



代码阅读训练

CODE Reading

LONGCHEER CONFIDENTIAL

目录

- 1 认识代码阅读.....
- 2 代码阅读的工具.....
- 3 代码阅读的方法与技巧.....
- 4 大型项目代码阅读.....
- 5 课程总结 分享感言.....

目录



认识代码阅读.....



代码阅读的工具.....



代码阅读的方法与技巧.....



大型项目代码阅读.....



课程总结 分享感言.....

1

认识代码阅读

--入门与提高的必由之路

1.1

▶ 为什么阅读代码

1.2

▶ 如何阅读代码

1.3

▶ 本培训的范围

为什么阅读代码

- 1 信息时代，我们整个社会的知识更新速度越来越快，而位于信息时代风口浪尖上的计算机技术日新月异。作为程序员，总是要不断地学习新的知识。
- 2 计算机科学是一门实践性很强的科学，许多内容往往在书本上根本学不到。代码中往往凝聚着许多实践性的知识，通过阅读代码才能真正掌握软件开发的真谛。
- 3 代码阅读不是一件容易的工作，但却是一件不得不做的工作，无论是工作的移交、新手的人门或是加入新的项目，都要阅读大量由他人编写的代码。
- 4 学习编写伟大代码的方式是阅读代码，阅读大量的代码。否则，我们就会不断地重做别人已经完成的工作，重复过去已经发生过的成功和错误。

1

认识代码阅读

--入门与提高的必由之路

1.1

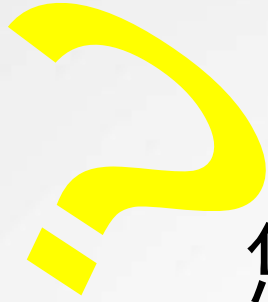
▶ 为什么阅读代码

1.2

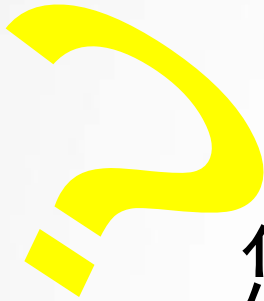
▶ 如何阅读代码

1.3

▶ 本培训的范围



您是会阅读中文吗？



您是如何阅读中文的？

1、指精读。就是说在每句阅读时，先理解每字的意思，然后通解一句之意，又通解一章之意，相接连作去，明理演文，一举两得”这是传统的三步精读法。它是培养学生阅读能力最主要最基本的手段。……

2、速读。在现代社会当中，对信息的筛选能力和筛选速度尤其重要。如果每篇文章都字斟句酌，则很难适应时代的要求，跟上时代的步伐。作为教师的我们应指导学生根据自身所需选择读物进行速读，当然在速读的同时也不能忽略对内容的理解，这样学生们就能在最少的时间获取尽量多的信息。

3、写读书笔记。文章中富有教育意义的警句格言、精彩生动的词句、段落，可以摘录下来，积存进自己设立的“词库”中，为以后的作文准备了丰富的语言积累。目前许多学生将读书笔记作为一项硬性任务，我想我们可以将读书笔记做得更鲜活一些，比如做成贺卡、书签等，这样阅读就会变得更精彩，更有实效。

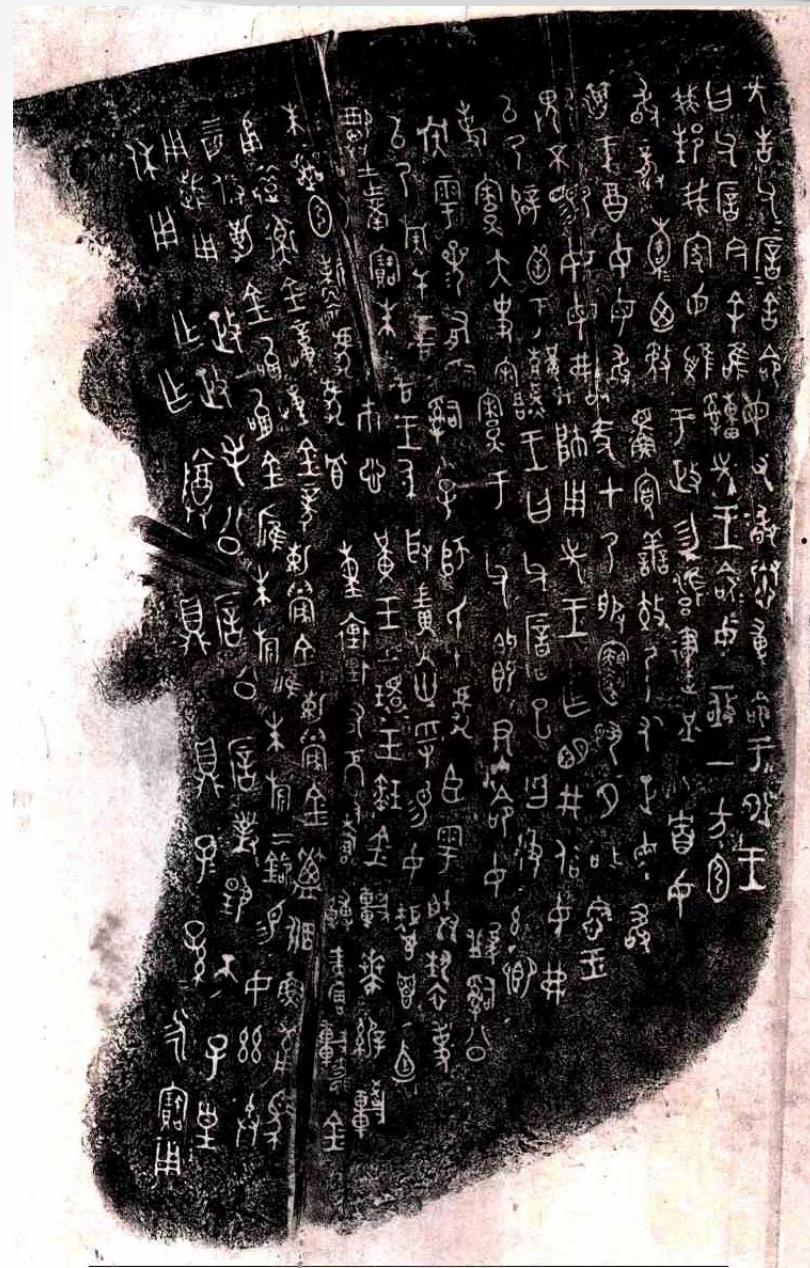
-- 《小学语文阅读方法》

西周散文阅读



毛公鼎为西周晚期的宣王时期器物。清道光末年于陕西岐山出土的重器。为皇皇钜制，被誉为“抵得一篇尚书”。

毛公鼎为西周晚期宣王时期器物。清道光末年于陕西岐山出土。通高53.8公分，腹深27.8公分，口径47.9公分，腹围145公分。重量为34.54公斤。器形作大口，半球状深腹，兽蹄形足，口沿上树立形制高大的双耳，浑厚而凝重，整个器表装饰整洁，素朴典雅。



readme.txt

- 毛公鼎为西周晚期的宣王时期器物。清道光末年于陕西岐山出土的重器。为皇皇钜制，被誉为“抵得一篇尚书”。由作器人毛公得名。直耳，半球腹，矮短的兽蹄形足，口沿饰环带状的重环纹。
- 铭文32行499字，乃现存最长的铭文：完整的册命。
- 其内容是周王为中兴周室，革除积弊，策命重臣毛公，要他忠心辅佐周王，以免遭丧国之祸，并赐给他大量物品，毛公为感谢周王，特铸鼎记其事。
- 其书法是成熟的西周金文风格，奇逸飞动，气象浑穆笔意圆劲茂隽，结体方长，较散氏盘稍端整。李瑞清题跋鼎时说：“毛公鼎为周庙堂文字，其文则尚书也，学书不学毛公鼎，犹儒生不读尚书也。”

铭文解读

第一段

王若曰：“父歆，丕显文武，皇天引戾厥德，配我有周，膺受大命，率怀不廷方亡不覲于文武耿光。唯天将集厥命，亦唯先王略又厥辟，属谨大命，肆皇天亡，临保我有周，丕巩先王配命，畏天疾威，司余小子弗，邦将曷吉？迹迹四方，大从丕静。呜呼！惧作小子溷湛于艰，永巩先王”。

这是毛公鼎铭文的第一段文字。该鼎铭全文分五段，每段均从“王若曰”起，文辞完整而精妙。其一，此时局势不宁；其二，宣王命毛公治理邦家内外；其三，给毛公予宣示王命之专权，着重申明未经毛公同意之命令，毛公可预示臣工不予奉行；其四，告诫勉励之词；其五，赏赐与对扬。是研究西周晚年政治史的重要史料。



我们一起总结一下阅读的技巧

每组总结三条：

- 1 _____
- 2 _____
- 3 _____

三步精读法

在每句阅读时，先理解每字的意思，然后通解一句之意，又通解一章之意，相接连作去，明理演文，一举两得”这是传统的三步精读法。

-- 《小学语文阅读方法》

1

认识代码阅读

--入门与提高的必由之路

1.1

▶ 为什么阅读代码

1.2

▶ 如何阅读代码

1.3

▶ 本培训的范围

课程目标

- 建立正确的代码阅读观念与态度
- 学习代码阅读的技巧
- 识别自己在代码阅读上的问题
- 学会代码阅读，提高工作效率
- 提升软件调试的效率

本培训的范围

- 我们探讨
 - 阅读和理解他人代码的方法技巧
- 我们探讨
 - 阅读和理解他人代码的方法技巧

小窍门

- 1.要养成一个习惯,经常花时间阅读别人编写的高品质代码.
- 2.要有选择地阅读代码,同时,还要有自己的目标.您是想学习新的模式|编码风格|还是满足某些需求的方法.
- 3.要注意并重视代码中特殊的非功能性需求,这些需求也许会导致特殊的实现风格.
- 4.在现有的代码上工作时,请与作者和维护人员进行必要的协调,以避免重复劳动或产生厌恶情绪.
- 5.请将从开放源码软件中得到的益处看作是一项贷款,尽可能地寻找各种方式来回报开放源码社团.
- 6.多数情况下,如果您想要了解"别人会如何完成这个功能呢?",除了阅读代码以外,没有更好的方法.
- 7.在寻找bug时,请从问题的表现形式到问题的根源来分析代码.不要沿着不相关的路径(误入歧途).
- 8.我们要充分利用调试器|编译器给出的警告或输出的符号代码|系统调用跟踪器|数据库结构化查询语言的日志机制|包转储工具和Windows的消息侦查程序,定出的bug的位置.

小窍门

- 9.对于那些大型且组织良好的系统,您只需要最低限度地了解它的全部功能,就能够对它做出修改.
- 10.当向系统中增加新功能时,首先的任务就是找到实现类似特性的代码,将它作为待实现功能的模板.
- 11.从特性的功能描述到代码的实现,可以按照字符串消息,或使用关键词来搜索代码.
- 12.在移植代码或修改接口时,您可以通过编译器直接定位出问题涉及的范围,从而减少代码阅读的工作量.
- 13.进行重构时,您从一个能够正常工作的系统开始做起,希望确保结束时系统能够正常工作.一套恰当的测试用例(test case)可以帮助您满足此项约束.
- 14.阅读代码寻找重构机会时,先从系统的构架开始,然后逐步细化,能够获得最大的效益.
- 15.代码的可重用性是一个很诱人,但难以理解与分离,可以试着寻找粒度更大一些的包,甚至其他代码.
- 16.在复查软件系统时,要注意,系统是由很多部分组成的,不仅仅只是执行语句.还要注意分析以下内容:文件和目录结构|生成和配置过程|用户界面和系统的文档.
- 18.可以将软件复查作为一个学习|讲授|援之以手和接受帮助的机会.

第一章就到这里，稍后进入下一章

1

认识代码阅读

2

代码阅读的工具

3

代码阅读的方法与技巧

4

大型项目代码阅读

5

课程总结 分享感言

2

代码阅读的工具

2.1

▶ source insight的使用

2.2

▶ source navigator的使用

2.3

▶ beyond compare简介

2.4

▶ visual assist X简介

2

代码阅读的工具

2.1



source insight的使用

2.2



source navigator的使用

2.3



beyond compare简介

2.4

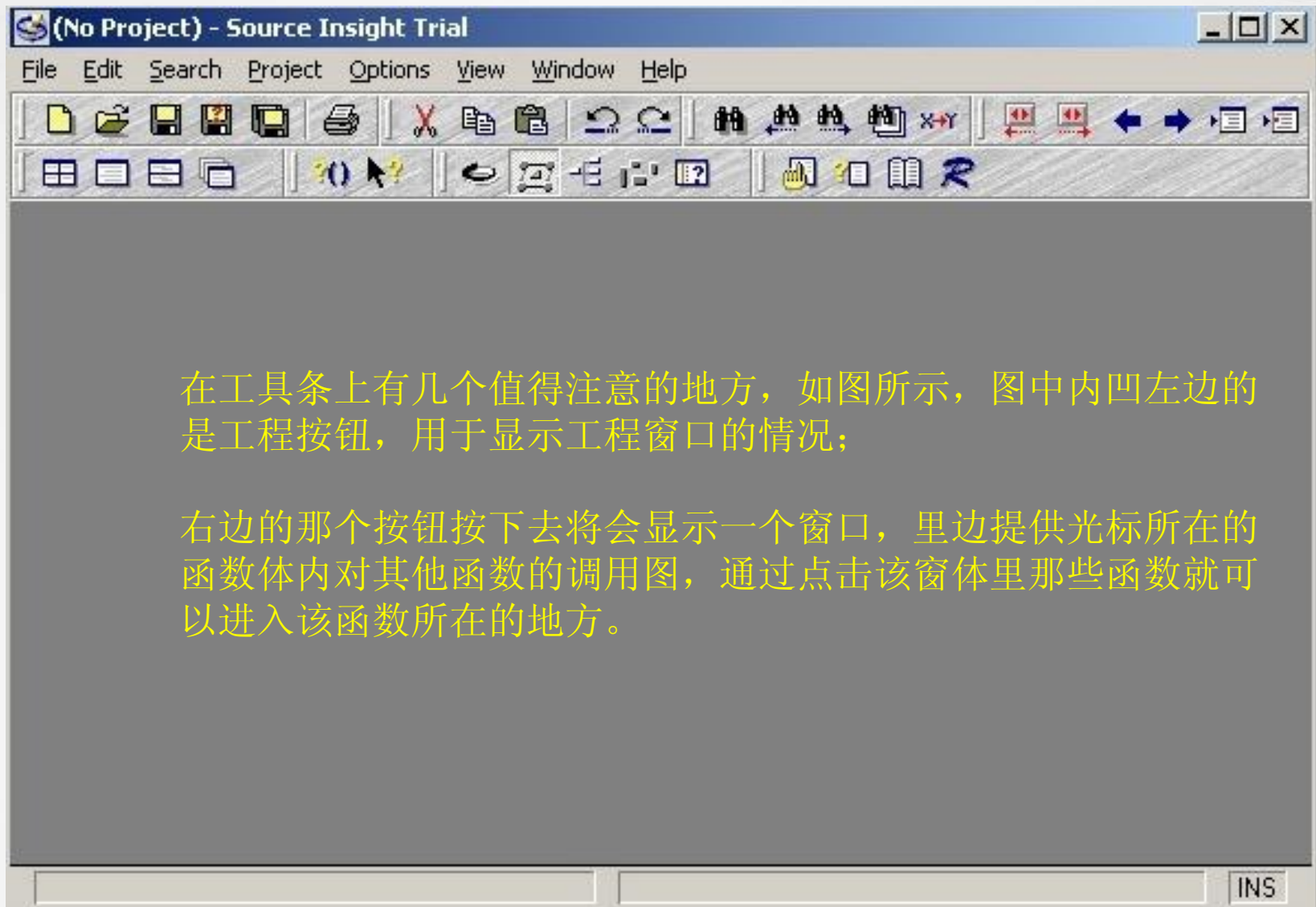


visual assist X简介

2.1 source insight的使用

- 在Window平台上，使用一些常见的集成开发环境，效果也不是很理想，比如难以将所有的文件加进去，查找速度缓慢，对于非Windows平台的函数不能彩色显示。
- Source insight 3.5一个强大的源代码编辑器，它的卓越性能使得学习源代码的难度大大降低，它是一个Windows平台下的共享软件，可以从 <http://www.sourceinsight.com/>上边下载30天试用版本。

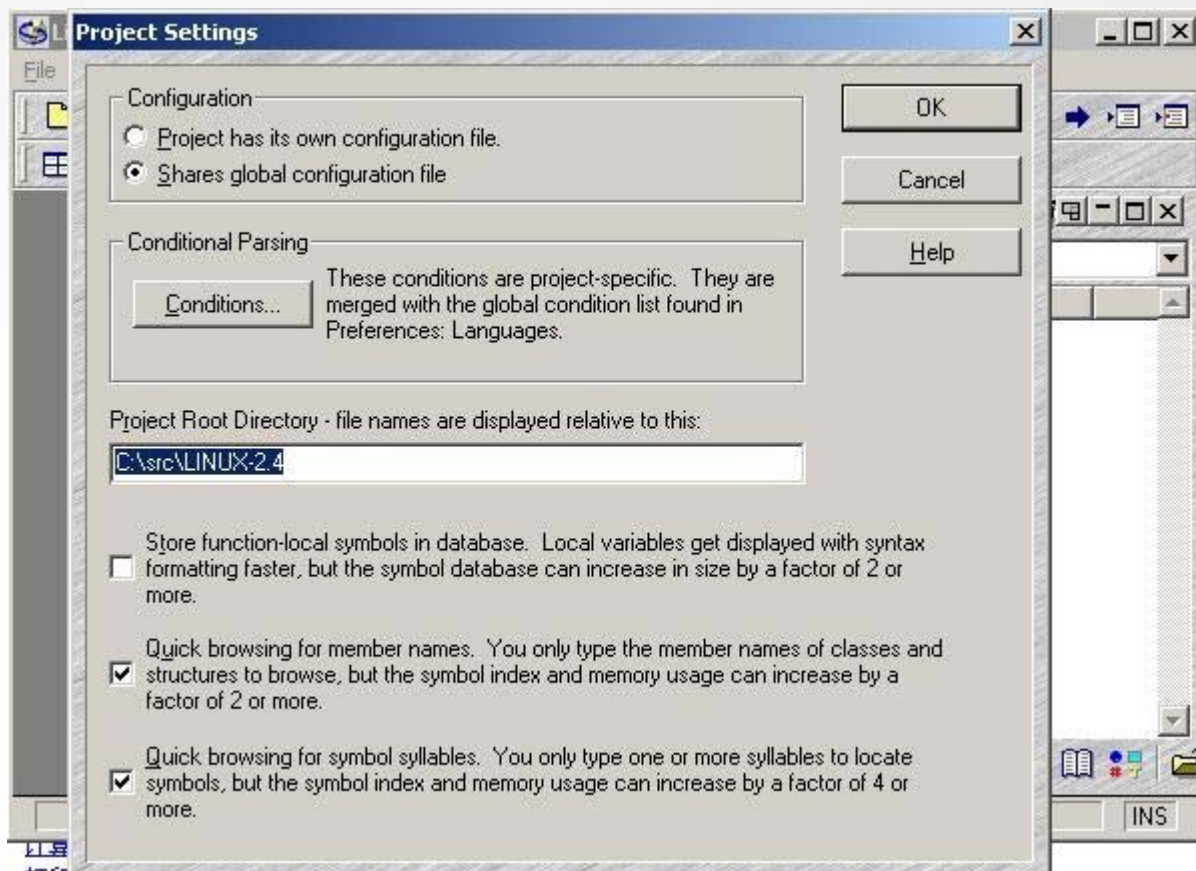
安装Source Insight并启动程序，可以进入界面



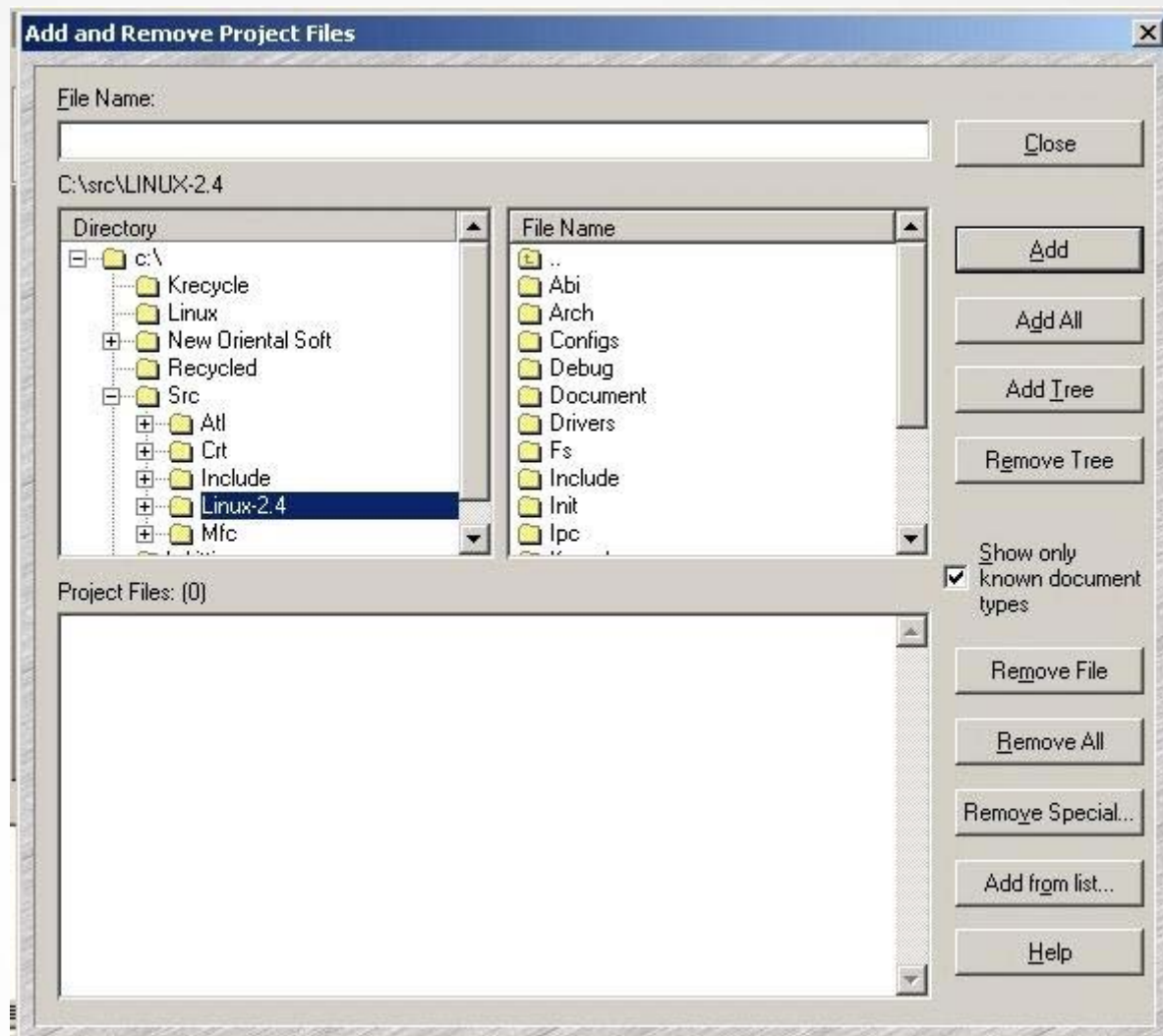
在工具条上有几个值得注意的地方，如图所示，图中内凹左边的是工程按钮，用于显示工程窗口的情况；

