稳定，设计好后，不能再有大的变动。但在软件工业里，需求总是在不断地变化，如果要将图形对象持久化，即要将图形对象写入到文件或数据库中，那就要在基类中增加虚方法(如serialize)，然后在每个子类中重写它，然后重新编译。如果要改变内部的继承结构，也要重编译。假设身处在一个开发团队中，团队使用一个比较成熟的程序库，因个人需要，要求增加功能而使得程序库需要修改或重编译，这样的大成本动作一般不会得到允许。

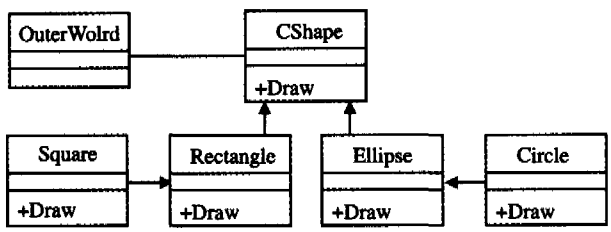

图1　一个典型的OO设计

2)面向对象技术本身十分难以把握，有太多的原则，在≪Object-Oriented Design Heuristics≫[1]中就总结出61条，设计者必须十分地小心。常常为了满足某些原则，却又要违反另一些原则，一不小心就会造成严重的只在未来才显现的后果，因此设计总是在其中艰难地寻找平衡点。

3)抽象方法与描述机制的问题：OO用类来描述对象，将

*)基金项目：福建省自然科学基金；项目编号A0210018。陈叶旺　硕士，研究方向为软件工程；余金山　教授，主要研究领域为软件工程、网络计算和人工智能应用。

对象限定在一个结构内，强调数据与算法的捆绑。用继承机制将子类与父类相耦合，形成概念间的层次关系。这使得两者耦合性大增，妨碍了组件的独立性、弹性、交互性，即造成所谓的代码交织(Code Scattering and Code Tangling)的问题，这是AOP(面向方面)出现的主要原因之一。另外，OO将一切都视为对象，然而有些类型无法对象化，像C++中的Int、double这些基本类型就不是对象，不能用类来描述。

4)复杂的系统中大量地运用虚方法，导致成本的增加。目前的编程语言，大多以虚方法表(vTable)方式来实现多态，要为每个对象增加一个指向vTable的指针，vTable再指向实际函数，这使得时空开销增大。特别是小型对象，浪费更严重，一些比较小的对象本身只有1～4个字节，增加一个指针就要4个字节，这样的开销对小型对象就太大了。

5)很多OO系统的设计过多依赖继承。其实，滥用继承相当危险，继承缺点如有：(1)破坏了封装性，因为这会将一部分父类的实现细节暴露给子类；(2)父子类之间耦合性强，当父类的实现更改时，子类也不得不会随之更改；(3)从父类继承来的实现将不能在运行期间进行改变。

当需求有变动时，我们希望能尽量减小现有代码的更改、希望类的数量按线性增长，但是随着类的层次的增加，类的增长却变得难以控制，这个速度常常是几何级的。

所有这些造成面向对象的抽象性、通用性、可重用性以及运行效率都受到了很大的限制。

1.2 设计模式

设计模式[2]是面向对象领域为实现高级代码复用而总结出来的各种方案，每一种模式都是对在所处环境中不断重复发生的问题提供的解决之道，其最根本的意图就是适应需求变化。正确使用模式，将提升软件的健壮性、可复用性、可扩展性、可维护性。然而，模式无法摆脱OO固有的顽疾，其主要缺点如下：

(1)各种不同的模式之间存在着很多功能上的交织，比如Abstract Factory模式与Factory Method模式就存在很大的相似性：前者是用于创建一组相互关联的对象，而后者用于创建单个对象。从功能上看，两者并无多大区别，相同的功能部分完全可以重用，但模式自身难以做到。

(2)模式并非易拉罐，不是随手拿来就可使用，其最根本意图是为适应变化。只有当它适合的环境出现时才应该被使用，但是要作出这种判断决非易事。为了模式而使用模式是本末倒置，然而这种情形经常发生，主要是开发人员见山只是山、见水只是水，看不清模式的背后不过是OO的简单原则，只为单纯追求模式的应用，这却恰恰背离了模式的初衷。