右边的那个按钮按下去将会显示一个窗口，里边提供光标所在的函数体内对其他函数的调用图，通过点击该窗体里那些函数就可以进入该函数所在的地方。

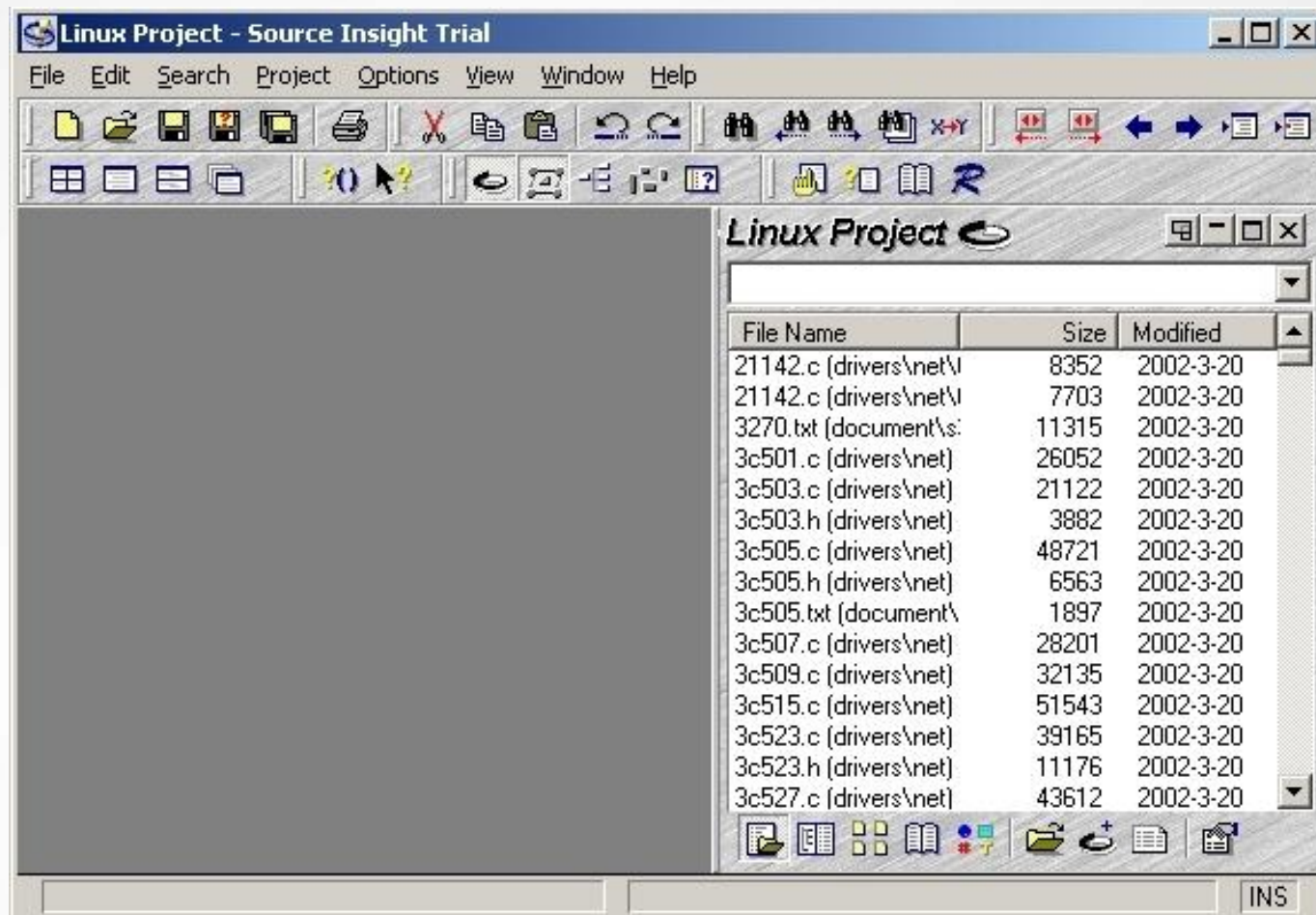
为了有效的阅读源程序，首先必须选择功能菜单上的“Project”选项的子菜单“New Project”新建一个项目，项目名称可以自由选定，当然也可以选择删除（Remove）一个项目。当删除一个项目的时候，并不删除原有的源代码文件，只是将该软件生成的那些工程辅助文件删除。设定之后，将会弹出一个对话框如图，接受默认选择，如果，硬盘空间足够，可以将第一个复选框选上，该选项将会需要与源代码大致同等的空间来建立一个本地数据库以加快查找的速度。



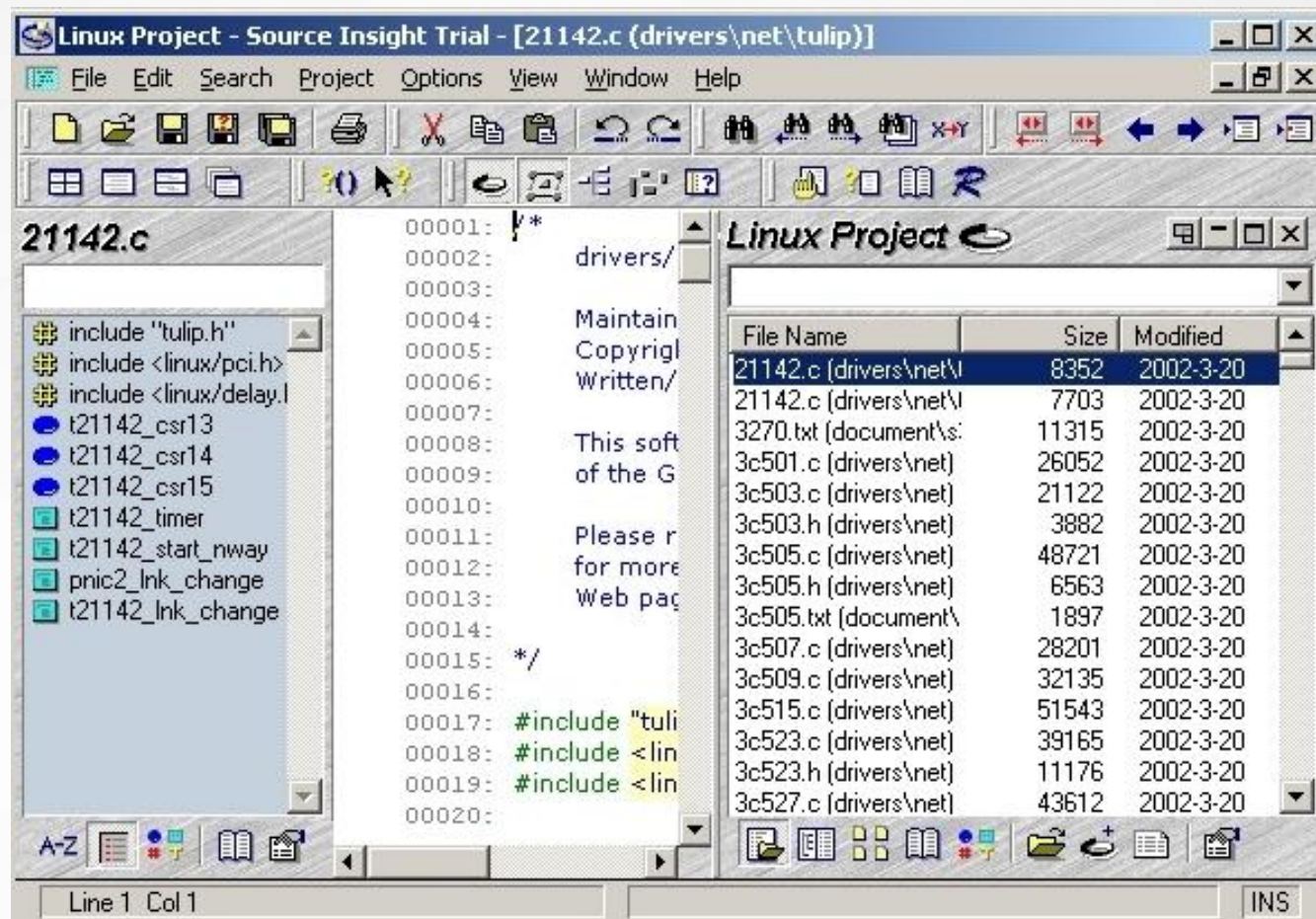
点击“OK”按钮，接受选择后，将会有一个新的对话框弹出，在这个对话框里，可以选择将要阅读的文件加入工程，一种方式是通过在File Name中输入要阅读源代码文件的名称，点击“Add”按钮将其加入，也可以通过其中“Add All”和“Add Tree”两个按钮可以将选中目录的所有文件加入到工程中，其中“Add All”选项会提示加入顶层文件和递归加入所有文件两种方式，而“Add Tree”相当于“Add All”选项的递归加入所有文件，可以根据需要使用，就我来说，更喜欢“Add Tree”一些。由于该程序采用了部分打开文件的方式，没有用到的文件不会打开，所以，加入数千个文件也不用担心加入的文件超出程序的所能容忍的最大值。



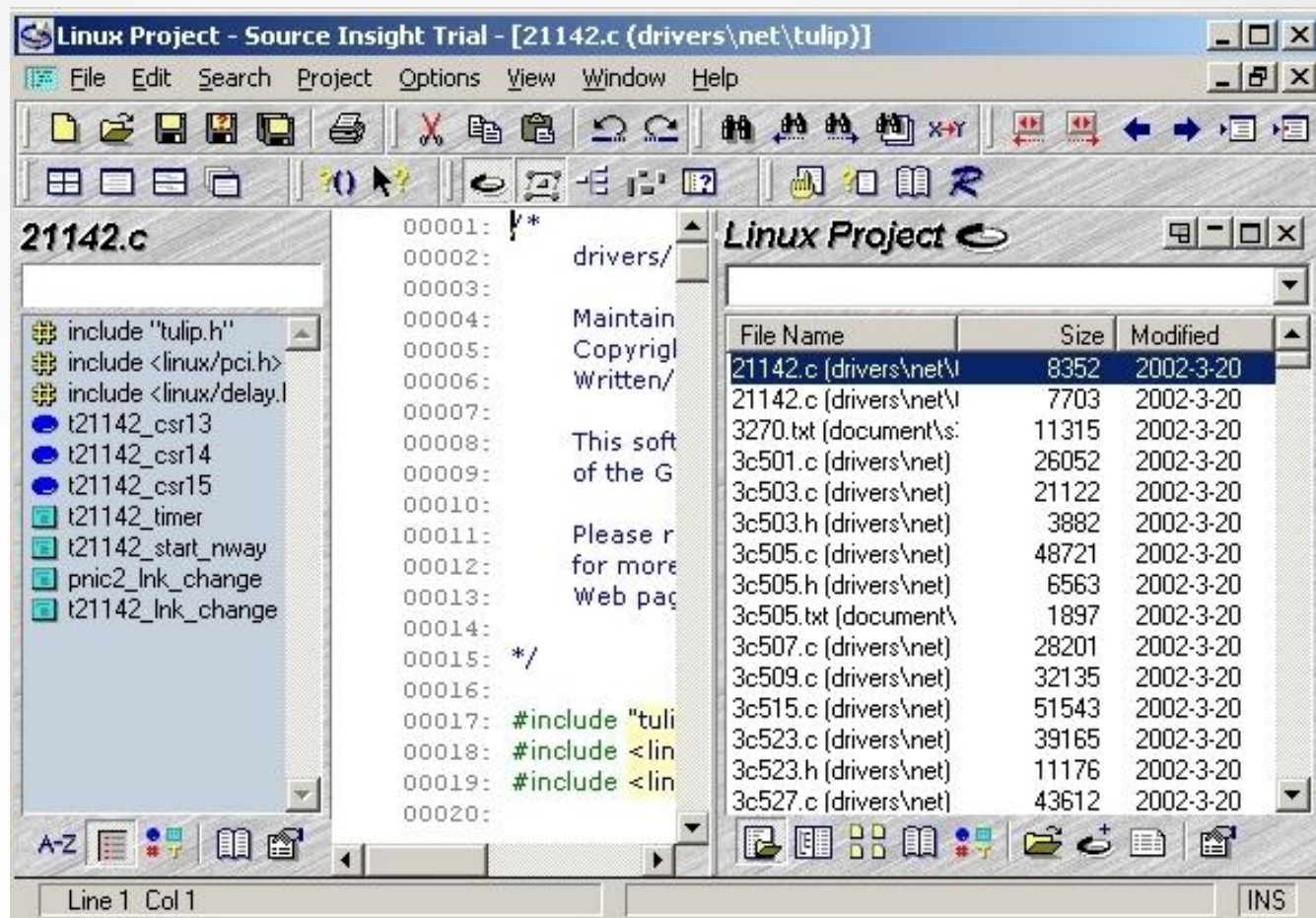
加入文件后，
点击一个
文件，可
以出现使
用界面，
如图所
示，其
中，右边
的那个窗
口（xxx
Project，
即工程窗
口）缺省
按照字母
顺序列出
当前工程
中所有的
文件。



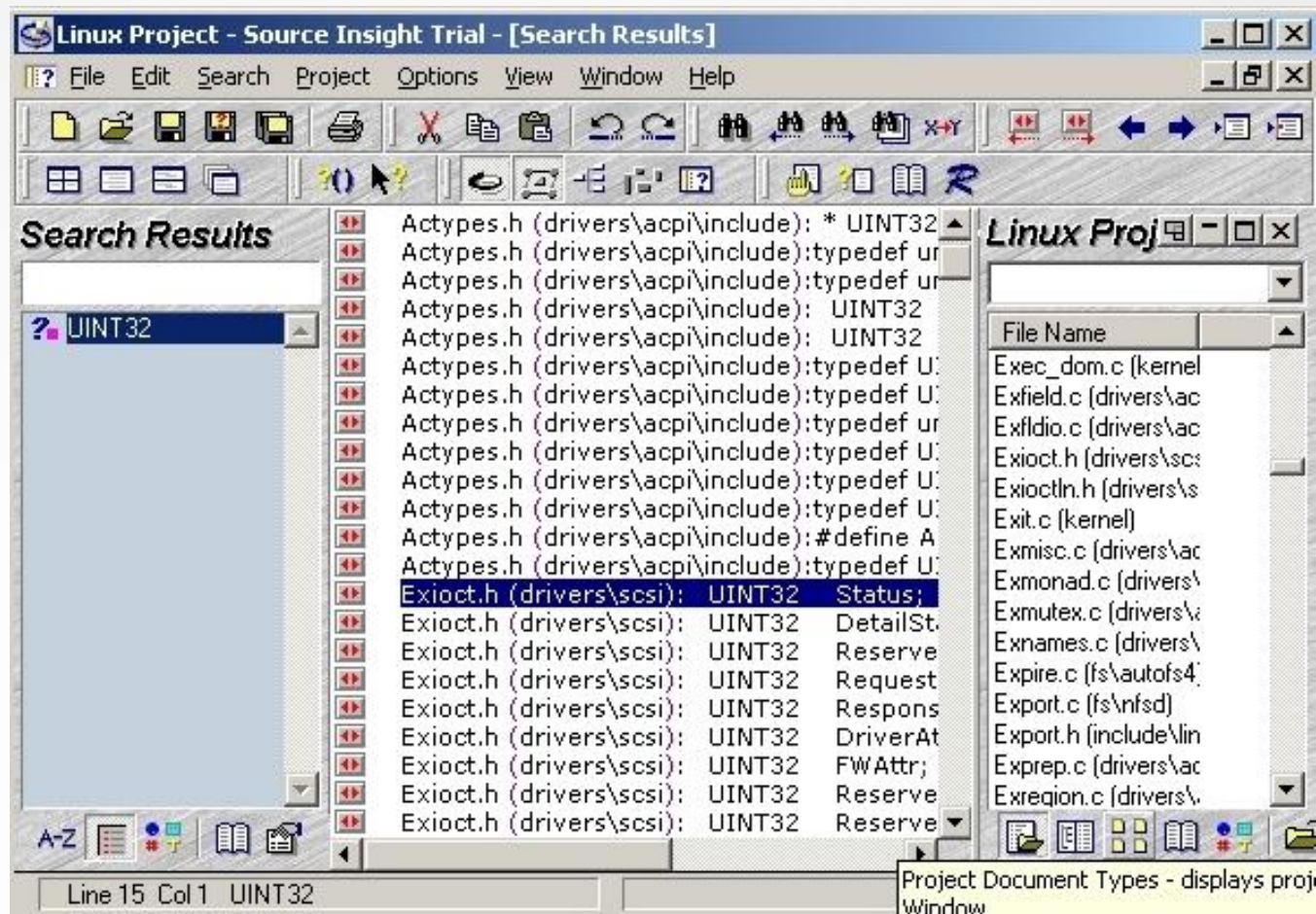
点击一个文件就可以打开该文件，显示如图所示，进入到右边的那个窗口分别可以以文件列表的方式，列出所有的文件，每个窗体下边有一排按钮，左边的窗口（filea.c）从左至右分别为：按字母顺序排列所有标记、按照文件中行数顺序排列标记、按照类型排列标记、浏览本地文件标记、标记窗口属性。



右边的窗口 (xxx Project) 从左至右分别为：按字母顺序文件列表、显示文件夹、按照文件类型归类文件、全部文件的所有标记列表、按照标记类型归类标记、跳转到定义处、显示标记信息、浏览工程标记、查找函数调用、工程属性，其中全部文件的所有标记列表选项可能要一段时间抽取标记，同步到数据库去，如果开始选择了建立标记数据库，将会在今后节省同步时间，最有用的莫过于浏览标记信息和查找函数调用，前者可以通过“Jump”按钮在不同的地方查找同样的标志，还可以通过“Reference”按钮结合后者进行全局的标记查找。



Reference功能是Source Insight的特色之一，它可以在速度极快的在整个工程中找到所有的标记，并且在该行程序的前边加上红色箭头的小按钮链接上。图是一个Reference搜索后的结果，它可以有两种模式，一种集中显示结果，图6显示的就是这种模式，在这种模式下，可以通过前边的红色箭头小按钮进入另外一种模式，该标记的具体所在处，也可以通过标记的具体所在处点击红色箭头小按钮进入另一种模式，还可以通过工具条上的两个红色小箭头直接在第二种模式下前后移动，察看相应信息。它的这个强大的功能使得阅读Linux源程序有如神助。但是要注意的是，当进行了第二次“Reference”时，它会提示你将结果集后附加在第一个结果集的后边还是取代第一个结果集。如果选择前者，不能对结果集根据前后两次搜索结果进行分类，然后在子类里进行移动，只能在整个结果集里移动；如果选择后者，结果集将被替换为第二次搜索的结果，略微有些不方便。



其他

Source Insight 还提供了一些其他常见的便利。

- 右键菜单几乎包含了程序的所有功能
- 可以在编辑窗口为程序加上行号
- 还可以统计整个工程的程序行数
- 当然还有功能强大却用不上自动完成功能
- 似乎连它的30天试用期也是一——可以迫使你尽可能快速的阅读源程序，其他一些技巧大家可以在使用过程中慢慢摸索。怎么样？爱好读源代码的朋友，不妨马上去下载一个，去开始代码阅读之旅吧

2

代码阅读的工具

2.1

▶ source insight的使用

2.2

▶ source navigator的使用

2.3

▶ beyond compare简介

2.4

▶ visual assist X简介

源代码管理分析工具 Source Navigator

- Source-Navigator是原来redhat开发的一个源代码管理分析工具，它可以在Windows，Linux等多种平台下工作。功能类似于windows下的Sourceinsight，它可以显示类，函数以及成员之间的关系，对阅读分析源代码机器有用。
- Source-Navigator 支持C, C++, Java, Tcl, [incr Tcl], FORTRAN 和 COBOL, 并且提供SDK给开发者开发自己的语言解析器

可以使用Source-Navigator:

- 分析某处源码的变化对其他模块的影响
- 查找某个函数被调用的地方
- 查找所有包含某个头文件的文件
- 利用grep工具在源码中进行搜索

- 项目主页：
<http://sourcenv.sourceforge.net/>
- 文档地址：
<http://sourcenv.sourceforge.net/online-docs/index.html>
- 下载地址：
<http://sourcenv.sourceforge.net/download.html>

详细使用说明

- http://sourcenv.sourceforge.net/online-docs/userguide/index_ug.html

2

代码阅读的工具

2.1

▶ source insight的使用

2.2

▶ source navigator的使用

2.3

▶ beyond compare简介

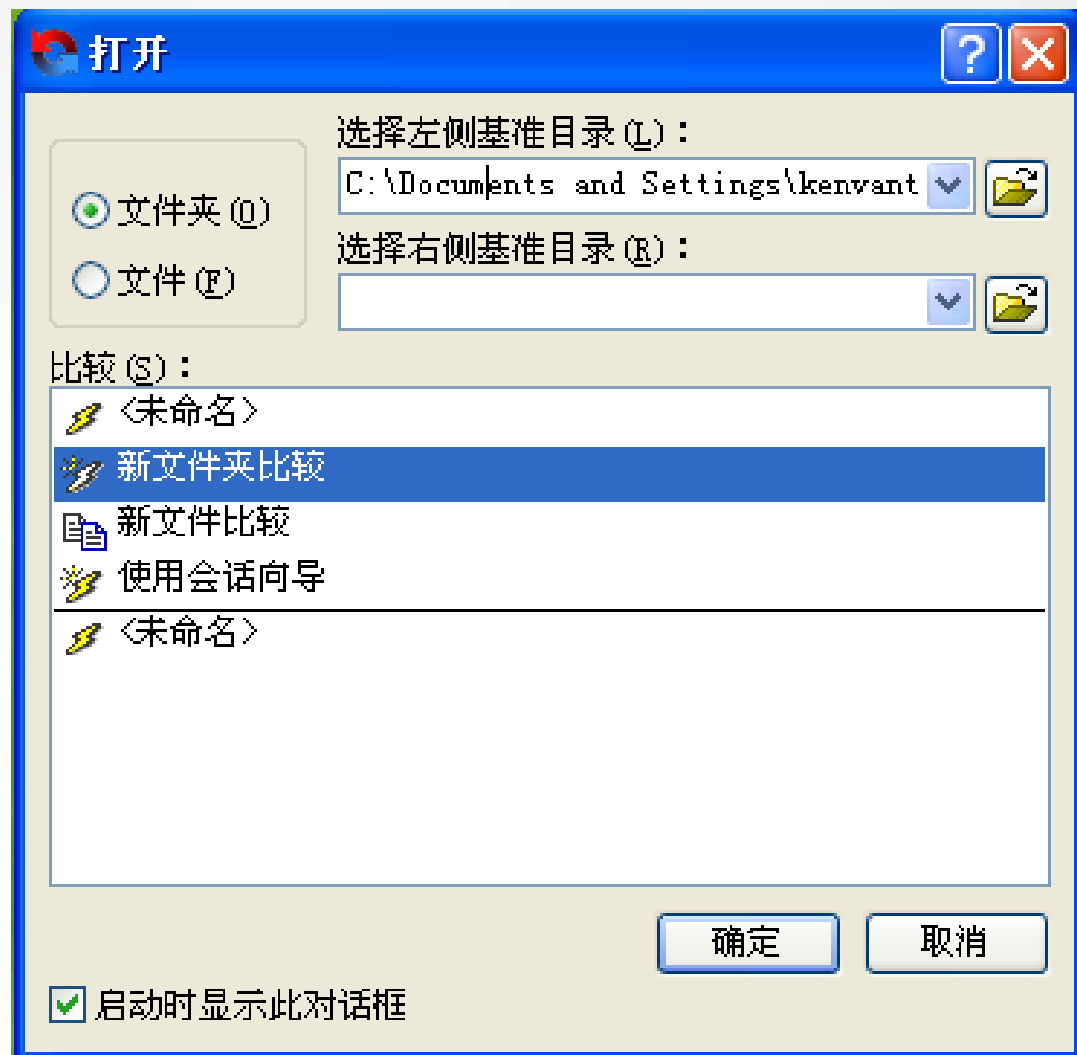
2.4

▶ visual assist X简介

Beyond Compare 使用

- Beyond Compare这是一款用于文件及文件夹比较软件，不仅可以快速比较出两个文件夹的不同之处，还可以详细的比较文件之间的内容差异。

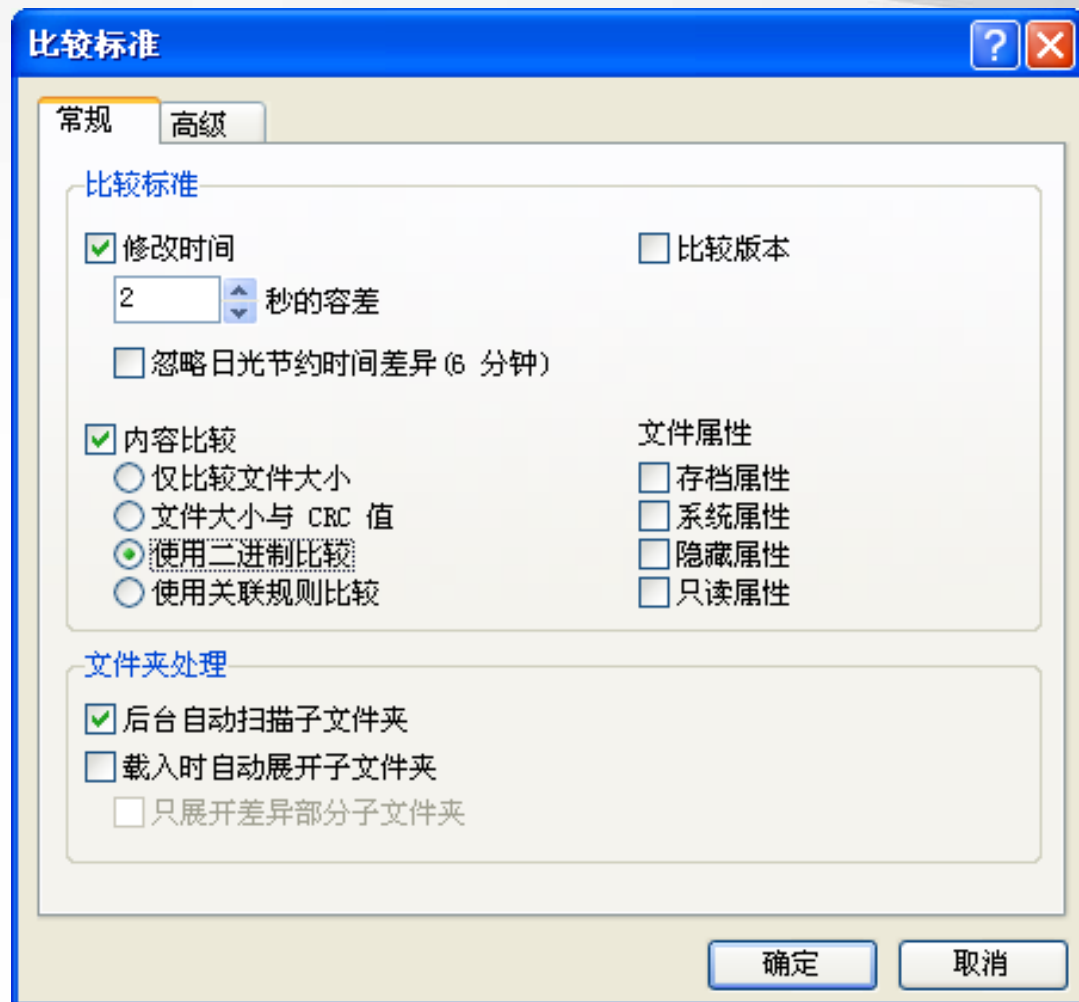
安装好
Beyond
Compare
后，我们运
行这个软
件，会弹出
引导窗口，
让我们选择
要进行对比
的文件。






































BC支持在线
对FTP的文件
进行对比
和更新，这
使得我们更
新网站程序
文件非常方
便。



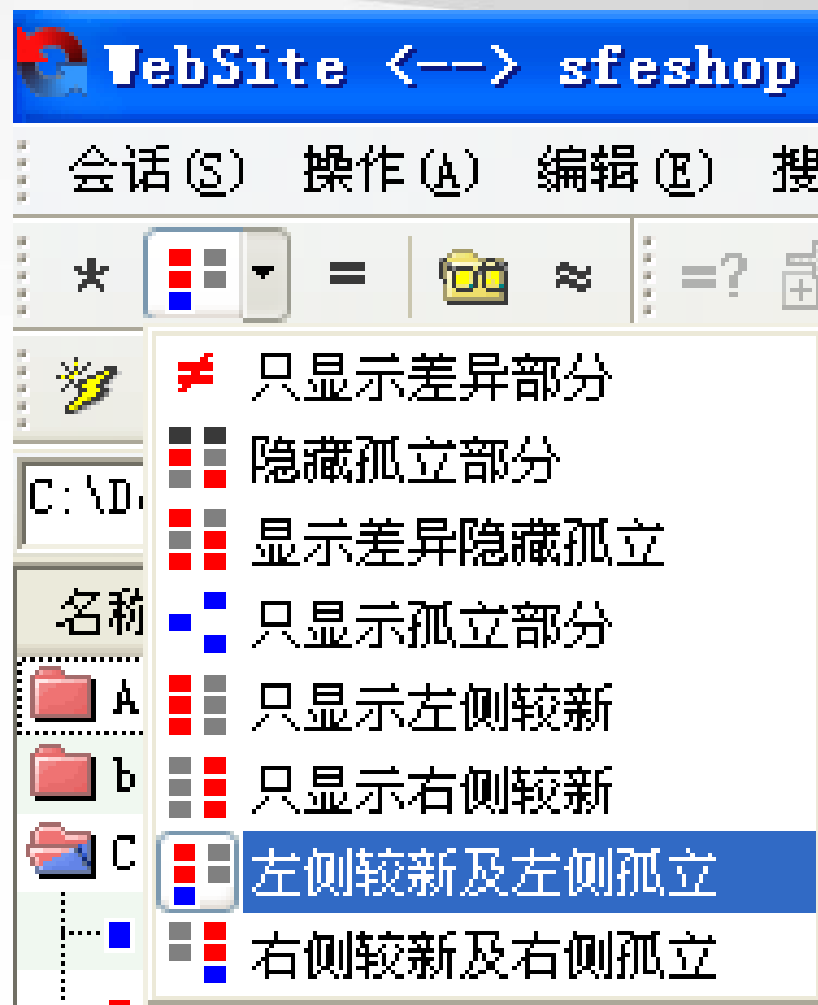
点击确定后，程序进入主界面。我们要设置比较标准：使用二进制比较。是为了实际内容，如果使用普通比较的话，如果文件时间不同，内容会被视为差异文件。



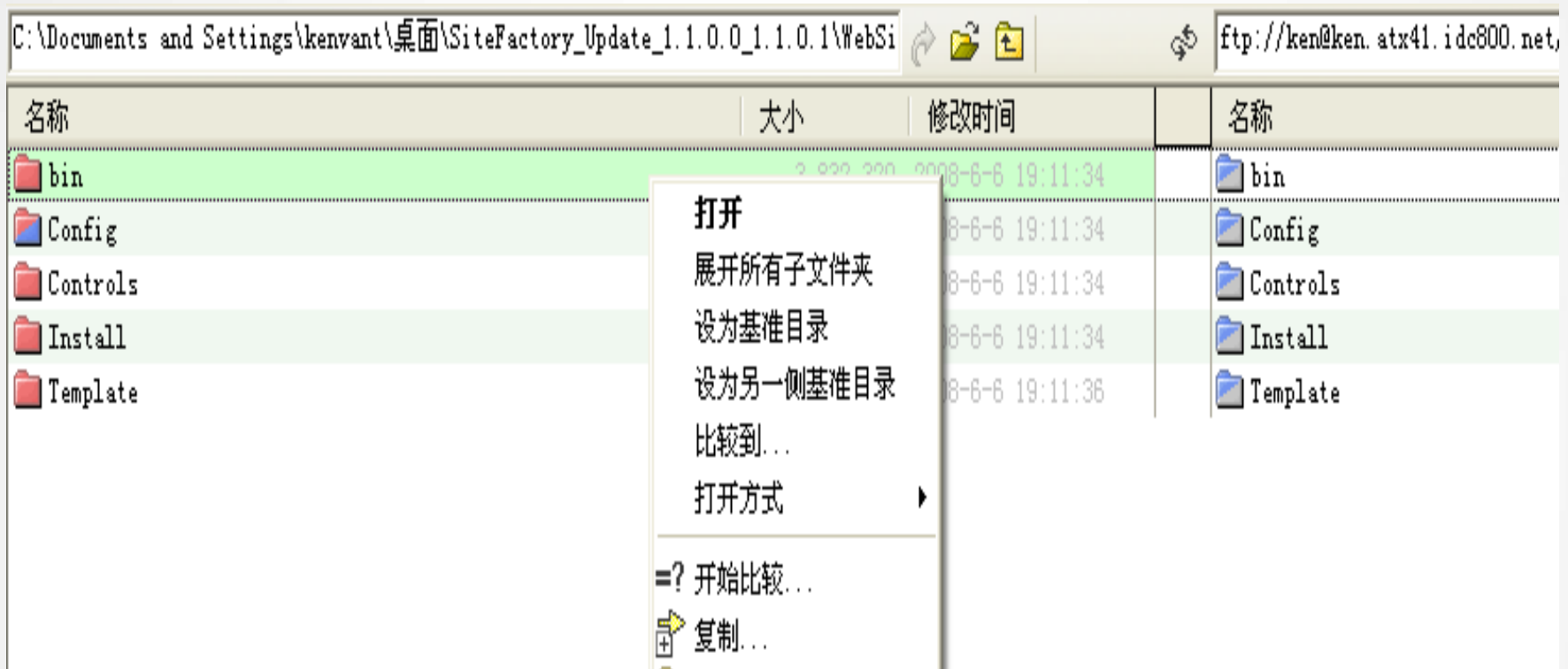
完成后软件便开始文件对比。几秒钟后比较完成，左右两侧列出文件夹的所有文件，有不同的文件则图标显示红色。

名称	大小	修改时间	名称	大小	修改时间
 Admin	134,587	2008-6-6 19:11:32	 Admin	3,928,495	2008-5-6 16:33:00
			 Analytics		2008-5-6 16:33:00
			 API		2008-6-4 10:49:00
			 App_Browsers		2008-5-6 16:16:00
			 App_Data		2008-6-4 10:49:00
			 App_GlobalResources		2008-5-6 16:19:00
			 App_Themes		2008-5-6 16:19:00
 bin	3,832,320	2008-6-6 19:11:34	 bin	11,458,940	2008-6-4 10:50:00
			 Comment		2008-5-6 16:19:00
			 Common		2008-6-4 10:50:00
			 CommonTemplate		2008-6-4 10:50:00
 Config	28,091	2008-6-6 19:11:34	 Config	121,138	2008-6-10 16:21:00
 Controls	1,126	2008-6-6 19:11:34	 Controls	327,298	2008-5-6 16:20:00
			 Editor		2008-5-6 16:21:00
			 IAA		2008-5-6 16:21:00
			 Images		2008-5-6 16:22:00
 Install	32,377	2008-6-6 19:11:34	 Install	187,087	2008-5-6 16:22:00
			 JS		2008-6-4 10:53:00
			 Languages		2008-5-6 16:22:00
			 PayOnline		2008-5-6 16:22:00
			 promote20080501		2008-5-8 15:18:00
			 Prompt		2008-5-6 16:24:00
			 Rss		2008-5-6 16:24:00
			 Shop		2008-5-6 16:24:00
			 Skin		2008-5-6 16:24:00
			 Survey		2008-5-6 16:25:00
			 Temp		2008-5-22 15:46:00
 Template	257,460	2008-6-6 19:11:36	 Template	8,853,848	2008-6-4 11:04:00
			 UploadFiles		2008-5-8 13:58:00

这时候，界面里显示出文件列表太多了，我们的查看部分，我们点击这个按钮，只显示左边的较新的或孤立的文件，也就是更新过的文件。



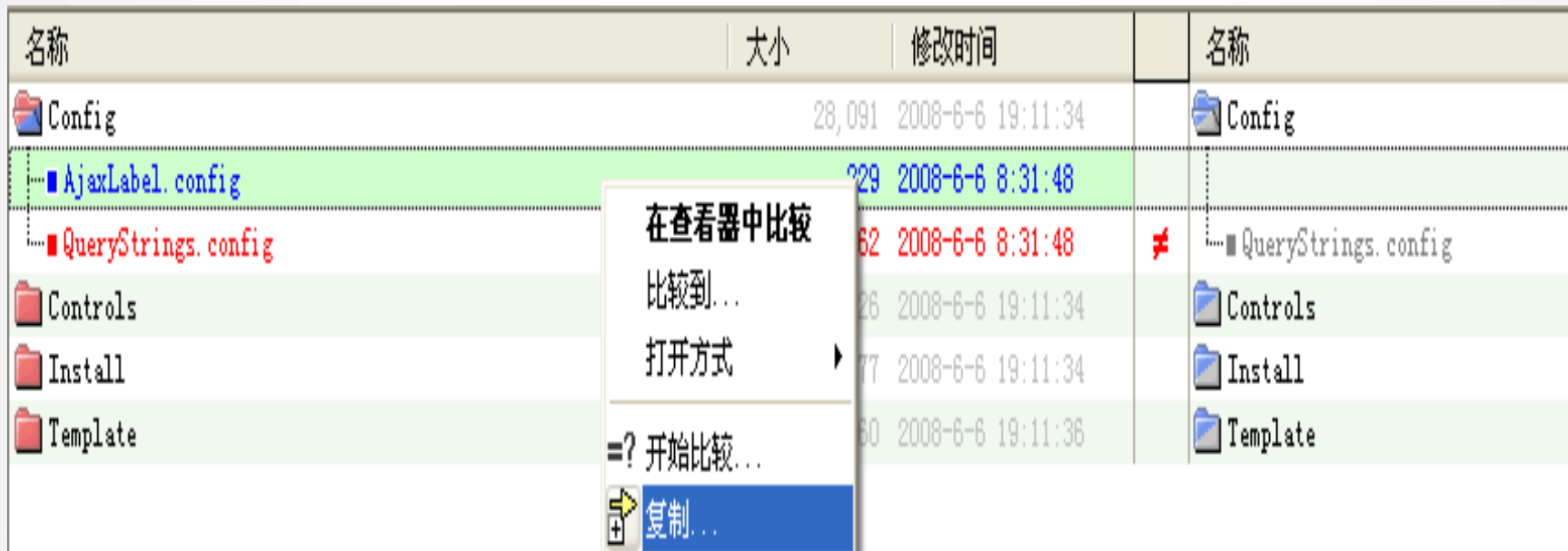
接着就是进行网站更新操作了。对于Admin、Controls、Install、Bin等文件夹，我们可执行复制命令，将这几个文件夹里的文件直接覆盖ftp里的相应文件。执行办法是，在左侧的文件夹点右键，弹出菜单中选择复制。复制完成后，左侧的红色文件夹图标变为灰色，表示已更新。