(3)设计模式是指导性的设计方案，具体的应用要做出必要的调整，且模式的应用要求增加更多类，要进行更多的设计工作，要求设计者熟练撑握复杂的面向对象技术。

所有这些限制了设计模式的通用性、可重用性及实际应用，也很难被一般的开发人员所掌握，造成设计与实现之间存在一定的脱钩，后果常常是模式应用的失败。

自从1995年GOF的设计模式提出之后，模式在软件开发中大量应用，新的设计模式也不断涌现，模式社群中有许多人试图用各种方法改善设计模式[3,4]。国内这方面的研究也比较广泛，如武汉大学的软件工程实验室提出的角色模型化[5]、模式形式化[6]，但这些都没有质的飞跃，不能根本解决问题。而且形式化和角色化方法的应用更加困难，对于开发人员的要求更高。以AOP(面向方面)的方式改进OO及设计模式近年来比较热门，但是AOP要有独立的语言支持，如AspectJ，目前主流的OOP语言做不到，因而这些改进收效不大。

要克服OO与设计模式的这些困境，需要OO以外的更抽象，更高效的新方法。

2 泛型编程

泛型编程(Generic Programming，GP)，起源于1968年Douglas MciLory在“Mass Produced Software Components”提出的可复用软件组件思想，直到上世纪90年代中后期。STL (Stand Template Library)异军突起，它是一个通用容器和算法库，因其极高的抽象程度、通用性和运行效率，1994年成为ISO C++标准语言库的重要组成部分。STL是GP思想发展的一个里程碑，是现今最成功的GP范例。在GP思想的指导下，克服了OO的抽象程度不高、通用性不足的缺点。引入STL后，C++的编程风格有极大的改观，同时促使其它语言完善基础设施以支持GP，1998在德召开了以GP为议题的首届国际研讨会，从此在产业界和学术界受到广泛关注。

泛型编程与OO从不同的角度看待问题，它的基本观点是把抽象的软件组件和它们的行为用标准的分类学分类，出发点就是要建造真实的、高效的和不取决于语言的算法和数据结构。即GP强调数据与算法的分离，算法不依赖于具体的数据，数据完全不知算法，但提供必要的操作上的支持，这得益于Iterators(迭代器)。简单表示为图2。

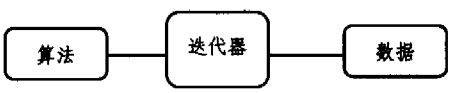

图2 GP中数据与算法的关系

算法与数据中间被迭代器隔离开，算法通过迭代器操作数据，数据则通过迭代器为算法提供必要的操作上的支持，迭代器成功地剪断了两者间的耦合性。

GP的最基本概念：

1)Concept(概念)。Concept是一组能支持相应的操作类型数据的定义，这些操作与其具体的数据类型无关，也可以认为它是一组类型的集合。任何属于某一Concept的类型，称之为这个Concept的Model。例如一个简单的Concept---Assignable，其定义为：类型T对象值可以被复制并可被赋值，也就是说这个类型对象要提供‘=’操作。C++中的基本类型，如int、bool，都是这个Concept的model。用户自定义的类如果能提供‘=’操作，那么这个类也是Assignable的Model。

2)Container(容器)。是一种对象，可以包含并管理其它的对象。它提供迭代器，用以对其所包含之元素进行操作。

3)Iterators(迭代器)。是STL的核心，它们是泛型指针，是一种指向其他对象的对象。迭代器能够遍历由对象所形成的区间，让我们得以将容器与作用其上的算法分离。大多数的算法自身并不直接操作于容器上，而是操作于迭代器所形成的区间中。

简单的实例：

```
List(int)L ; //L 是个链表 ,L 存放 int 类型数据
L. push_back (3);//往 L 中加一个数 3
L. push_back (1);
List(int)::iterator result = find
```

(L. begin(),L. end(),3);//查找 3

本例 List 是个 Container ,可以存放指定类型的数据(本例是 int 型);find 是一个泛型算法,它在 L 中查找是否有 3;[L. begin(),L. end())则形成一个指向 Int 型数据的区间,find 算法在这个区间内进行查找,结果返回指向'3'的 iterator,若失败则返回 L. end()。它不关心参与的数据具体类型,本例也可以是 float、double 或你自己定义的类,只要这个类型能进行等值比较,List〈数据类型〉::iterator 是成功分离 find 算法与数据类型的关键。

3 GP 与 OO 的关系

GP 与 OO 都有自己的思想体系,有自己的方法论,是独立的世界观。其不同点见表 1。

表 1 GP 与 OO 比较

	GP	OO
抽象方法	依据类型所能满足的操作需求分类	按对象所承担的责任,封装数据与算法,以形成类
层次结构	每个 Concept 层次可以包含无穷互不相关的类型,可以独立更改与扩充	基类与子类间组成紧耦合的层次结构关系,变动基类或继承结构是巨大的工程
数据与算法关系	严格分离,互不相知	紧密耦合
设计重点	语义与语义分类(将类型按操作需求分类)	定义类及架构层次关系

GP 是 OO 的自然延伸。OO 精髓在于抽象概念和具体的实现方式的分离,极力强调类型的一致性;GP 则注重于对类型按操作需求进行语义分类,分类的结果是得到不同的概念(Concept)。Concept 不是类,一个 Concept 可以包含无数的类与非类类型数据(如 Int、double 等 C++ 中的原始数据类型),一个类可以同时分属于不同 Concept,所以说 Concept 是比类更具有一般性的观念。从这个角度看 GP,它是针对抽象的 Concept 编程,以共同的需求条件为共性抽象。OO 与 GP 本质上都是对世界的抽象,针对抽象编程,这是二者统一点,但 GP 走得更远,更加抽象,OO 毕竟还要强调类型一致性,GP 将其剥离得只剩 Concept 。

GP 与 OO 两者可以相互结合,以 GP 的思想将 OO 中静止的那部分抽出来,最大限度交融 GP 和 OO。前者可以改善后者,特别是可以降低 OO 中对象组织、互操作的复杂性;而后者是前者的基础,GP 的运作依赖于 OO。

4 GP 的不足

虽然 GP 相对于 OO 有极大的优越性,但实际运用中,OO 依然有很大的优势:OO 采用的类表述容易,继承机制语义清晰,便于实现,OO 能得到语言级别上的支持,GP 则不然。直到目前,还没有一个主流的程序设计语言能直接支持 GP,无法定义 Concept,这使得 Concept 的观念主要通过人工的约定来遵循,并要求开发人员严格服从这种方式,这是其很不理想的环节。GP 大量运用 Template,使得编译时间加长。

5 国内外 GP 的发展动态

自从 GP 出现,许多学者致力于它的一般化理论研究。IFIP(International Federation For Information Processing) WG2.1 从事有关 GP 论文的收集工作,并于 2002 年 5 月出版相关文集。STL 之父 Alexander Stepanov 正从事研究出一种 GP 语言。目前的主流是以 C++ 模板方式实现泛型,Java JDK1.4 之后开始支持 GP。微软的. Net Framework2.0 也将在 2005 年全面支持 GP,正在计划推出的 Generic Java 或 C# with generics 所采用的泛型模型都不是真正的"基于需求"意义上的泛型编程。David R. Musser 是 GP 设计方面的权威,从事 GP 设计的理论研究。国内近几年引进了不少 STL 与涉及 GP 方面的图书,GP 也开始在国内受到广泛关注,北京大学语言研究所孙斌等正从事 GP 语言研究工作[7]。

6 设计模式的泛化

2001 年,Andrei Alexandrescu 首次提出运用 GP 思想将设计模式泛化[8],以便少量的代码就可以实现组合性的具有无穷数量的结构和行为,又可广泛应用于各种程序库中。Andrei 与 GOF 之一 John Vlissides 在 C++ Report 不断推出最新研究成果。国内还很少有人从事这方面的工作。

GOF≪设计模式≫中所描述的 23 种模式,应用范围广、可复用程度高,但这些设计模式到了实际的应用中要依情况而相应做调整。不同的模式存在各自不足之处,不同应用领域也存在许多其它的设计模式,这使得其应用就有很大的难度。然而,运用新的泛型编程技术,设计出一组可复用的泛型组件,用于生成规范的设计模式代码。即将设计模式的实现自动化,让软件开发人员能够以"Design-Oriented"(面向设计)方式自行装配所需机能,产生能表达原始设计意图的代码,实现软件工业中设计与编码之间的无缝过渡。