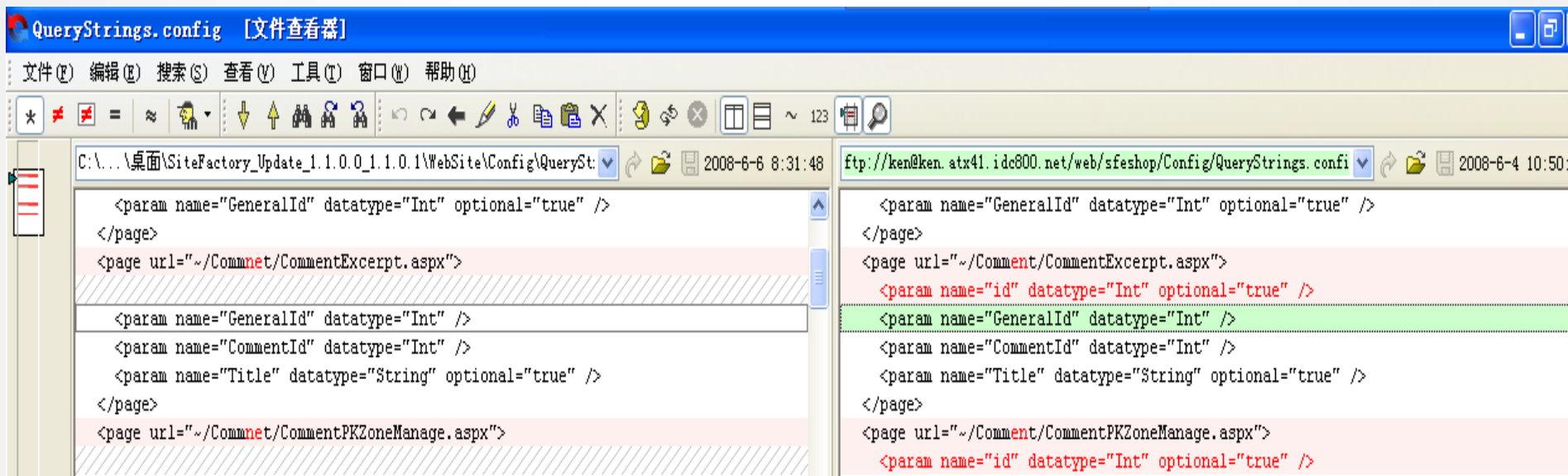
对于config、template文件夹，我们需要逐个文件进行对比覆盖。操作方法为（以config文件夹为例）：

首先双击文件夹展开文件。

1、对蓝色的左侧的孤立文件AjaxLabel.config，使用右键复制到ftp中去。



- 2、对红色的差异文件QueryStrings.config，双击打开文件查看器，进行逐行对比覆盖或编辑。点击“红色的不等号”的按钮可以只显示有差异的代码行。如果我们曾经编辑过ftp里的QueryStrings.config文件，并且想保留我们增加或修改的代码，那么我们可不对该代码进行覆盖修改。



Ok, 到此为止, Beyond Compare这款强大的工具的基本使用就介绍完毕了

2

代码阅读的工具

2.1



source insight的使用

2.2



source navigator的使用

2.3



beyond compare简介

2.4



visual assist X简介

visual assist X简介

- Visual Assist X是开发环境的辅助工具，使用该工具可以让用户更加轻松地编写代码。下面以Visual Assist 6.0为例，详细介绍它的安装与使用。

Visual Assist X主要有3个功能，具体如下。

(1) 成员列表框的出现更加频繁、迅速，并且结果更加准确。参数信息更加完善，并带有注释。含有所有符号的停驻工具提示。

使用Visual
Assist X前的
提示如
图所示。

```
class CData
{
public :
    int      Get();//取得函数
    void     Set(int data);
private:
    int      m_iData ;
};

int CData::Get()
{
    return m_iData ;
}

void CData::Set(int data)
{
    m_iData = data ;
}

void CEx010106Dlg::OnButton1()
{
    CData data ;
    data.|
}
```

A screenshot of a Visual Assist X tooltip. The tooltip is a small rectangular box with a light gray background and a thin border. It contains three items, each with a small icon to its left: a purple diamond icon next to the text 'Get', a green diamond icon next to the text 'm_iData', and a purple diamond icon next to the text 'Set'. The tooltip is positioned over the end of a line of code in a text editor.

使用Visual
Assist X后
的提示如图
所示。

```
class CData
{
public :
    int      Get(); //取得函数
    void     Set(int data);
private:
    int      m_iData ;
};

int CData::Get()
{
    return m_iData ;
}

void CData::Set(int data)
{
    m_iData = data ;
}

void CEx01010601g::OnButton1()
{
    CData data ;
    data.
}

Get
m_iData
Set
void Set (int data)
```

（2）智能提示。输入“da”，Visual Assist X会自动给出一个提示“data”（如图1-25所示），如果提示正确，可直接按回车键使用提示的内容。

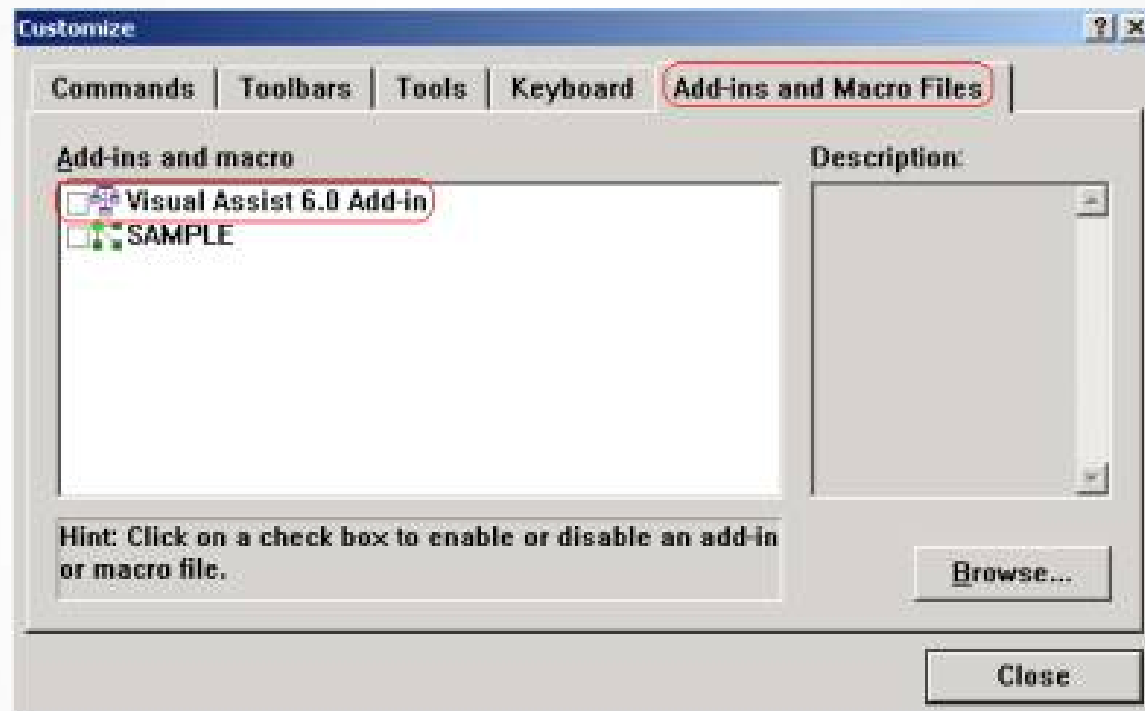
```
void CEx010106Dlg::OnButton1()  
{  
    data  
    cvaca data ;  
    da  
}
```

3) 错误自动校正： 监控您的IDE，对那些简单但耗时的错误进行即时校正。在以下代码中，输入“Cdata”，再输入空格，“Cdata”会自动变成“CData”。

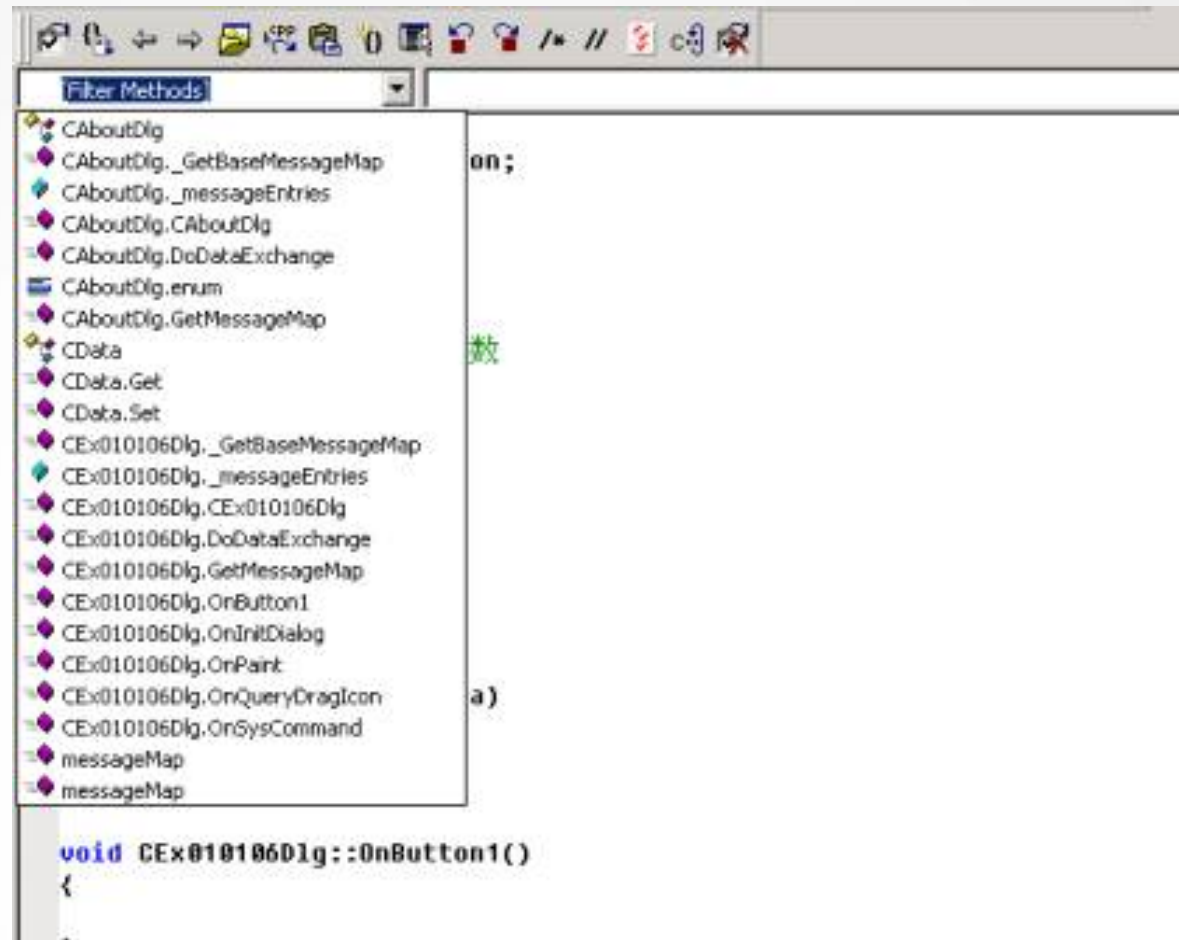
```
void CEx010106Dlg::OnButton1()  
{  
    Cdata  
}
```

Visual Assist禁用、启用

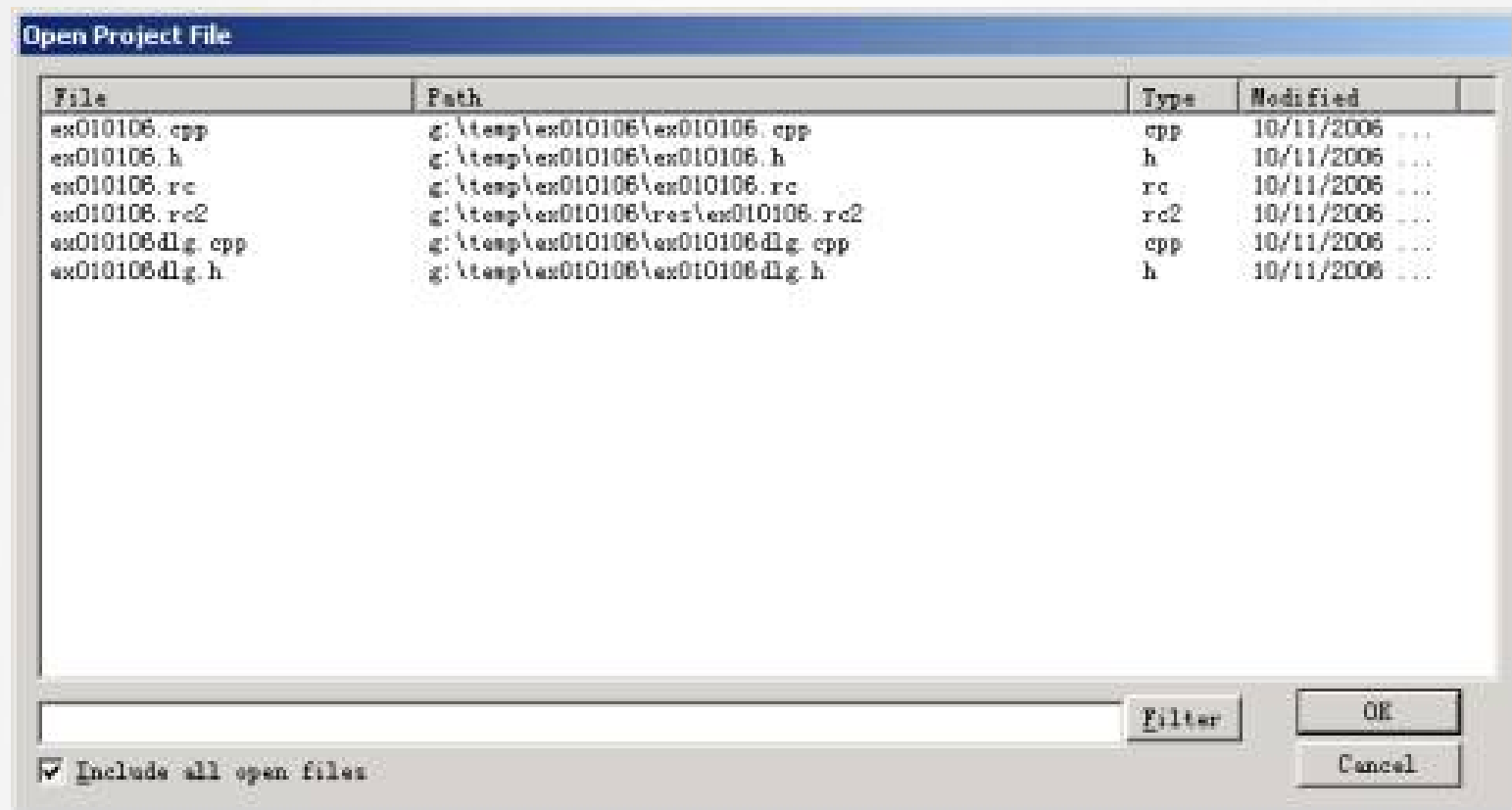
- 如果只是暂时不想使用此工具栏，可以将其禁用。选择菜单“Tools→Customize”命令，在弹出的“Customize”对话框中，取消“Visual Assist 6.0 Add-in”复选框就可以禁用Visual Assist，如图所示。
- 选中“Visual Assist 6.0 Add-in”复选框就可以重新启用Visual Assist。



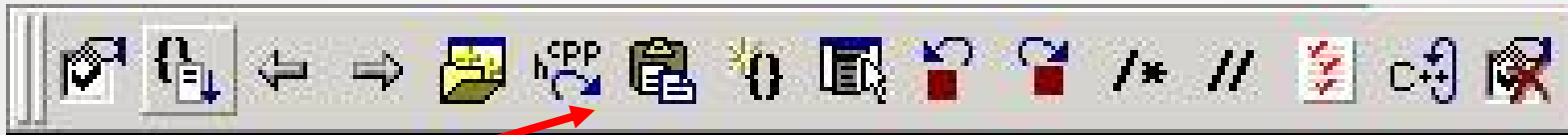
- Open Project File: 单击该按钮，打开如图所示的“Open Project File”对话框，列出本工程的部分文件，选择一个文件，直接单击“OK”按钮，即可打开此文件。



5 单击“Goto Method in Current File”按钮弹出下拉列表



Open Project File



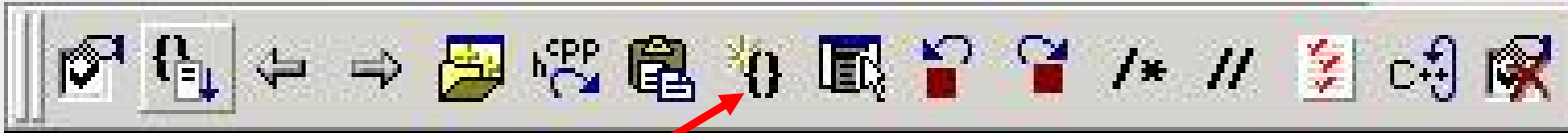
Open Corresponding.h or .cpp: 打开对应的头文件或源文件。


Paste from Multiple Buffers: 单击该按钮，从多缓冲区中进行粘贴操作。可以复制或剪切多次，然后从中选择要粘贴的内容，效果如图1-35所示。

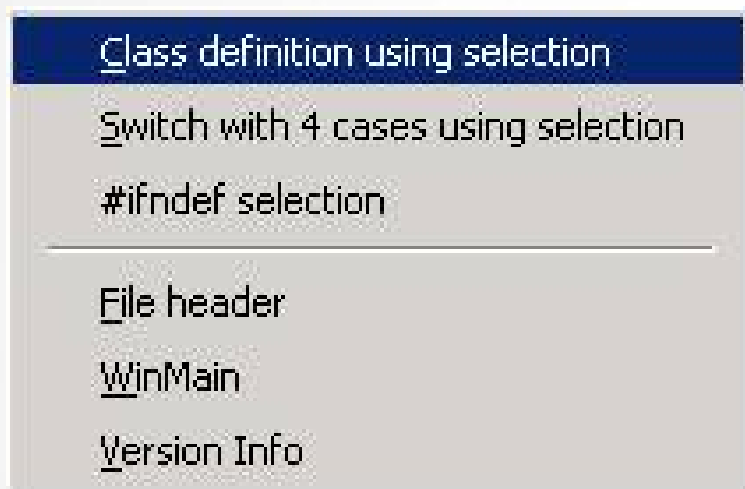
```
void CEx010106Dlg::OnButton1()  
{  
  
}
```

```
void CData::Set(int data){m_iData = ...  
int CData::Get(){return m_iData ;}  
class CData{public :int Get();//取得...
```

Paste from Multiple Buffer



-  **Insert Code Template:** 插入代码模板。单击此按钮，在弹出的菜单中选择“Class definition using selection”，如图1-36所示。



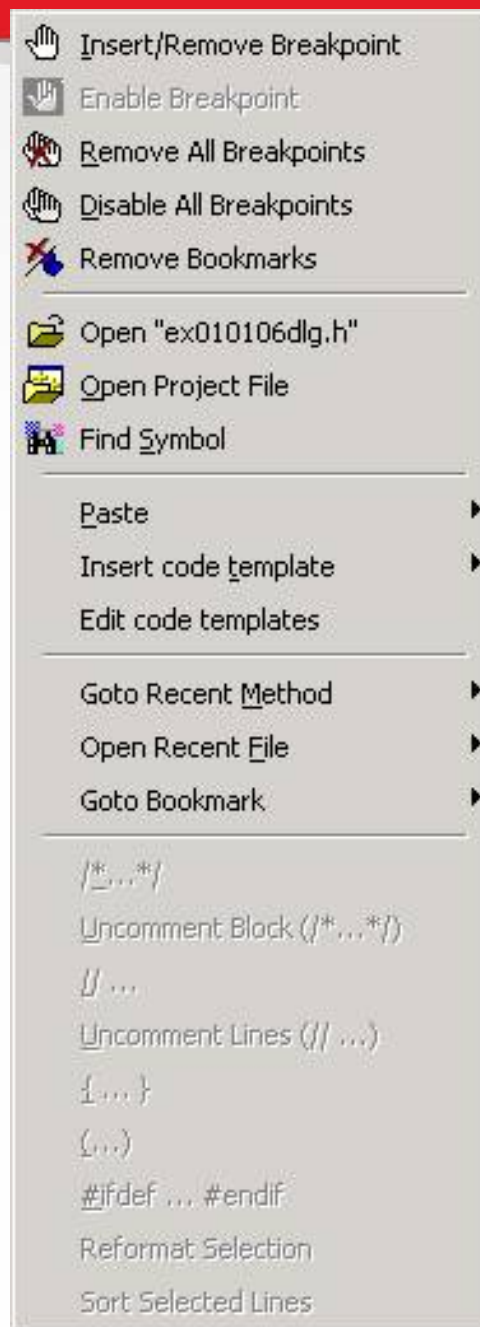
Insert Code Template

辅助工具会自动添加如下代码：

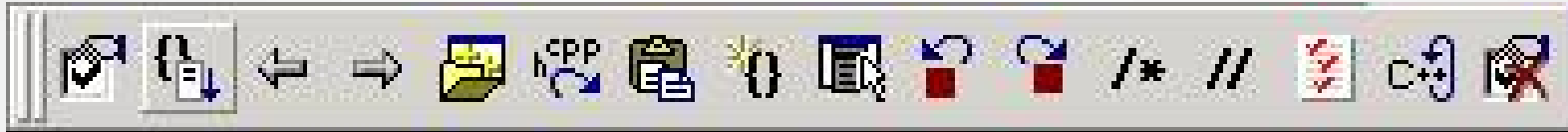
```
class
{
public:
    ();
    ~();
protected:

private:
};
```

- Display Context Menu: 显示上下文菜单，效果如图所示。



上下文菜单



-  Find Previous by Context: 联系上下文查找上一个。
-  Find Next by Context: 联系上下文查找下一个。
-  Comment Selection: 用/*和*/注释选择的内容。
-  Comment Selection: 用//注释选择的内容。
-  Spell Check Selection: 对选中内容进行拼写检查。
-  Reparse Current File: 重新分析当前文件。
-  Enable/Disable Visual Assist: 启用、禁用Visual Assist。

本章小结

- 1 认识代码阅读.....
- 2 代码阅读的工具.....
- 3 代码阅读的方法与技巧.....
- 4 大型项目代码阅读.....
- 5 课程总结 分享感言.....