软件工业中设计就是制定约束与规范。泛型组件使之得到简化,为用户提供所有的设计可能 ,让设计成为在软件解域空间中做选择题,指定类型与行为,规化出合理的设计。即按需求选择不同的泛型组件,以实现设计的约束,从而得到简洁的易于表达且易于维护的代码,最大限度弥合设计与实现之间的差距。这样,在得到极大规模的代码复用的同时,也能得到了巨大的弹性。

当前,C++ 语言理想地达成了抽象和高效的统一,提供 GP 所需的一切深度与广度:(1)C 内存模型保证原始效率,(2)面向对象是 GP 运作基石,(3)模板技术成就了编译期的静态多态,从而提供代码自动生成能力。这些使得 GP 思想在 C++ 中实现得非常出色。

因此本文以下所举例子将用 C++ 语言,以正交分解法将特定设计模式所要完成的功能分解为多个互不相关的最小的、可复用的、只负责完成某一方面行为的主题——称之为 Policy 。每个 policy 针对选定主题提供标准接口,可以为一个 Policy 选择任何适当的实现方式,特定的 Policy 的具体实现类称之为 Policy-Class,借由不同的 Policy-Classes 组合成高端功能集合——模式组件,使用者却无需关心 Policy-Class 的具体实现方式。

具体的实例:提供线程约束的能创建对象模式组件。

步骤 1 从需求可以分析与分解出两个主题:(1)线程约束;(2)创建对象。为此,可引入两个相应的 Policy:Thread-Modle Polcicy 和 Creator Policy。

步骤 2 定义两个 Policy。Policy 只是抽象定义,与语言无关,也不涉及具体实现:

(1)Creator Policy 定义:使用者需提供对象类型 T,它的实作类需提供一个 Create()方法供外界使用,传回一个 T 类

型对象指针。

(2)ThreadModle Policy 定义:它的实作 Classes 提供一个 Check()方法,接受一个类型为 T 的对象作为参数。

步骤 3 实作出两个 Creator Policy 类,用以生成 T 类型对象(若有需要,还可实作出更多的不同的类),如表 2。

表 2 两个 Creator Policy Classe

用 new 方法生成对象	用 Malloc 生成对象

Template<class T>
Struct OpNewCreator{
Static T * Create(){
Return new T;
}
}
	Template <class T>
	Struct MallocCreator{
	Static T * Create(){
	Void * buf = std::malloc(sizeof(T))
	If(! buf)return 0;
	Return new (buf)T; }}

步骤 4 实作出 2 个 ThreadModel Policy 类,用以检查 T 类型对象,如表 3。

表 3 两个 ThreadModle Policy classes

单线程	Template<class T>SingleThread{
	Static bool Check(T){
	If(…)
	Return true
	Else
	Return false;}};

多线程	Template<class T> MultiThread{
	Static bool Check(T ptr){
	if (Lock(ptr)); //锁定
	{……
	Return true;}
	Return false };

步骤 5 用 Policy classes 以组合或继承方式实现特定功能集合:

```
Template
< class T,//提供的类型参数
  template<class> class CreationPolicy,
      //对象创建的泛型组件参数
  template<class> class ThreadModle
      //线程约束的泛型组件参数
>
class ObjCreator:public CreatorPolicy<T>,
    public ThreadModle<T>{};
//本例以继承方式实现,也可选择组合方式
```

步骤 6 组件的使用

本例中有两个 Policy 主题,两主题又各有两个实作类,按排列组合可产生出 4 种使用方式:

(1)以单线程方式用 new 方法产生 Printer 对象:

```
Typedef ObjCreator
<Printer,OpNewCreator,SingleModle>NSObjCreator
If (NSObjCreator. Check())
  Printer * aPr = NSObjCreator. Create();
```

(2)以单线程方式用 malloc 方法产生 Printer 对象:

```
Typedef ObjCreator
<Printer,MallocCreator,SingleModle>MSObjCreator
If (MSObjCreator. Check())
  Printer * aPr = MSObjCreator. Create()
```

(3)以多线程方式用 new 方法产生 Printer 对象:

```
Typedef ObjCreator
<Printer,OpNewCreator,MultiModle> NMObjCreator
If (NMObjCreator. Check())
  Printer * aPr = NMObjCreator. Create();
```