3

代码阅读的方法与技巧

3.1

基本编程元素

3.2

C数据结构

3.3

控制流程

3.4

文档

3.5

命名规范与约定

3

代码阅读的方法与技巧

3.1

基本编程元素

3.2

C数据结构

3.3

控制流程

3.4

文档

3.5

命名规范与约定

一个完整的程序

Standard library headers

```

• #include <stdio.h>      <-- b
• #include <stdlib.h>     <-- c
• #include <string.h>     <-- d

• int  main __P((int, char *[]));    <-- e

• int
• main(argc, argv)
•     int argc;
•     char *argv[];
• {
•     int nflag;
•     /* This utility may NOT do getopt(3) option parsing. */
•     if (*++argv && !strcmp(*argv, "-n")) {
•         ++argv;
•         nflag = 1;
•     }
•     else
•         nflag = 0;

•     while (*argv) }    <-- f
•         (void)printf("%s", *argv);
•         if (*++argv)    <-- g
•             putchar(' ');    <-- h
•     }
•     if (!nflag)    <-- i
•         putchar('\n');
•     exit(0);    <-- j
• }

```


全局变量和函数

- 在分析重要的程序时，最好首先识别出重要的组成部分。如下，重要的组成部分是全局变量和函数。

```
int  nstops;  
int  tabstops[100];
```

← Global variables
全局变量

```
static void getstops(char *);  
        int  main(int, char *);  
static void usage (void);
```

← Forward function declarations
函数声明

要了解一个函数的功能可以使用下面的策略:

- 猜, 基于函数名;
- 阅读位于函数开始部分的注释;
- 分析如何使用该函数;
- 阅读函数体的代码;
- 查阅外部的程序文档;

While 循环、条件和块（1）

- While循环
 - 只要圆括号中指定的条件为true(在C /C++中，它求值的结果不为0)，while语句就会重复地执行其循环体。

```
while ((c = getc(pf)) != EOF)
{
    putc(c, active);
    . . .
}
```

While 循环、条件和块（2）

- while语句的循环体既可以是单个语句，也可以是一个代码块——括在花括号中的一个或多个语句。
- 所有控制程序流程的语句，也就是if,do,for和，switch，都是如此。程序中一般会将组成控制语句的相关语句(即组成if, do, for或switch代码体的语句)缩进。
- 但是，缩进只是对人类程序阅读者的一种直观提示;如果没有给出花括号，控制结构将只影响控制语句之后的单个语句，不管缩进如何。

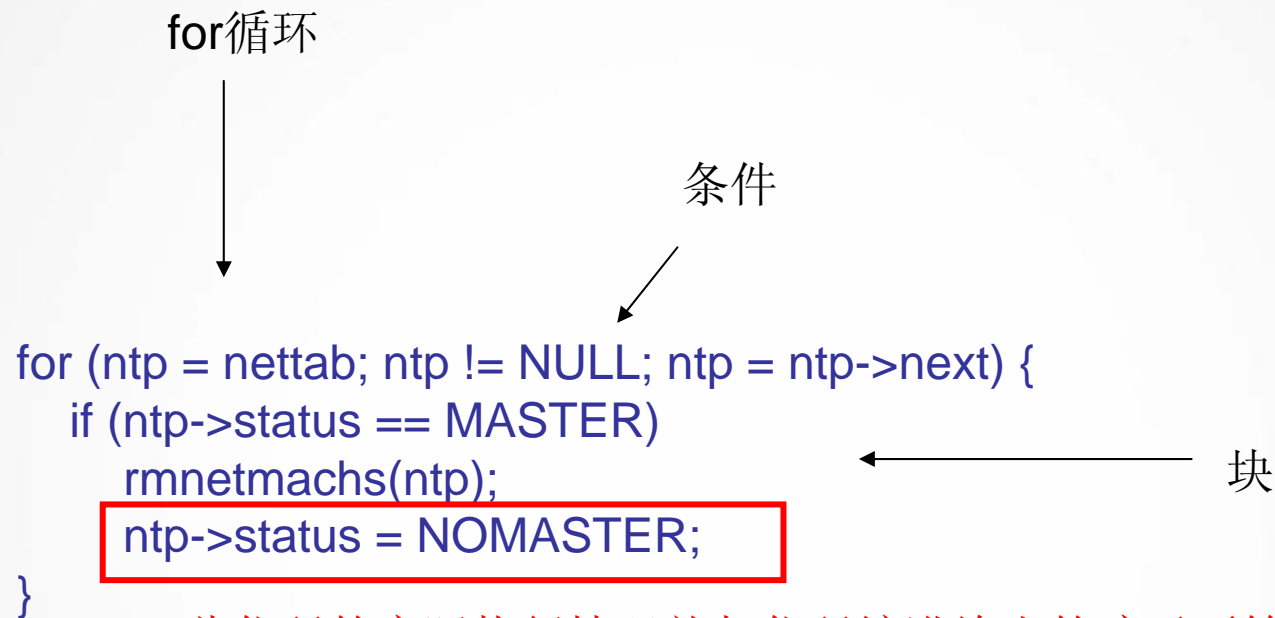
块与缩进错误示例

for循环

条件

```
for (ntp = nettab; ntp != NULL; ntp = ntp->next) {  
    if (ntp->status == MASTER)  
        rmnetmachs(ntp);  
    ntp->status = NOMASTER;  
}
```

块



此代码的实际执行情况就与代码缩进给出的暗示不符

Switch语句 （1）

- switch语句用于处理若干离散的整型或字符类型的值。处理每个值的代码之前是一个case标记。与case标记中的某个值匹配时，程序将从该点向前执行语句。
- 当switch语句中表达式的值如果没有标记值与表达式的;否则，switch块中没有代码会执行。请注意，执行控制权转移到一个标记后，遇到其他标记不会结束switch块中语句的执行;要停止处理switch块中的代码，跳转到switch语句之外继续进行，必须执行一个break语句。
- 这项特性经常用来将多个。case标记归到一起，合并公共的代码元素。

Switch语句（2）

- 缺少default标记的switch语句会默然忽略意外的值。即使知道switch语句只处理一系列确定的值，也要尽量包括default标记，这是一个好的保护性编程习惯。这类default标记能够捕获产生意外值的编程错误，并能警告程序的维护人员

```
switch (program) {  
    case ATQ:  
[...]  
    case BATCH:  
        writefile(time(NULL), 'b');  
        break;  
    default:  
        panic("Internal error");  
        break;  
}
```


~

- 一个阅读代码的模式已经慢慢形成。代码阅读有许多可选择的策略;自底向上和自顶向下的分析、应用试探法和检查注释和外部文档, 应该依据问题的需要尝试所有这些方法。

for循环

- for循环一般由3个表达式指定:在循环开始之前求值的表达式、每次迭代前都要求值以确定是否进入循环体的表达式、以及循环体执行结束后求值的表达式。for循环经常用于对一段代码体执行指定的次数。

```
for (i = 0; i < len; i++) {
```

这种类型的循环在程序中应用十分普遍;应该学会将它们读作“执行代码体len次。”

另外, 和这个风格的任何差异, 比如:初始值不是。或比较运算符不是<都警告您要小心地推理该循环的行为。请考虑下面的例子中代码体被执行的次数。

for循环

- 循环次数

1、 `for (i = 0; i <= extrknt; i++)...`

2、 `for (i = 1; i < month; i++)...`

3、 `for (i = 1; i <= nargs; i++)...`

4、 `for (code = 255; code >= 0; code--) {...`

break和continue语句

- break语句将程序转移到最内层的循环或switch语句之后执行。
- 大多数情况下，break用于提前退出循环。
- continue语句则跳过该语句到循环末尾之间的语句，继续最内层循环的迭代。continue语句会再次计算while条件表达式的值，并执行循环。
- 在for循环中，该语句将首先计算第三表达式的值，之后是条件表达式。
- continue用在循环体分开处理不同情况的地方；每种情况一般都以continue结束，以便进行下一次循环迭代。

字符和布尔型表达式

- 阅读布尔表达式时，要注意，多数现代语言中，布尔表达式只对需要的部分进行求值。
- 在用&&运算符(逻辑与)连接起来的表达式序列中，第一个表达式的求值结果如果为false，则会结束整个表达式的求值，并生成false结果。
- 在用||运算符(逻辑或)连接起来的表达式序列中，如果第一个表达式求值为true，则会终止对整个表达式的求值，产生一个true结果。
- 很多表达式都基于这种短路求值(short-circuit evaluation)特性，在阅读时也应该采用同样的方式。在阅读逻辑乘表达式时，总是可以认为正在分析的表达式以左的表达式均为true;在阅读逻辑和表达式时，类似地，可以认为正在分析的表达式以左的表达式均为false。

goto语句

- 执行某些行动后〔比如打印一条错误消息，或释放分配的资源〕.常常用goto语句退出程序或函数。

```
static int gen_init(void)
{
    [...]
    if ((sigaction(SIGXCPU, &n_hand, &o_hand) < 0) &&
        (o_hand.sa_handler == SIG_IGN) &&
        (sigaction(SIGXCPU, &o_hand, &o_hand) < 0))
        goto out;

    n_hand.sa_handler = SIG_IGN;
    if ((sigaction(SIGPIPE, &n_hand, &o_hand) < 0) ||
        (sigaction(SIGXFSZ, &n_hand, &o_hand) < 0))
        goto out;

    return(0);

out:
    syswarn(1, errno, "Unable to set up signal handler");
    return(-1);
}
```

do循环和整型表达式

- do循环的循环体至少执行一次。
- 表达式 $a \& 7$ ，用于控制循环处理代码中第一个do循环，也很有意思。 $\&$ 运算符执行两个操作数之间的逐位与 (bitwise-and) 操作。在这里，我们不是为了处理二进制位，而是屏蔽掉a变量的一些最高有效位，它返回a除以8后的余数。
- 执行算术运算时，当 $b = 2^n - 1$ 时，可以将 $a \& b$ 理解为 $a \% (b + 1)$ 。这样书写表达式是为了将除法替换为逐位与指令(有时计算起来更高效)。实际上，现代的优化编译器能够识别出这类情况，独立地完成替换，而且，除法和逐位与指令在现代处理器上运行时，速度上的差别已经不像过去那么大。
- 我们应该学会阅读使用这些技巧的代码，但要避免使用它。

3

代码阅读的方法与技巧

3.1

基本编程元素

3.2

C数据结构

3.3

控制流程

3.4

文档

3.5

命名规范与约定

C数据结构

- 1 向量
- 2 矩阵和表
- 3 栈
- 4 队列
- 5 映射 (map)
- 6 集合 (Set)
- 7 链表
- 8 树
- 9 图 (graph)

程序的作用是将算法应用到数据之上。数据的内部组织对算法的执行至关重要。

许多不同的机制都可以将相同类型的元素组织成集合，每种机制都提供各自不同形式的存储与访问数据的方式。

本章的目标是，了解如何根据底层的抽象数据类型，阅读直接的数据结构操作。

1 向量

- 我们遇到的最常用的数据结构就是向量(用于临时存储时，常被称为缓冲区)。
- 向量在一个内存区块中存储相同类型的元素，并且可以线性或随机两种方式对元素进行处理。
- C语言中，一般使用内建的数组类型实现向量，不再对底层实现进行抽象。
- 为了避免混淆，从此处开始，我们使用术语-一数组(array)，来指代基于数组的向量实现，这在C程序中很普遍。向量的应用领域很广，激发我们释放出所有的聪明才智和创造力。

1 向量

示例：

```
static char buf[128];  
memset(buf, 0, sizeof(buf));
```

示例：

```
if (fwrite(buf, sizeof(char), n, fp) != n) {  
    message("write() failed, don't know why", 0);  
}
```

2 向量

- `#define MAX_ADS 5`
- `struct dr { /* accumulated advertisements */`
- `[...]`
- `n_long dr_pref; /* preference adjusted by metric */`
- `} *cur_drp, drs[MAX_ADS];`
- `[...]`
- `struct dr *drp;`
- `[...]`
- `for (drp = drs; drp < &drs[MAX_ADS]; drp++) {`
- `drp->dr_recv_pref = 0;`
- `drp->dr_life = 0;`
- `}`

2 矩阵和表

- 在实践中，经常会遇到二维的数据结构，在数据处理领域中称为表(table)，在数学领域中称为矩阵(matrix)。
- 这两种结构在其他方面也存在不同:矩阵的元素均为相同的类型，而表的元素大多数情况下类型不同。这个区别决定了在C语言中每种结构的存储方式。

表

```
struct user *usr, *usrs;
```

```
[...]
```

```
if (!(usrs = (struct user *)malloc(nusers * sizeof(struct user))))
```

```
    errx(1, "allocate users");
```

```
[...]
```

```
for (usr = usrs, n = nusers; --n >= 0 && usr->count; usr++) {
```

```
    printf("%5ld", (long)SIZE(usr->space));
```

```
    if (count)
```

```
        printf("\t%5ld", (long)usr->count);
```

```
    printf("\t%_8s", usr->name);
```

```
    [...]
```

```
}
```

Allocate table memory

为表分配内存

Traverse table

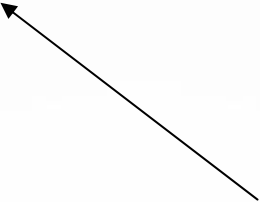
遍历整个表

Field access

访问表中的字段

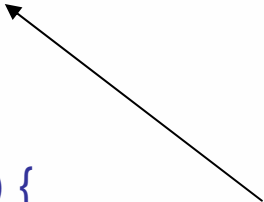
矩阵

```
typedef double Transform3D[4][4];  
[...]  
IdentMat(Transform3D m)  
{  
    register int i;  
    register int j;  
  
    for (i = 3; i >= 0; --i)  
    {  
        for (j = 3; j >= 0; --j)  
            m[i][j] = 0.0;  
        m[i][i] = 1.0;  
    }  
}
```



矩阵

```
mi_nu_compute_nurb_basis_function( order, i, knots, rknots, C );  
    [...]  
void  
mi_nu_compute_nurb_basis_function( order, span, knots, kr, C )  
    [...]  
    ddFLOAT kr[][MAXORD]; /* reciprocal of knots diff */  
    [...]  
{  
    [...]  
    for (k = 1; k <order; k++) {  
        t0 = t1 * kr[span-k+1][k];  
  
        ....  
    }  
}
```



3 栈

- 相比于向量提供的访问方式，对栈的访问方式极为有限.栈只允许以后人先出（last-in-first-out，简称LIFO）的方式在栈的结尾添加或移除元素。
- 与栈操作相伴的一个特定问题是对上溢 (overflow)和下溢C underflow)的检查，它随机地散布在程序的代码中。

3 栈

```
#define STACKMAX 32 <-- a

static int opstack[STACKMAX];
static int opsp;
[...]
```

```
PushOp(int op)
{
    if (opsp==STACKMAX) {
        strcpy(disptr,"stack error");
        entered=3;
    } else
        opstack[opsp++]=op;
}
```

```
int PopOp()
{
    if (opsp==0) {
        strcpy(disptr,"stack error");
        entered=3;
        return(kNOP);
    } else
        return(opstack[--opsp]);
}
```

```
<-- b
int isopempty()
{
    return( opsp ? 0 : 1 );
}
```

4 队列

- 队列是一个元素集合，它允许数据项以先入先出(first-in-first-out)的次序在尾部 (tail)加入，从头部(head)移除。队列经常用在两个系统连接的地方。
- 在现实世界中，队列往往形成于一个实体(例如，普通大众、订单)与其他实体(例如，银行出纳、登记处、公共汽车、生产线)的交界处。类似地，软件世界中，队列用在两个软件系统连接到一起时，或者用在软件系统与硬件交界的地方。
- 所有的情况下，队列都是用来管理两个系统不同的数据生成与处理特征。前一种情况的例子包括:窗口系统与用户应用程序的连接，操作系统对输入网络包的处理，以及邮件消息在邮件传输代理程序之间转发的方式。队列用在硬件交互中的例子包括;对网卡、磁盘驱动器、串行通信设备和打印机所生成的数据和请求的处理。所有情况下都是由一个系统生成数据，放入一个队列，在适当的间隔后，由另外的系统汉寸其进行处理。

4 队列

- 队列是一个元素集合，它允许数据项以先进先出(first-in-first-out)的次序在尾部(tail)加入，从头部(head)移除。
- 队列经常用在两个系统连接的地方。

4 队列

- 在现实世界中，队列往往形成于一个实体与其他实体的交界处。
 - (例如，普通大众、订单)
 - (例如，银行出纳、登记处、公共汽车、生产线他们的交界处。
- 软件世界中，队列用在两个软件系统连接到一起时，或者用在软件系统与硬件交界的地方。

4 队列

- 所有的情况下，队列都是用来管理两个系统不同的数据生成与处理特征。
 - 前一种情况的例子包括:窗口系统与用户应用程序的连接，操作系统对输入网络包的处理，以及邮件消息在邮件传输代理程序之间转发的方式。
 - 队列用在硬件交互中的例子包括;对网卡、磁盘驱动器、串行通信设备和打印机所生成的数据和请求的处理。
- 所有情况下都是由一个系统生成数据，放入一个队列，在适当的间隔后，由另外的系统对其进行处理。

4 队列

```
struct adbCommand adbInbound[ADB_QUEUE]; <-- 队列数组
```

```
int adbInCount=0; <-- 元素计数
```

```
int adbInHead=0; <-- 第一个要移除的元素
```

```
int adbInTail=0; <-- 第一个空元素
```

```
Void adb_pass_up(struct adbCommand *in)
```

```
{
```

```
    if (adbInCount>=ADB_QUEUE) { <-- 检查队列是否溢出
```

```
        printf_intr("adb: ring buffer overflow\n");
```

```
        return;
```

```
    }
```

```
    [...]
```

```
    adbInbound[adbInTail].cmd=cmd; <-- 添加元素
```

```
    <-- 调整尾索引，元素计数
```

```
    adbInCount++;
```

```
    if (++adbInTail >= ADB_QUEUE)
```

```
        adbInTail=0;
```

```
    [...]
```


4 队列

```
void
adb_soft_intr(void)
{
    [...]
    <-- 队列中有原属的情况
    while (adbInCount) {
        [...]
        cmd=adbInbound[adbInHead].cmd;    <-- 取出一个元素
        [...]
        adbInCount--;    <-- 调整头索引或者元素计数
        if (++adbInHead >= ADB_QUEUE)
            adbInHead=0;
    }
}
```