(4)以多线程方式用 malloc 方法产生 Printer 对象:

```
Typedef ObjCreator
<Printer,MallocCreator,MultiModle> MMObjCreator
If (MMObjCreator. Check())
  Printer * aPr = MMObjCreator. Create();
```

通过此例可以清楚看到,使用者可以根据实际需求自由组装,实现了设计约束所要求达到的机能——称之为 Policy Based Designing。这种方法带来非常大的弹性,从整体上大幅度提高了模式的可重用度、抽象程度、可扩展性、灵活性及编程效率。

(1)灵活性。本例中用户可以从外部改变对象的生成方式,根据需求提供自己的 Polices class,用以适应采用不同内存模式的系统。这实现了通过'设计选择'来满足需求,即'面向设计'的理念,设计与实现了自然链接,也提高了设计的可扩展性。

表 4 OO 继承方式与泛型组件比较

	纯 OO 继承方式	泛型组件方式

增加的新类	按排列组合方式需要定义的 4 个类:
	(1)Template <class T> class MSObjCreator
	:public MallocCreator<T>,
	public SingleThread<T>{…}
	(2)Template <class T>class MMObjCreator
	:public MallocCreator<T>,
	public MultiThread<T>{…}
	(3)Template <class T>class NMObjCreator
	:public OPNewCreator<T>,
	public MultiThread<T>{…}
	(4)Template <class T>class NMObjCreator
	:public OPNewCreator <T>,
	public MultiThread<T>{…}
		只定义一个 ObjCreator:
		Template
		<class T,
		template<class> class CreationPolicy,
		template<class> class ThreadModle >
		class ObjCreator:public CreatorPolicy<T>,
		public ThreadModle<T>
		{
		……
		}

效果	随着模板参数的增加,类数量将以指数的增长	只增加模板参数,由编译器根据使用者提供的模板参数生成相应的类

(2)设计简单。泛化的结果是产生一个个经过良好设计与严格测试的最小的泛型组件,每个组件只完成一个主题任务,使其能很好地解决不同设计模式之间功能交织的问题,重用度有了很大程度的提升。现在设计过程只是在组件库中进行选择,设计变简单了。

(3)执行成本。泛化的组件一般很少使用虚函数,很大程度地减少运行期的成本。因为编译器承担了很多工作,它把很多开销转移到了编译期。

(4)编译期的错误检查。C++的 Template 的代码生成功能帮助开发人员生成规范代码,减少出错几率,更得到编译期的错误检测。本例中,若 OpNewCreateor 没有提供 Create() 方法,在编译期就会得到出错信息,减少了运行期出错所造成的巨大代价。

(5)编程效率。相对于用纯 OO 的继承手工方式一个个实现类的定义,此方法用少量的代码就可以实现组合性的具有无穷数量的结构和行为。表 4 是纯 OO 继承方式与泛型组件方式的简单比较。

结束语 泛型编程是面向对象的进一步补充与扩展,近年来越来越受人们的关注,国内对 GP 的研究和应用也越来越广。本文分析了面向对象、设计模式不足之处;针对这些不足引入 GP 的核心思想,论述了 GP 的优势;之后讨论设计模式的泛化问题,泛化的结果是带来了更大规模的复用、更好的通用性、更高抽象程度。同时,本文讨论了 GP 的发展动态与不足。GP 本身也处在一个不断发展的阶段,尚有极大潜力有待挖掘。

参 考 文 献

1 Arthur J. Riel Object-Oriented Design Heuristics [M]. Addision Welly ,1996

2 Gamma E, Helm R, Johnson R, et al. Design Patterns: Elements of Reusable Object-Oriented Software [M]. Addison-Wesley, 1995

3 McNatt W B, Bieman J M. Coupling of Design Patterns: Common Practices and Their Benefits [J]. Computer Software & Applications Conf, 2001

4 Mak J K H, Choy C S T, Lun D P K. Hierarchical relationships among bad design patterns and good design patterns [J]. In: Proceedings of the IASTED International Conference on Computer Science and Technology, 2003. 7~13

5 徐永松,何克清,卓识,等. 一个更容易应用软件模式的方法[J]. 计算机工程,2003,29(9):83

6 方海棠,何克清,卓识,等. 一个基于模式和动作语义的 MDA 实现方法[J]. 计算机工程,2004,30(4):67

7 孙斌. 面向对象,泛型程序设计与类型约束检查[C]. 计算机学报,2004,27(11):1492

8 Alexandrescu A. Modern C++Design —Generic Programming and Design Patterns Applied [M]. Addison-Wesley, 2001

(上接第 229 页)