功能：有显式元素计数器的队列。

5 映射（map）

- 用数组的索引变量访问数组的元素效率非常高，一般可以用一个或两个机器指令实现。
- 对于组织以从零开始的连续整数为键(key)的简单映射(map3和查找表(lookup table)来说，这项特性使得数组成为它们的理想之选。
- 在预先知道数组元素的情况下，可以直接用这些元素初始化数组。

5 映射（map）

```
static const int    mon_lengths[2][MONSPERYEAR] = {  
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },  
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }  
};
```

上面的代码将一个二维数组初始化为存储正常年份和闰年中每个月的天数。

您可能会觉得存储在`mon_lengths`数组中的信息没有什么用处，仅仅是浪费空间。并非如此，数组经常用来对控制结构进行高效编码，简化程序的逻辑。考虑下面的代码如果使用`if`语句，应该怎样编写。

5 映射（map）

```
static struct key {
    char *name; <-- a
    void (*f)(struct info *); <-- b
    [...]
} keys[] = {
    <-- c
    { "all",    f_all,    0 },
    { "cbreak", f_cbreak, F_OFFOK },
    [...]
    { "tty",    f_tty,    0 },
};
[...]
<-- f
Void
f_all(struct info *ip)
{
    print(&ip->t, &ip->win, ip->ldisc, BSD);
}
```

5 映射 (map)

```
Int ksearch(char ***argvp, struct info *ip)
{
    char *name;
    struct key *kp, tmp;

    name = **argvp;
    [...]
    tmp.name = name;
    if (!(kp = (struct key *)bsearch(&tmp, keys, <-- d
        sizeof(keys)/sizeof(struct key), sizeof(struct key), c_key)))
        return (0);
    [...]
    kp->f(ip); <-- e
    return (1);
}
```

6 集合 (Set)

- 有些情况下，我们希望高效地表达和处理元素的集合(set)。当这些元素可以表达为比较小的整数时，常使用的典型实现是将该集合表达为二进制位的数组，集合中每个元素都基于特定位的值。
- C语言中，没有数据类型可数用来将二进制位表达式数组，并像数组一样进行操作。为此程序中一般使用某种整型数据类型（char, int）作为底层存储元素，用移位和逐位与/或运算符来定位特定的位。

```
pbitvec[j/BITS_PER_LONG] |= ((unsigned long)1 << (j % BITS_PER_LONG));
```

7 链表

- C程序中，最简单，也是最常用的链式数据结构是链表(finked list)。
- 链表通过指针将表达链表元素的结构联合到一起，构造而成。
- 元素一般添加到链表的前面，即链表的头部(head)。然而，由于链表中所有的元素都通过指针链接在一起，因此元素可以高效地从链表中的任何位置添加或移除，而在大型的数组中执行这类操作可能需要巨大的开销。
- 要在链表中定位一个特定的项，必须从链表的开始遍历链表;不可能随机地访问其中的元素。链表经常用来存储顺序访问的数据，这些数据在程序运行期间可以动态地扩展。

- 链表经常使用十分特殊的编码风格;因此, 代码序列的用途比较容易确定。如果结构中含有指向结构自身、名为next的元素, 一般说来, 该结构定义的是单向链表的结点。

```
struct host_list {  
    struct host_list *next;  
    struct in_addr addr;  
} *hosts;
```



```
Int search_host(struct in_addr addr)
{
    struct host_list *hp;

    [...]
    for (hp = hosts; hp != NULL; hp = hp->next) {
        if (hp->addr.s_addr == addr.s_addr)    <-- b
            return(1);
    }
    return(0);
}
```

```
Void remember_host(struct in_addr addr)
{
    struct host_list *hp;

    if (!(hp = (struct host_list *)malloc(sizeof(struct host_list)))) {
        err(1, "malloc");
        /* NOTREACHED */
    }
    hp->addr.s_addr = addr.s_addr;    <-- d
    hp->next = hosts;
    hosts = hp;
}
```

8 树

- 大量的算法和组织信息的方法依靠树作为底层数据结构。
- 树的正式定义表明:树的结点通过边连接到一起，从每个结点到树的根有且只有一条路径。树的结点可以在每一层进行扩展，这种方式经常用来有效地组织和处理数据。
- 20层深的二叉树(从最低层到根有20个结点)能够保持大约1百万个元素。许多查找、排序、语言处理、图形和压缩算法都依赖于树形数据结构。
- ，树还用来组织数据库文件、目录，设备、内存体系、属性(如Microsoft windows的注册表，X Window系统的默认规格)、网络路由、文档结构和显示元素。

- 在支持指针的语言，或支持对象引用的语言中，一般通过将父结点与它的子结点链接起来实现树形数据结构。这要用到递归的类型定义，定义中声明树由指向其他树的指针或引用构成。

```
typedef struct tree_s{  
    tree_t    data;  
    struct tree_s *left, *right;  
    short     bal;  
}  
tree;
```

```
tree_t
tree_srch(tree **ppr_tree, int (*pfi_compare)(), tree_tp_user)
{
    register int l_comp;

    ENTER("tree_srch")
    if (*ppr_tree) {
        l_comp = (*pfi_compare)(p_user, (**ppr_tree).data);
        if (l_comp > 0)
            return (tree_srch(&(**ppr_tree).right, pfi_compare, p_user))
        if (l_comp < 0)
            return (tree_srch(&(**ppr_tree).left, pfi_compare, p_user))
        /* not higher, not lower... this must be the one.
        */
        return ((**ppr_tree).data)
    }
    /* grounded. NOT found.
    */
    return (NULL)
}
```

```
int
tree_trav(tree **ppr_tree, int (*pfi_uar)())
{
    [...]
    if (!*ppr_tree)
        return (TRUE)
    if (!tree_trav(&(**ppr_tree).left, pfi_uar))
        return (FALSE)
    if (!(*pfi_uar)((**ppr_tree).data))
        return (FALSE)
    if (!tree_trav(&(**ppr_tree).right, pfi_uar))
        return (FALSE)
    return (TRUE)
}
```



9 图 (graph)

- 结点存储
- 边的表示
- 边的存储
- 图的属性
- 隐含结构
- 其他表示方法

- 图(graph)是由边连接起来的一系列的顶点(或称结点)所组成。这种定义极为宽泛, 包括树(没有回路的连通图)、集合(没有边的图)和链表(有且仅有一个边通向每个顶点的连通图)等数据组织结构。
- 本节中, 我们将分析那些不能归入上述分类的情况。遗憾的是, 图的普遍性与应用程序对它广泛多样的需求, 为图的存储与操作提供了大量令人迷惑的选项。
- 虽然从中抽取出几个具有代表性的应用模式并非完全不可能, 但是, 通过几个设计轴中找出图所处的位置, 可以对各种类型的图进行分析。

点

- 对图进行处理的算法，需要一种可靠的方式访问其中的所有结点。不同于链表和树，图中的结点不一定通过边联接起来；即使联接起来，图结构中的回路(cycle)可能致使沿着边进行系统性遍历难以实现。为此，经常要使用一种外部数据结构，将图的结点作为集合来存储与遍历。二种最常使用的方案是，将所有的结点存入一个数组或将它们链接成链表，就如同Unix tsort(拓扑排序)程序中结点的存储方式。



```
struct node_str {  
    NODE **n_prevp; /* pointer to previous node's n_next */  
    NODE *n_next; /* next node in graph */  
    NODE **n_arcs; /* array of arcs to other nodes */  
    [...]  
    char n_name[1]; /* name of this node */  
};
```

```
struct edge {  
    int id;  
    int code;  
    uset edom;  
    struct block *succ;  
    struct block *pred;  
    struct edge *next; /* link list of incoming edges for a node */  
};
```

边

- 图中的边一般不是隐式地通过指针，就是显式地作为独立的结构来表示。在隐式模型中，只是简单地将边表示为从一个结点指向另一个结点的指针。

```
struct arcstruct {  
    struct nl      *arc_parentp; /* pointer to parent's nl entry */  
    struct nl      *arc_childp;  /* pointer to child's nl entry */  
    long           arc_count;     /* num calls from parent to child */  
    double         arc_time;      /* time inherited along arc */  
    double         arc_childtime; /* childtime inherited along arc */  
    struct arcstruct *arc_parentlist; /* parents_of_this_child list */  
    struct arcstruct *arc_childlist; /* children_of_this_parent list */  
    struct arcstruct *arc_next;     /* list of arcs on cycle */  
    unsigned short  arc_cyclecnt; /* num cycles involved in */  
    unsigned short  arc_flags;     /* see below */  
};
```

边

- 图的边一般使用两种不同的方法来存储:作为每个结点结构中的数组, 或作为锚定在每个图结点上的链表。
 - 一种在定义结点的结构中声明一个数组, 一种在定义结点的结构中声明一个指向链表元素的指针, 也就是链表的头部。“锚定”的意思是指通过存储在结点中的一个变量, 可以访问到整个链表, 就好像找到船的锚, 就能够沿着铁链找到船一样。
- 大多数情况下, 图在生命期内都会更改自身的结构, 因此这个数组一般需要动态分配, 并通过存储在结点中的指针来使用。
 - 更改自身的结构是指图内的边或结点发生变化, 比如增加一条边、删除一条边等, 因此, 存储边的数组不是固定大小, 故而要动态分配。
- 对于将图的边存储为链表, 锚定在结点上的情况, 我们可以参考make程序存储元素之间依赖关系的方式。

- 分析make程序使用的结点的完整定义。每个结点(一般是程序编译配置的一部分)依赖于其他结点(它的子女), 同时被用在其他结点编译中(它的双亲)。在结点结构中, 这二者都存储为链表。

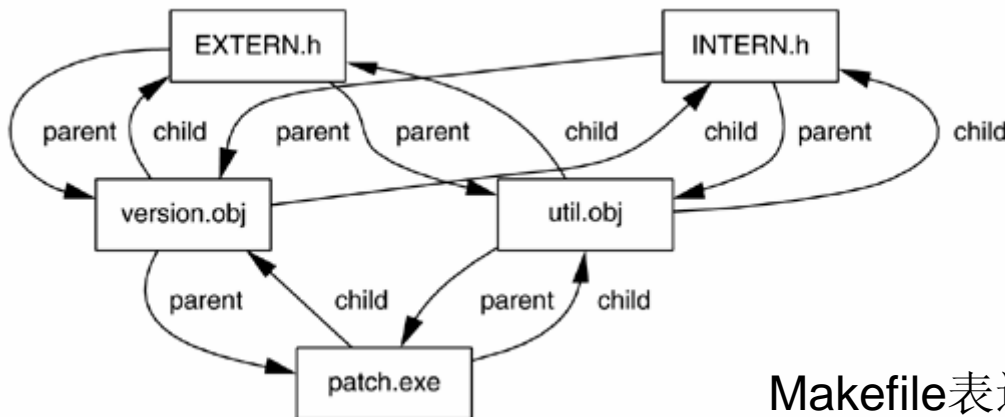
```
typedef struct GNode {  
    [...]  
    Lst    parents; /* Nodes that depend on this one */  
    Lst    children; /* Nodes on which this one depends */  
    [...]  
} GNode;
```

以makefile为例

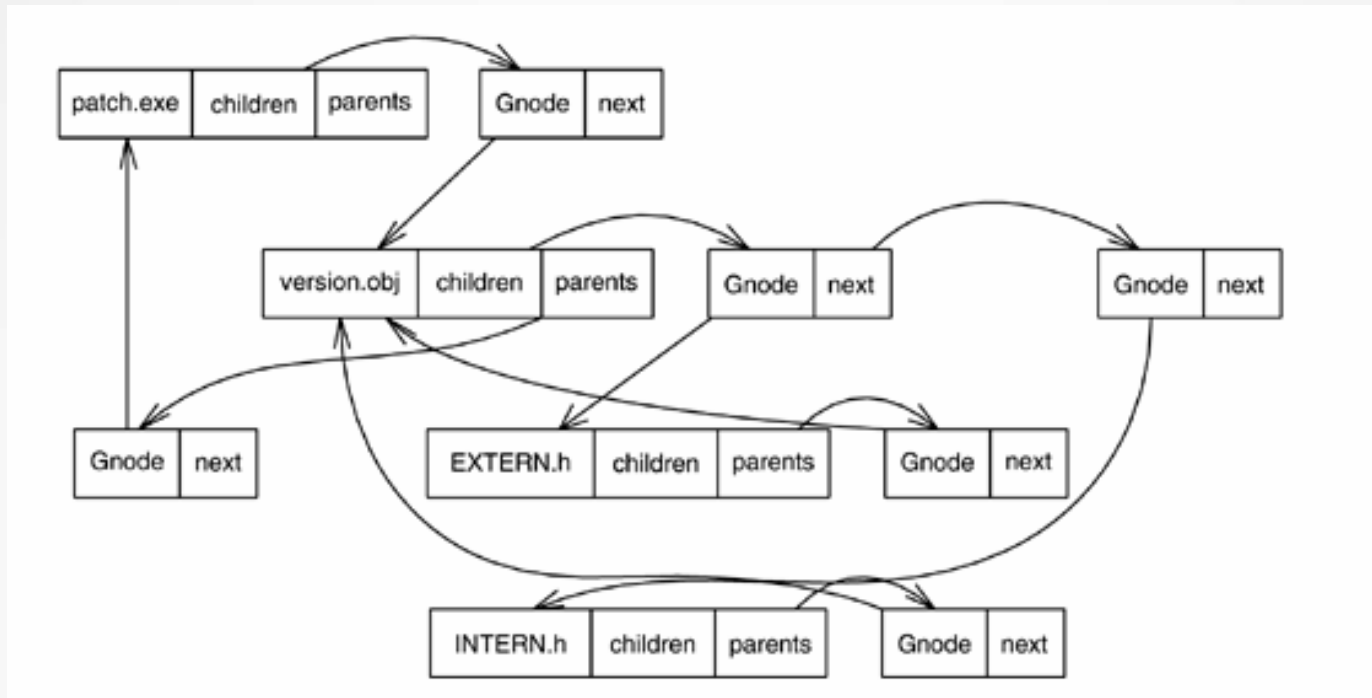
OBJS = [...] util.obj version.obj
HDRS = EXTERN.h INTERN.h [...]

patch.exe: \$(OBJS)
\$(CC) \$(OBJS) \$(LIBS) -o \$@ \$(LDFLAGS)

util.obj: \$(HDRS)
version.obj: \$(HDRS)



Makefile表达式的程序依赖关系



Make程序的数据结构中表达的依赖关系

3

代码阅读的方法与技巧

3.1

基本编程元素

3.2

C数据结构

3.3

控制流程

3.4

文档

3.5

命名规范与约定

高级控制流程

一些并不常见的部分对许多应用程序也很重要。

- 递归代码 (recursive code) 经常用相似的定义来反映数据结构或算法。
- 异常(exception)在C++和Java中用来组织对错误的处理。通过使用软件或硬件的并行性(parallelism), 程序可以增强响应性、有条理地分配工作, 或者有效地使用多处理器的计算机。
- 当并行机制不可用时, 程序可能必须采用异步信号(asynchronous signal)(能够在任意时间发出的信号)和非局部跳转(nanlocal jump)来响应外部事件。
- 为了提高效率, 程序员往往在平常调用C函数的地方, 使用C语言预处理器(C preprocessor)的宏替换(macro substitution)功能。

1 递归

- 许多数据结构——如树和堆，操作——如类型推断 (type inference) 和类型合一 (unification)，数学实体——如斐波纳契数和乡妇暇图，以及算法——如快速排序、树遍历和递归下降分析，都采用递归定义。
- 实体和操作的递归定义用它自身来定义它的对象。虽然这些定义乍看起来好像是无限循环，但实际上并非如此，这是因为基准范例的定义 (base case definition) 一般会定义一个特例，它不依赖于递归定义。
- 例如，虽然整数 n 的阶乘.

递归地打印分析命令树

```
STATIC void cmdtxt(union node *n)
```

```
{
```

```
    union node *np;  
    struct nodelist *lp;
```

```
    if (n == NULL)  
        return;
```

```
    switch (n->type) {
```

```
        case NSEMI:
```

```
            cmdtxt(n->nbinary.ch1);  
            cmdputs("; ");  
            cmdtxt(n->nbinary.ch2);  
            break;
```

```
        case NAND:
```

```
            cmdtxt(n->nbinary.ch1);  
            cmdputs(" && ");  
            cmdtxt(n->nbinary.ch2);  
            break;
```

```
        /* ... */
```

```
        case NPIPE:
```

```
            for (lp = n->npipe.cmdlist ; lp ; lp = lp->next) {  
                cmdtxt(lp->n);  
                if (lp->next)  
                    cmdputs(" | ");
```

```
            }
```

```
            break;
```

```
        /* ... */
```

```
    }
```

```
}
```



2 异常

- 异常机制允许程序员将处理错误的代码从代辦的正常控制流程中分离开来。在C++和Java程序中都会遇到类似的构造，i这些语言中用异常处理的一些错误.通过信号（signal）报告给C程序。基于异常的错误处理不同于C语言中基于信号的代码，异常作为语言的一部分(而非由库提供的功能)，能够沿着程序的词法和函数(或方法)的调用栈传播，允许忍序员以结构化的方式处理它们。

```
try {  
    [...]  
}catch (MalformedURLException e) {  
  
}catch (IOException e) {  
  
}
```

3 并行处理（parallelism）

- 硬件和软件并行性
- 控制模型
- 线程的实现

- 有些程序并行地执行部分代码，以增强对环境的响应，安排工作的分配，或有效地使用多个计算机或多处理器计算机。这种程序的设计与实现属于一个不断发展的研究领域;因此，您可能要遇到许多不同的抽象和编程模型。
- 我们将分析常见的分布式工作模型(work distribution model)中一些具有代表性的例子，避开较为特殊的实现，如涉及细粒度并行(fine-grained parallel)或向量计算(vector computation)的操作。并行运作的能力要受到硬件层和软件层的影响。

硬件和软件并行性

- 同一处理器运行多个执行单元
- 智能的集成或外部设备
- 多任务的硬件支持
- 多处理器计算机
- 粗粒度分布模型(coarse-grained distributed model)
- 进程
- 线程

控制模型

```
main(int argc, char **argv)
{
    [...]
    /* start all but one here */
    for (i=1; i<nthreads; i++) {
        closure = (struct closure *) xalloc(sizeof(struct closure));
        xthread_fork(do_ico_window, closure); <-- a
    }
    [...]
    <-- b
    void * do_ico_window(closure)
        struct closure *closure;
    {
```

- (a) Start a thread for each window
- (b) Function executed for each thread

4 信号（signal）

```
Fork::ForkProcess::ForkProcess (bool kill, bool give_reason)
: kill_child (kill), reason (give_reason), next (0)
{
  if (list == 0) {    <-- a
    <-- b
    struct sigaction sa;
    sa.sa_handler = sighnd (&Fork::ForkProcess::reaper_nohang);
    sigemptyset (&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sigaction (SIGCHLD, &sa, 0);

  }
  pid = fork ();
  if (pid > 0) {
    next = list;
    list = this;
  } [...]
}
```

```

void Fork::ForkProcess::reaper_nohang (int signo)    <-- c
{ [...]
    ForkProcess* prev = 0;
    ForkProcess* cur = list;
    while (cur) {
        if (cur->pid == wpid) {    <-- d
            cur->pid = -1;
            if (prev)
                prev->next = cur->next;
            else
                list = list->next;
            [...]
            delete cur;
            break;
        }
        prev = cur;
        cur = cur->next;
    } [...]
}

```

由信号处理器引发的竞争条件

5 非局部跳转 (nonlocal jump)

```

Int main(int argc, char *argv[])
{
    [...]
    signal(SIGINT, signal_int); <-- a
    if ((status = setjmp(env)) != 0) {
        fputs("\n?\n", stderr);
        sprintf(errmsg, "interrupt");
    } else {
        init_buffers();
        [...]
    }
    <-- b
    for (;;) {
        [...]
        if (prompt) {
            printf("%s", prompt);
            fflush(stdout);
        }
        [...]
    }
}

```

```

<-- c
void
signal_int(int signo)
{
    [...]
    handle_int(signo);
}

void
handle_int (int signo) <-- d
{
    [...]
    longjmp(env, -1); <-- e
}

```



6 宏替换

- 阅读含有宏的代码，要注意，宏既非函数，又非语句。
- 定义宏的时候，应该采用谨慎的编码准则，以避免一些常见的陷阱。
- 通过将宏的所有参数都括号化，代码避免了优先次序的问题。
- 更为困难的情况是宏参数被求值的次数。

```
#define IS_IDENT(c) (isalnum(c) || (c) == '-'  
                    || (c) == '.' || '$')  
#define IS_OCTAL(c) ((c) >= '0' && (c) <= '7')  
#define NUMERIC_VALUE(c) ((c) - '0')  
  
#define getvndxfer(vnx) do {  
    int s = splbio();  
    (vnx) = (struct vndxfer *)get_pooled_resource(&vndxfer_head);  
    splx(s);  
}while (0)
```


3

代码阅读的方法与技巧

3.1

基本编程元素

3.2

C数据结构

3.3

控制流程

3.4

文档

3.5

命名规范与约定

- 意义重大的编码工作，或大型、有组织体制之下的项目，比如GNU和BSD，都会采纳一套编码规范、指导原则或约定。
- 让计算机语言和编程系统为程序员如何表达一个给定的算法提供了大量的余地。
- 代码规范提供风格上的指导，目标是增强代码的可靠性、易读性和可维护性。
- 在阅读代码时，规范和约定可以为您提供特定代码的额外指引，从而增加阅读的效率。

1 文档的类型

- 应用传统工程方法的项目，在开发过程中，会生成大量不同的文档。当这些文档得到正确维护时，它们确实能够帮助您理解系统的基本原理、设计和实现。

系统规格说明文档 (system specification document)

- 系统规格说明文档(system specification document)详细描述系统的目标、系统的功能需求、管理和技术上的限制、以及成本和日程等要素。通过系统的规格说明文档能够了解所阅读代码的实际运行环境。同样一段绘图代码，用在屏幕保护程序中，或作为核反应控制系统的一部分，您对它的解读态度会完全不同。

软件需求规格说明

(software requirements specification)

- 软件需求规格说明(software requirements specification)提供对用户需求和系统总体构架的高层描述，并且详细记述系统的功能和非功能性需求，比如数据处理、外部接口、数据库的逻辑模式以及设计上的各种约束。由于软硬件环境和用户需求的不断变化，文档中还可能描述预期的系统演化。软件需求规格说明是阅读和评估代码的基准。

设计规格说明(design specification)

- 设计规格说明(design specification)一般描述系统的构架、数据和代码结构，还有不同模块之间的接口。面向对象的设计会勾画出系统的基本类型以及公开方法。细化的设计规格一般还包括每个模块(或类)的具体信息，比如它执行的处理任务、提供的接口，以及与其他模块或类之间的关系。另外，设计规格说明还会描述系统采用的数据结构，适用的数据库模式等。我们可以将系统的设计规格说明作为认知代码结构的路线图、阅读具体代码的指引。

系统的测试规格说明(test specification)

- 系统的测试规格说明(test specification)包括测试计划、具体的测试过程、以及实际的测试结果。每个测试过程都会详细说明它所针对的模块以及测试用例使用的数据(从中可以得知特定的输入由哪个模块处理)。利用测试规格说明文档提供的数据,可以预演正在阅读的代码。

用户文档(user documentation)

- 由许多不同文档组成的用户文档(user documentation), 这些文档包括功能描述、安装说明、介绍性的导引、参考手册和管理员手册。
- 用户文档的最大优点是, 它常常是我们惟一有可能获得的文档。
- 在接触一个未知系统时, 功能性的描述和用户指南可以提供重要的背景信息, 从而更好地理解阅读的代码所处的上下文。
- 从用户参考手册中, 我们可以快速地获取, 应用程序在外观与逻辑上的背景知识, 从管理员手册中可以得知代码的接口、文件格式和错误消息的详细信息。

2 阅读文档

- 利用文档可以快捷地获取系统的概况，了解提供特定特性的代码。
 - 以Berkeley Unix快速文件系统(fast file system)的实现为例。
 - 在系统的管理者手册中，这个文件系统的帮助信息由14个编排好的页面组成，详细描述系统的组织方式、参数化、布局策略、性能以及功能L的增强。
 - 阅读这些文字要比分析构成系统的4 586行源代码容易得多即使是在分析较短的代码片断时，通过阅读文档了解了它们的用途之后，它们的功能也会变得更为清晰。

2 阅读文档

- 文档给出软件系统的规格说明，我们可以依照它对代码进行审查。
- 功能性规格说明可以作为阅读代码的起点。
- 代码大多会支持某种特定的产品或接口规范，所以，即使找不到指定系统的功能性规格说明，也可以使用对应的规范作为向导。
 - 以审查apache web服务器对HTTP协议的顺从性为例。

[练习]在apache源代码中，您会发现下面的代码序列。

2 阅读文档

```
switch (*method) {  
    case 'H':  
        if (strcmp(method, "HEAD") == 0)  
            return M_GET; /* see header_only in request_rec */  
        break;  
    case 'G':  
        if (strcmp(method, "GET") == 0)  
            return M_GET;  
        break;  
    case 'P':  
        if (strcmp(method, "POST") == 0)  
            return M_POST;  
        if (strcmp(method, "PUT") == 0)  
            return M_PUT;  
        if (strcmp(method, "PATCH") == 0)  
            return M_PATCH;
```

对照HTTP协议的规格说明文档，RFC-2068，对代码进行检查，可以容易地验证所实现命令的完全性，以及是否存在扩展。

- 文档对非功能性需求背后的理论基础—安全.做了详尽的描述，对理解它在源代码中的实现很有帮助。
- 此外，在系统的文档中，您经常能够找到设计者当时的观点;系统需求的目标、目的和意图，构架，以及实现;还有当时否决的其他方案，以及否决它们的理由。

- 文档还会说明内部编程接口。大型的系统一般会划分成多个小一些的子系统，这些子系统通过精确定义的接口进行互操作。
- 另外，重要的数据集合经常组织为抽象数据类型，或类似的接口定义完善的类。

- 文档也提供测试用例，以及实际应用的例子。源代码或系统的功能性文档，常常不会为我们提供如何实际使用系统的信息。tcpdump程序支持一种复杂的语法，可以精确地指定希望分析的网络包。然而，如果没有实际应用的例子，就难以理解源代码正试图完成什么；测试用例为我们提供可以对代码进行预演(dry-runs)的材料。

程序的手册页——和大多数unix手册页相同——提供了10种不同的典型应用，如下：

实现问题或bug

- 文档常常还会包括已知的实现问题或bug。有时，您可能为了解源代码的特定部分如何正确地处理一个给定的输入，在该代码上花费数个小时的时间。通过阅读文档，您或许会发现系统根本不处理您正在分析的特殊情况，这是一个文档编制上的bug.

文档的变更

- 文档的变更能够标出那些故障点。许多系统的源代码中都包括某种形式的维护文档。它们至少会记录每次发布中所做的更改，不是以我们在fi. 5节中描述的那样以修订控制系统日志条目的形式，就是记录在纯文本文件中，一般命名为Changel,og(对更改的注释按发生的时间进行排列)。分析变更日志常常会揭示出那些重复，有时互相冲突，做出更改的区域，或类似的修复应用到源代码的不同部分。有时，第一种类型的更改表示存在根本性的设计缺陷，从而使得维护人员需要用一系列的修补程序来修复。Linux smbfsC}\Vind}\ws网络文件系统)代码的ChangeLog中的下列条目就是一个典型的例子。

changelog, 如下:

- 2001-09-17 Urban [...]
- * proc.c: Go back to the interruptible sleep as reconnects
- seem to handle it now.
- [...]
- 2001-07-09 Jochen [...]
- * proc.c, ioctl.c: Allow smbmount to signal failure to reconnect
- with a NULL argument to SMB-IOC-NEWCONN (speeds up error
- detection).
- [...]
- 2001-04-21 Urban [...]
- * dir.c, proc.c: replace tests on conn-pid with tests on state
- to fix smbmount reconnect on smb_retry timeout and up the
- timeout to 30s.
- [...]
- 2000-08-14 Urban [...]
- * proc.c: don't do interruptable_sleep in smb_retry to avoid
- signal problem/race.
- [...]
- 1999-11-16 Andrew [...]
- * proc.c: don't sleep every time with win95 on a FINDNEXT

3 文档存在的问题

在阅读文档时，要时刻注意，文档常常会提供不恰当的信息，误导我们对源代码的理解。两种不同类型的歪曲是未记录的特性(undocumented feature)和理想化表述(idealised presentation).

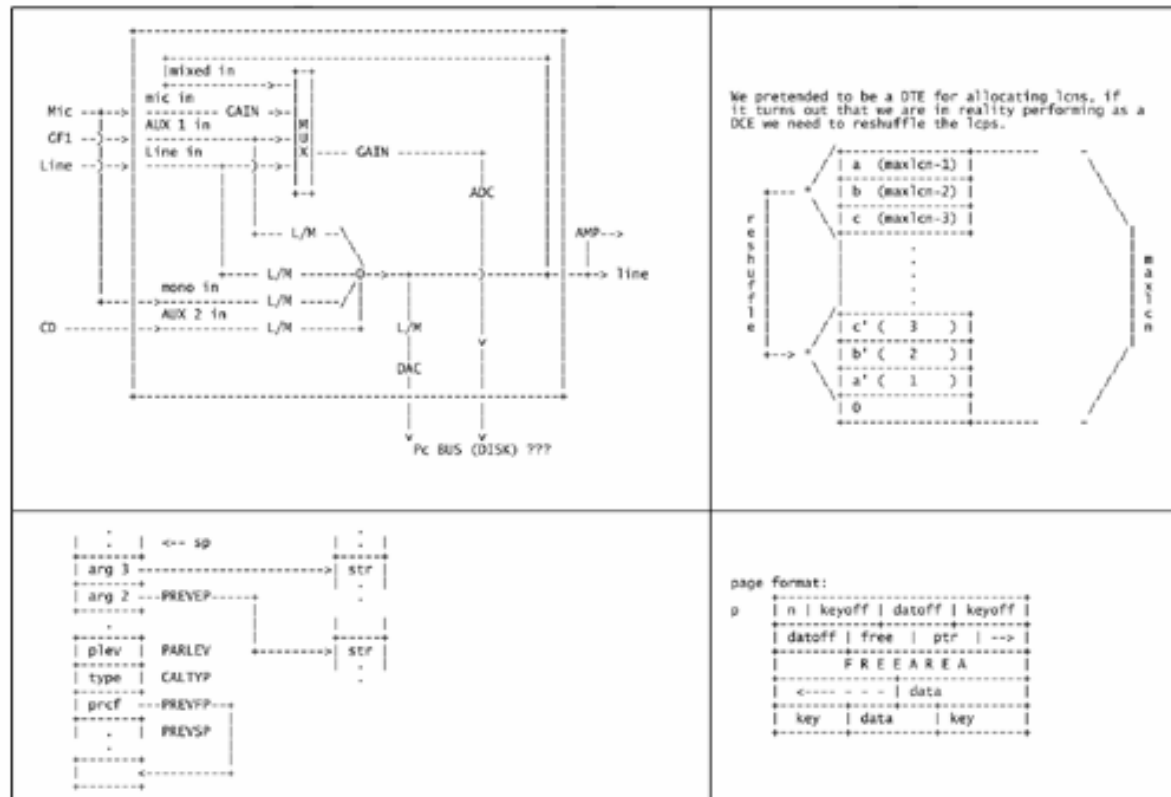
- 软件代码中已经实现的特性可能由于许多不同的原因(有些是正当的、可以理解的)故意没有编入文档。
- 特性之所以没有记录可能是因为该特性：
 - 并非正式支持；
 - 只是作为一种支持机制，提供给合适的、经过培训的工程师使用；
 - 试验性或打算用在将来的版本中；
 - 由产品提供商使用，从而在竞争中取得优势；
 - 实现的不好；
 - 安全威胁；
 - 仅仅供少数用户或产品版本使用；
 - 特洛伊木马、时间炸弹或后门。

第二种错误是对系统的理想化表述

- 在说明源代码的过程中，文档常常犯的第二种错误是对系统的理想化表述。有时，文档在描述系统时，并非按照已完成的实现，而是按照系统应该的样子或将来的实现。造成这种差异的本意常常是可敬的：文档的作者可能基于功能性规格说明来编写最终用户文档，并真诚地希望系统能够按照规定来实现。其他情况下，当结构性的更改在软件发布之前仓促实现时，构架性的设计文档没有随之更新。所有的情况中，传递给您——代码阅读者——的消息是，请降低您给予文档的信任级别。

4 其他文档来源

- 在寻找源代码文档时，要考虑到非传统的来源，比如注释、规范、出版物、测试用例、邮件列表、新闻组、修订日志、问题跟踪数据库、营销材料和源代码本身。在研究大量的代码时，人们一般会寻找更为正式的来源，比如需求和设计文档，常常会忽略嵌在注释中的文档。然而，源代码注释经常比相应的正式文档维护得更好，并且常常隐藏着珍贵的信息，有时甚至包括用ASCII码画成的图示。



源代码注释中的ASCII码图示

- * We wish to prove that the system's computation of decay
- * will always fulfill the equation:
- * $\text{decay}^{**} (5 * \text{loadavg}) \sim .1$
- *
- * If we compute b as:
- * $b = 2 * \text{loadavg}$
- * then
- * $\text{decay} = b / (b + 1)$
- *
- * We now need to prove two things:
- * 1) Given factor $^{**} (5 * \text{loadavg}) \sim .1$, prove factor $== b/(b+1)$
- * 2) Given $b/(b+1) ^{**} \text{power} \sim .1$, prove power $== (5 * \text{loadavg})$
- *
- * Facts:
- * For x close to zero, $\exp(x) \sim 1 + x$, since
- * $\exp(x) = 0! + x^{**}1/1! + x^{**}2/2! + \dots$
- * therefore $\exp(-1/b) \sim 1 - (1/b) = (b-1)/b$.
- * For x close to zero, $\ln(1+x) \sim x$, since
- * $\ln(1+x) = x - x^{**}2/2 + x^{**}3/3 - \dots -1 < x < 1$
- * therefore $\ln(b/(b+1)) = \ln(1 - 1/(b+1)) \sim -1/(b+1)$.
- * $\ln(.1) \sim -2.30$
- *
- * Proof of (1):
- * Solve (factor) $^{**}(\text{power}) \sim .1$ given power $(5 * \text{loadavg})$:
- * solving for factor,
- * $\ln(\text{factor}) \sim (-2.30/5 * \text{loadavg})$, or
- * $\text{factor} \sim \exp(-1/((5/2.30) * \text{loadavg})) \sim \exp(-1/(2 * \text{loadavg})) =$
- * $\exp(-1/b) \sim (b-1)/b \sim b/(b+1)$. QED
- *
- * Proof of (2):
- * Solve (factor) $^{**}(\text{power}) \sim .1$ given factor $== (b/(b+1))$:
- * solving for power,
- * $\text{power} * \ln(b/(b+1)) \sim -2.30$, or
- * $\text{power} \sim 2.3 * (b + 1) = 4.6 * \text{loadavg} + 2.3 \sim 5 * \text{loadavg}$. QED

代码注释中的数学证明

5 常见的开放源码文档格式

- 文档一般有两种类型;二进制文件.它幻的生成和阅读那要使用专利产品..最常通到的文档格式如下:
 - **Troff**
 - **Texinfo**
 - **DocBook**
 - **Javadoc**
 - **Doxygen**

troff

- troff文档编排系统一般与传统的man或新的BSD mode宏一同使用.共同创建Uni x手册页。

Texinfo

textinfo的宏由TeX文档编排系统来处理.许多GNU计创下实现的项目都健用它来创建在线和打印文档。

javadoc

- javadoc应用程序能够处理用特殊标记的注释进行评注的Java源代码，生成多种格式的义档。javadoc的注释都以/**，开头，可以包含标有@符号的标记。嵌入在文本中的标记要括在花括号中。

Doxygen

- Doxygen文档系统针对的是以C++、Java, IDL和C编写的源代码。它可以从一系列适当标注的源代码文件, 生成在线文档(HTML格式)和离线参考手册(LaTeX格式)。它还可以生成RTF, Postscript,超链接PDF和Unix man手册页。
- 基于标记语言的文档一般与源代码集成得更为紧密, 这是因为这类文档常常使用修订控制系统进行维护, 能使用文本处理工具自动操纵和生成, 并且完全集成到产品的编译过程中。更为紧密的集成是将文档作为源码的一部分, 常用在Java源代码中的CWEB系统和javadoc注释即为如此。

mdoc format

Documentation in mdoc format.

.\" \$NetBSD: cut.1,v 1.7.2.1 1997/11/04 22:25:05 mellon Exp \$ <-- a

.Dd June 6, 1993 <-- b

.Dt CUT 1 <-- c

.Os <-- d

.Sh NAME <-- e

.Nm cut <-- f

.Nd select portions of each line of a file <-- g

[...]

<-- h

.Sh DESCRIPTION

The

.Nm

utility selects portions of each line (as specified by

.Ar list)

from each

.Ar file

(or the standard input by default), and writes them to the standard output.

Documentation in Texinfo format.

- @node Filesystems and Volumes, Volume Naming, Fundamentals, Overview
<-- a
- @comment node-name, next, previous, up <-- b
- @section Filesystems and Volumes <-- c
- <-- d
- @cindex Filesystem
- @cindex Volume
- @cindex Fileserver
- @cindex sublink
- <-- e
- @i{Amd}views the world as a set of filesevers, each containing one or
- more filesystems where each filesystem contains one or more
- @dfn{volumes}. Here the term @dfn{volume}is used to refer to a
- coherent set of files such as a user's home directory or a @TeX{}
- distribution. @refill

进入下一章节

3

代码阅读的方法与技巧

3.1

基本编程元素

3.2

C数据结构

3.3

控制流程

3.4

文档

命名规范与约定

1 文件的命名及组织

- 大多数规范都会说明文件应该如何命名，应该使用什么样的扩展名。例如，您也许没有意识到，C头文件惯常的.h后缀就是一种风格，没有任何语言或编译器规定必须要使用它。遵守文件名和后缀的约定能够优化对源代码的搜索。许多文件名和扩展名跨不同项目和开发平台通用；

通用文件名

File Name	Contents
README	Project overview
MANIFEST	List of project files with brief explanations
INSTALL	Installation instructions
LICENSE Or Copying	Licensing information
TODO	Wish list for future extensions
NEWS	Documentation on user-visible changes
ChangeLog Or Changes	Code change summary
configure	Platform configuration script
Makefile	Build specification
Makefile.SH	Shell script producing the above
config.h	Platform configuration definitions
config_h.SH	Shell script producing the above
version.h Or patchlevel.h	Project release version

常见的文件扩展名（1）

File Extension	Contents
<code>.digit</code> <code>.man</code>	Unix manual page source
<code>.encoding</code>	Text in a particular encoding (e.g., <code>.utf8</code>)
<code>.language-code</code>	Text in a particular language (e.g., <code>.de</code> for German)
<code>.a</code> , <code>.lib</code>	Library collection of object files
<code>.asm</code> , <code>.s</code>	Assembly language source (Microsoft DOS/Windows, Unix)
<code>.asp</code> , <code>.cgi</code> , <code>.jsp</code> , <code>.psp</code>	Web server-executed source
<code>.awk</code>	<code>awk</code> (language) source
<code>.bas</code> , <code>.frm</code>	Microsoft Visual Basic source
<code>.bat</code> , <code>.cmd</code> , <code>.com</code>	MS-DOS/NT, OS/2 <code>Rexx</code> , VMS commands
<code>.bmp</code> , <code>.xbm</code> , <code>.png</code>	Microsoft Windows, X Window System, portable bitmap file
<code>.c</code>	C source
<code>.C</code> , <code>.cc</code> , <code>.cpp</code> , <code>.cxx</code>	C++ source
<code>.cs</code>	C# source
<code>.class</code>	Java compiled file
<code>.csh</code> , <code>.sh</code>	Unix <code>csh</code> , <code>sh</code> (Bourne) shell source
<code>.def</code>	Microsoft Windows or OS/2 executable or shared library definition
<code>.dll</code> , <code>.so</code>	Shared object library (Microsoft Windows, Unix)

常见的文件扩展名（2）

<code>.dsp, .dsw, .vbp</code>	Microsoft Developer Studio project and workspace file
<code>.dvi</code>	Documentation in <i>troff</i> or <i>TeX</i> device-independent output
<code>.el</code>	<i>Emacs</i> Lisp source
<code>.eqn, .pic, .tbl</code>	Equations, pictures, tables to be typeset by <i>eqn</i> , <i>pic</i> , <i>tbl</i>
<code>.gz, .Z, .bz2</code>	File compressed with <i>gzip</i> , <i>compress</i> , <i>bzip2</i>
<code>.h</code>	C or C++ header file
<code>.hpp</code>	C++ header file
<code>.ico, .icon</code>	Microsoft Windows, X Window System icon
<code>.idl</code>	Interface definition file
<code>.info</code>	Documentation generated by GNU <i>Texinfo</i>
<code>.jar, .shar, .tar</code>	File collection in Java, Unix shell, tape archive format
<code>.java</code>	Java source code
<code>.jcl</code>	IBM JCL instructions
<code>.l</code>	<i>lex</i> (lexical analyzer generator) source
<code>.m4</code>	<i>m4</i> (macro processor) code
<code>.mak, .mk</code>	Makefile (also often without any extension)
<code>.mif</code>	Documentation exported by FrameMaker
<code>.mm, .me</code>	Documentation using <i>troff</i> <i>mm</i> , <i>me</i> macros
<code>.nr, .roff</code>	Documentation in plain <i>troff</i> format