3.2 离散余弦变换

离散余弦变换实际是傅立叶变换的实数部分:

$$F(0,0)=\frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)$$

$$F(\mu,0)=\frac{\sqrt{2}}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)\cos\frac{(2x+1)\mu\pi}{2N}$$

$$F(0,v)=\frac{\sqrt{2}}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)\cos\frac{(2y+1)\mu\pi}{2N}$$

$$F(\mu,v)=\frac{\sqrt{2}}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)\cos\frac{(2x+1)\mu\pi}{2N}\cos\frac{(2y+1)\mu\pi}{2N}$$

其中,$f(x,y)$为空间域中的二维向量,$x,y=0,1,2\cdots,N-1$,$F(\mu,v)$为变换系数矩阵,$\mu,v=0,1,2,\cdots,N-1$。

把 YCbCr 图像按 8*8 分块,然后按往返对角把这 64 个原始数据排序,对其进行离散余弦变换输出为 64 个空间频率振幅值,其低频分量都集中在左上角,高频分量分布在右下角(DCT 变换实际上是空间域的低通滤波器)。由于该低频分量包含了图像的主要能量,而高频与之相比,就不那么重要了,因此我们可以忽略部分高频分量,从而达到压缩的目的。量化操作就是将某一个值除以量化表中对应的值。由于量化表左上角的值较小,右上角的值较大,这样就起到了保持低频分量,抑制高频分量的目的。JPEG 使用的颜色是 YCbCr 格式。我们提到过,Y 分量代表了亮度信息,CbCr 分量代表了色差信息。人眼对图片上的亮度 Y 的变化远比色度 C 的变化敏感。我们可以对 Y 采用细量化,对 CbCr 采用粗量化,可进一步提高压缩比。量化操作相当于减低图像的分辨率。

3.3 熵编码(Huffman 编码)

基本原理是频繁使用的数据用较短的代码代替,较少使用的数据用较长的代码代替,每个数据的代码各不相同。这些代码都是二进制码,且码的长度是可变的。

产生 Huffman 编码需要对原始数据扫描两遍。第一遍扫描要精确地统计出原始数据中每个值出现的频率,第二遍是建立 Huffman 树并进行编码。

3.4 Jpg 的文件格式

Jpg 文件格式由标记码(Tag)和数据两部分组成。标记码由两个字节组成,高字节为 0XFF。标记码把 Jpg 文件进行分段,在标记码之间填充相应的数据。

1 图像开始标记 SOI 0xFFD8

2APP0 标记

3APPn 标记

4 量化表 DQT 0xFFDB

5 帧图像开始 SOF

6 哈夫曼表 DHT 0xFFC4

7 扫描开始标记 SOS 0xFFDA

8 图像结束标记 EOI 0xFFD9

得到了 Jpg 的数据块后,按文件格式填入相应位置,存储在 ROM 图像资源路径中。

结束语 本文讨论基于 S3C2410 和 ARMLinux 操作系统下 Web 服务器中视频图片压缩及其传输的实现方法,系统使用 Jpg 编码技术,图像转换回 BMP 图像后可以作为计算机图像处理、模式识别的原始图像,达到远程现场监控等目的。本论文给出的方法避免了对 MPEG4 的复杂编解码操作,节省了大量系统开销和处理时间,在一些要求不高,需要简单视频监控的情况下,经济适用,操作简单,安装方便,便于工厂环境大规模使用,配合了工业以太网总线下的视频图像压缩方法。

参 考 文 献

1 邹思轶. 嵌入式 Linux 设计与应用. 清华大学出版社,2002

2 张天予,王运坚. VC++数字图像处理. 人民邮电出版社,2003

3 张宏林,王健. 基于 Intel PXA255 平台的网络摄像机设计. 单片机及嵌入式系统应用,2005

4 何斌,马洋. VC++数字图像模式识别技术及工程实践. 人民邮电出版社,2003

5 杨枝灵,王开. VC++数字图像获取处理及实践应用. 人民邮电出版社,2003

6 谷口庆次[日]. 数字图像处理. 科学出版社,2003