常见的文件扩展名（3）

<code>.o, .obj</code>	Object file
<code>.ok</code>	Local additions to the spell-check dictionary
<code>.pl, .pm, .pod</code>	Perl, module source, documentation
<code>.ps</code>	Postscript source, or formatted documentation
<code>.py</code>	Python source
<code>.rb</code>	Ruby source
<code>.rc, .res</code>	Microsoft Windows resource, compiled resource script
<code>.sed</code>	<i>sed</i> (stream editor) source
<code>.t, .test</code>	Test file
<code>.tcl</code>	Tcl/Tk source
<code>.tex, .texi</code>	Documentation in <i>TeX</i> or <i>LaTeX</i> , GNU <i>Texinfo</i> format
<code>.y</code>	<i>yacc</i> (parser generator) source

2 缩进（Indentation）

- 现代块结构语言编写的程序，都使用缩进来强调每个块的嵌套层次。风格指南经常规定用来编排代码块的空格的数量和类型。
- Java代码约定规定使用4个空格，同时，允许使用8个字符宽度的制表符（tab）。而BSD风格指南规定使用8个字符宽度的制表符。是否允许使用制表符，以及它们的精确宽度十分重要。缩排代码块有两种方式：使用若干空格符或一个制表符。
- 阅读代码时，首先要确保您的编辑器和打印机的tab设置，与代码遵循的风格规范一致。如果代码不能正确排列，则表明设置存在错误。如果找不到正在分析的项目的风格指南，就试用一些常见的设置。另外，源文件有时会在注释中包括指示编辑器应该如何安排它们的编排设置。

```
for (int i = 0; i < tTable.size(); i++) {  
    Table table = (Table) tTable.elementAt(i);  
    [...]  
    for (int j = 0; j < columns; j++) {  
        Object o[] = t.getNewRow();  
        [...]  
  
        if (table.getColumnIsNullable(j)) {  
            nullable = DatabaseMetaData.columnNullable;  
        } else {  
            nullable = DatabaseMetaData.columnNoNulls;  
        }  
        [...]  
        t.insert(o, null);  
    }  
}  
[...]  
o[2] = new Integer(0);    // precision  
[...]  
o[9] = new Boolean(false);    // unsigned
```

3 编排

- 所有的代码规范都会规定声明以及每个具体语句的编排方式。
- 规格说明中包括空格和换行符(line break)的分布。对于花括号的放置，存在两个不同的学派。许多Windows程序倾向于将花括号放在单独的行中，经常还要缩排。
- BSD和Java程序经常在语句的同一行打开花括号，使用和语句相同的缩进级别关闭花括号。
- 只要一致地使用，两种变体都十分易读。BSD/Java变体的优点之一是可以少占用一些垂直空间，从而使代码块更容易放入单个页面中。存在问题的明显信号是代码的编排不一致。

关于花括号{}

```
if (ia->ia_maddr != MADDRUNK)
{
    sc->pPacketPagePhys = ia->ia_maddr;
}
```

```
if ((cp = strchr(*argv, ':')) != NULL) {
    *cp++ = '\0';
    a_gid(cp);
}
```


- 请注意，=运算符之后使用的空格数量不同，函数名后使用的空格也不一致。对于编排不一致的代码，应该立即给予足够的警惕。如果编写该段代码的程序员没有经过正确编排的训练，那么极有可能存在大量其他更严重的错误。对不一致性的其他解释也都指向问题点.该代码可能是集成不同代码库中的元件的结果。同样，如果集成者不协调代码的编排(在多数情况下，使用代码编排工具，如indent, 就能够轻易地修复)，那么他们就有可能也没有处理更为重要的集成问题。不一致性也可能是多个程序员在同一段代码上工作的结果。在这种情况下，编排上的冲突反映出团队的协调很差，或者许多程序员不尊重前辈工作。作为这类软件的使用者要当心。
- 风格指南还规定了注释的编排和内容。

- 最后，编码规范还规定了一些特殊的注释单词，使得所有的实现人员都能够容易地搜索和确认。标识符XXX传统上用于标志虽然不正确、但(大多数时候)能够工作的代码。

```
/* XXX is this correct? */  
v_evaluate(vq, s, FALSE);
```

```
switch (*regparse++) {  
/* FIXME: these chars only have meaning  
at beg/end of pat? */  
case '^':
```

```
/*  
 * TODO - sort output  
 */  
int  
aliascmd(argc, argv)
```

4 命名约定

- 大多数编码规范中，一个重要部分就是关于标识符的命名。
- 标识符名称的指导原则既有简洁明了型(“选择不会造成误解的变量名”(仅仅一句话，够简洁。))，也有详尽复杂型(后面您将看到匈牙利记法)。
- 变量、类和函数的标识符大致有3种不同的构造方式。

标识符大致有3种不同的构造方式

- (1)首字母大写
 - 标识符中每个新单词的首字母都大写。两个具有代表性的例子是Windows API函数SetErrorMode和Java类RandRandomAccess。多数Java和windows编码约定都推荐首字母大写的方法，首字母是否大写要依附加的规则而定。
- (2)用下划线分隔单词
 - 标识符中每个附加单词都通过一个下划线与前面的单词分隔开来(例如，exponent_is_negatives。这种方法在受Unix影响的代码中很流行，并且也是GNU编码规范的推荐方法。
- (3)只取首字母
 - 这个方案中，取每个单词的首字母合并组成新的标识符。例如，spib声的意思是“set processor (interrupt) level(for) buffered input output”(设置缓冲输入输出的处理器(中断) 级别)。

如何命名常量

- 所有这3种编码规范在如何命名常量上都是一致的:常量使用大写字母命名，单词用下划线分隔。
- 请记住，这种大写命名约定可能和宏互相冲突，因为宏也遵循相同的命令方案。

- 在遵循Java编码规范的程序中，包名 (package name)总是从一个顶级的域名开始（例如，org.com. sun. ），类名和接口名由大写字母开始，方法和变量名由小写字母开始。这个约定使得您能够快速区分出类的引用(静态)和实例变量和方法。

匈牙利命名记法(Hungarian naming notation)

- windows世界的程序中，经常可以看到难以发音的标识符名，看起来好像是应用某种复杂的方案构成的。实际上，我们可以对标识符名进行解码，从中获得有用的信息。例如，变量名cwsz可能用来表示以。结尾的字符串中单词的个数。这种编码方案称为匈牙利命名记法(Hungarian naming notation)，它是以其开发者，Charles Simnnyi 甲，的祖国命名的。
- 开发这种记法的前提是合理的:标识符名应该易于被程序员记住，并对阅读代码的人有提示价值，类似标识符的名称应该一致，并能够快速判定。
- 遗憾的是，这种记法时常被误用，将与具体实现相关的信息嵌入到标识符名中，而这些信息本来应该对标识符是不透明的。

- 匈牙利记法并非仅仅用于C和C++标识符。用Visual Basic编写的程序，不管是独立的版本，还是Visual Basic for Application,都经常用匈牙利记法来命名标识符。
- 在阅读GUI应用程序中处理用户界面的代码时，窗体中所有用户界面控件的操作经常放在同一个代码模块中。在中等复杂的应用程序中，会有数十个全局控件在该代码中都可见。用户界面控件名称之前的匈牙利记法的前缀类型标记可以帮助我们确定它的作用。

5 编程实践

- 许多编码指导原则都对如何编写可移植的软件做了明确的规定。
- 不同的编程规范对可移植构造的构成有不同的主张。对那些主要关注的并非是最大可能的可移植性的工作来说，尤为如此。
 - 例如，GNU和Microsoft编码指导原则都将可移植性的焦点放在处理器构架的不同上，并且认为特定的软件只会在GNU, Linux或Microsoft操作系统上运行。
- 这么狭窄的可移植性定义，无疑是来源于实用的考虑，或许还有附加的暗示：GNU编码规范明确地表明，GNLT项目不支持Ifi位构架。
- 在审查代码的可移植性，或以某种给定的编码规范作为指南时，要注意了解规范对可移植性需求的界定与限制。
- 编码指导原则还经常规定应用程序应该如何编码以保证正确或一致的用户界面。

提供命令行接口的程序应该相应地使用`getopt`和`getopt_long`来解析程序的选项。请考虑下面这段解析命令行参数的代码

```
while (argc > 1 && argv[1][0] == '-') {  
    switch(argv[1][1]) {  
        [...]  
        case 's':  
            sflag = 1;  
            break;  
        default:  
            usage();  
        }  
        argc--; argv++;  
    }  
    nfiles = argc - 1;
```

```
while ((ch = getopt(argc, argv,  
"mo:ps:tx")) != -1)  
    switch(ch) {  
        [...]  
        case 'x':  
            if (!domd5)  
                requiremd5("-x");  
            MDTestSuite();  
            nomd5stdin = 1;  
            break;  
        case '?':  
        default:  
            usage();  
        }  
    argc -= optind;  
    argv += optind;
```

6 过程规范

- 软件系统并非只是代码。许多编码指导原则都延伸到开发过程的其他领域，包括文档、生成和发布过程的组织。
- 许多指导原则都会规定标准的文档，以及编写它们的格式。由于最终用户文档经常与应用程序或发布过程紧密地绑定在一起，故而常常组织的最好。
 - Microsoft Windows应用程序一般都会包括一个帮助文件，GNU项目提供Texinfo手册，传统的Unix应用程序提供标准化的手册页。

6 过程规范

- 编译机制的特殊性变得比较困难。但是，许多指导原则都为如何组织编译过程建立了精确的规则。
- 这些规则经常是基于特定的工具或标准化的宏;熟悉了它们之后，您就能够很快地理解，遵循给定指导原则的任何项目的编译过程。
- 有时会对发布过程做出精确的规定，一般是为了满足应用程序安装过程的需求。
 - 准化的软件发行格式，比如Microsoft Windows installer使用的格式，或Red Hat Linux RPM格式，对组成发行包的文件类型、版本控制、安装目录和由用户控制的选项都有严格的规则。

本章小结

- 1 认识代码阅读.....
- 2 代码阅读的工具.....
- 3 代码阅读的方法与技巧.....
- 4 大型项目代码阅读.....
- 5 课程总结 分享感言.....

4

大型项目代码阅读

4.1

项目文档

4.2

命名规范与约定

4.3

应对大型项目

- 大型的多个文件项目与小型项目之间的不同，并非仅仅在于分析它们的代码时，您会遇到更多挑战，还在于它们提供了许多理解它们的机会。
- 在本章中，我们将回顾一些用在实现大型项目中的常用技术，之后，分析这类项目开发过程中具体的构成部分。
- 我们将描述大型项目的组织方式，它们的编译和配置过程，不同文件版本如何控制，项目专用工具的特殊角色，以及典型的测试策略。
- 我们简略地给出这些内容常见的典型设置，并提供一些提示，帮助您了解如何使用它们来增强您的导航和理解能力。

4

大型项目代码阅读




项目文档

4.2



命名规范与约定

4.3



应对大型项目

项目文档

- 了解文档结构
- 注意浏览与查阅方法使用
- 在浏览大型的项目时，要注意，项目的源代码远不只是编译后可以获得可执行程序的语言指令；一个项目的源码树一般还包括规格说明、最终用户和开发人员文档、测试脚本、多媒体资源、编译工具、例子、本地化文件、修订历史、安装过程和许可信息。

4

大型项目代码阅读

4.1

项目文档

命名规范与约定

4.3

应对大型项目

分析项目中的命名规范与约定

- 粗略的浏览代码
- 分析总结出代码的风格，命名规范等

4

大型项目代码阅读

4.1

项目文档

4.2

命名规范与约定

应对大型项目

1 设计与实现技术

- 大型的编码工作，由于它们的大小与范围，经常能够证明应用一些技术的必要性，而在其他情况下这些技术可能根本不值得使用。本书中多处论及了许多这类技术;但是，为了便于您的阅读，此处还是概括了常见的一些设计与实现方法，并指出了对它们进行详细论述的章节。

(1)可视化软件过程和实用准则

- 在大型项目中，软件生命周期的各个组成部分常常被划分成更小的任务进行处理，它们经常是软件代码库的一个组成部分。
- 另外，重大项目所固有的复杂性，以及大量开发人员的参与，都要求必需采用某种正式的准则来组织开发工作。
- 这些准则一般会规定如何使用特定的语言、项目各组成部分的组织方式、应该记录一些什么内容、以及项目生命周期中大部分活动的过程。

(2) 重要的构架

- 尽管小型的项目有时通过简单地堆砌代码块，直至满足要求的规格说明为止，也能够侥幸成功，但是，大规模的软件开发工作必须使用一种合适的构架来构造所要创建的系統，控制其复杂性。
- 这个构架一般指定系统的结构，控制的处理方式，以及如何对系統各个组成部分进行模块分解。
- 大型的系統还可能会从框架(framework)、设计模式(design pattern)和特定领域的构架中极取灵感，重用它们构架上的思想。

(3) 积极的分解

- 大型项目中，在实现层，经常使用诸如函数、对象、抽象数据类型和组件等机制，对系统的组成元素进行积极的分解。

(4) 多平台支持

- 大型的应用程序经常能够吸引相当广泛的用户团体。
- 这类应用程序经常需要处理大量的可移植问题，而在那些不怎么受人关注的项目中，常常可以不考虑这些问题。

(5)面向对象技术

- 大型系统的复杂性经常可以使用面向对象的设计和实现技术加以控制。
- 一般将对象组织成为不同的层次。
- 之后，就可以使用继承来提取出公共行为，动态调度技术使得单一代码体能够处理众多不同的对象。

(6) 运算符重载

- C++和Haskell等语言编写的实用代码集合中，常常会使用运算符重载，将项目专有的数据类型提升为“一等公民”，然后就可以使用语言的内建运算符集对它们进行操作。

(7) 库、组件和进程

- 在更大的粒度上，大型系统的代码经常分解为对象模块库，可重用组件，甚至独立的进程。

(8) 领域专用和定制的语言和工具

- 大规模的编码工作经常涉及专门工具的创建，或者从相似用途的现有工具中受益。

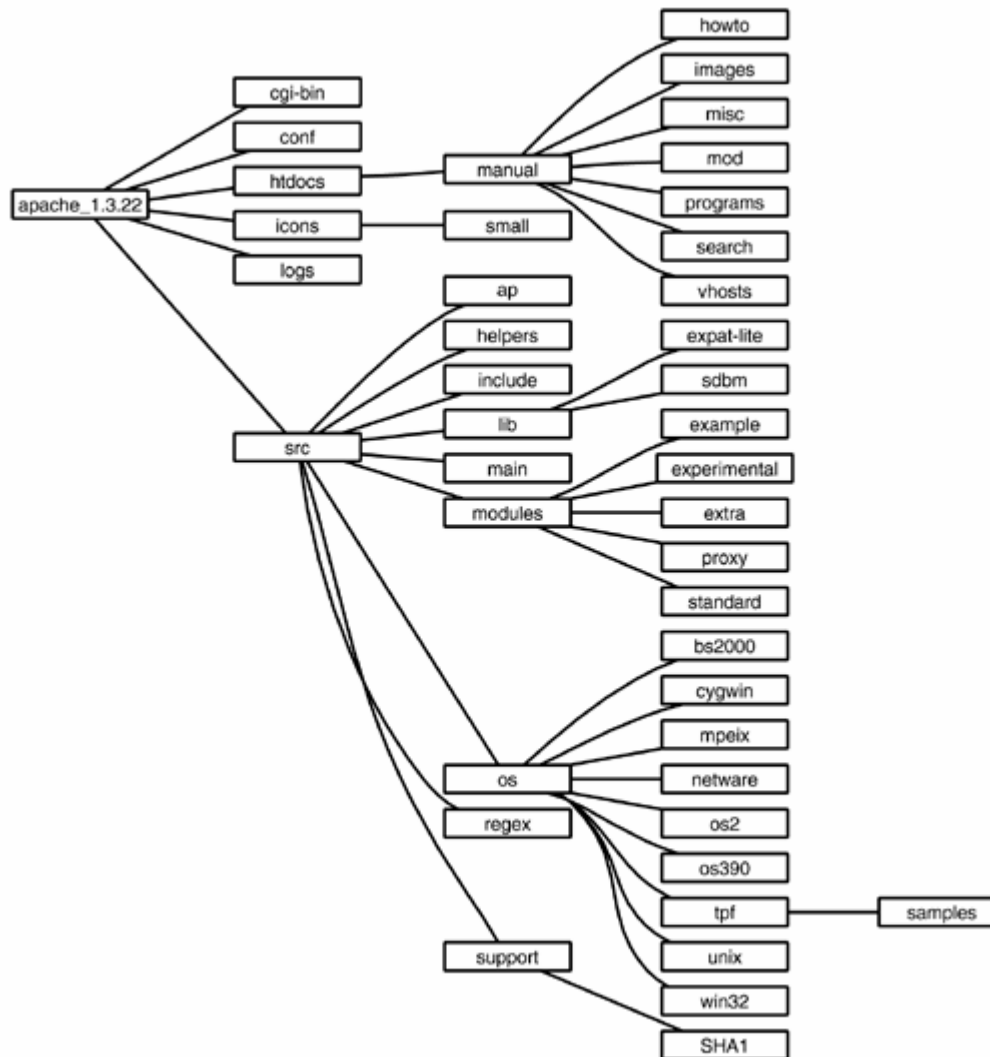
(9) 对预处理的积极使用

-
- 用汇编、C和C++语言实现的项目经常使用预处理器，用领域专有的结构对语言进行扩展。



2 项目的组织

- 我们可以通过浏览项目的源代码树——包含项目源代码的层次目录结构，来分析一个项目的组织方式。源码树常常能够反映出项目在构架和软件过程上的结构。



2 项目的组织

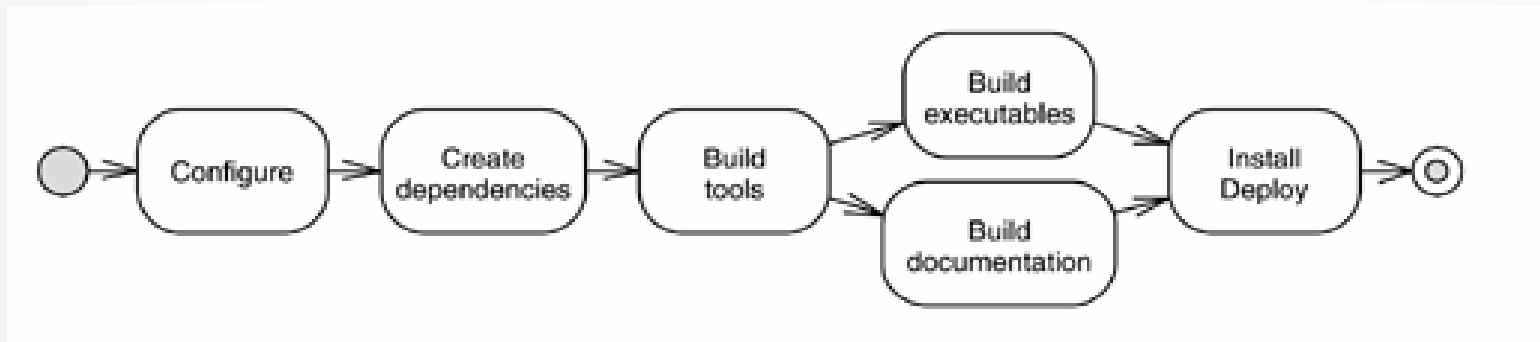
- 当您首次接触一个大型项目时、要花一些时间来熟悉项目的目录树结构。
- 许多人喜欢使用图形化的工具来完成这项任务，比如windows资源管理器，或GNU Midnight Commander。
- 如果这两种工具都不能使用，您可以使用Web浏览器打开本地源码目录，对目录结构进行图形化浏览。
- 在大多数Unix系统上，您还可以使用locate命令找出所寻找的文件，在windows平台上，资源管理器的文件搜索机制可以完成类似的任务。

3 编译过程和制作文件

- 大多数大型的项目使用一个复杂的编译过程。这类过程一般能够处理配置选项、多种类型的输入输出文件、错综复杂的相互依赖和多个编译目标。由于编译过程最终会影响生成的输出，所以能够‘阅读’项目的编译过程和阅读项目的代码同样重要。
- 然而，对于如何描述与执行编译过程，没有标准化的方式。每个大型项目和开发平台都使用自己专有的方式来组织编译工作。
- 大多数编译过程中，一些元素是公共的。

一个典型编译过程所涉及的各个步骤

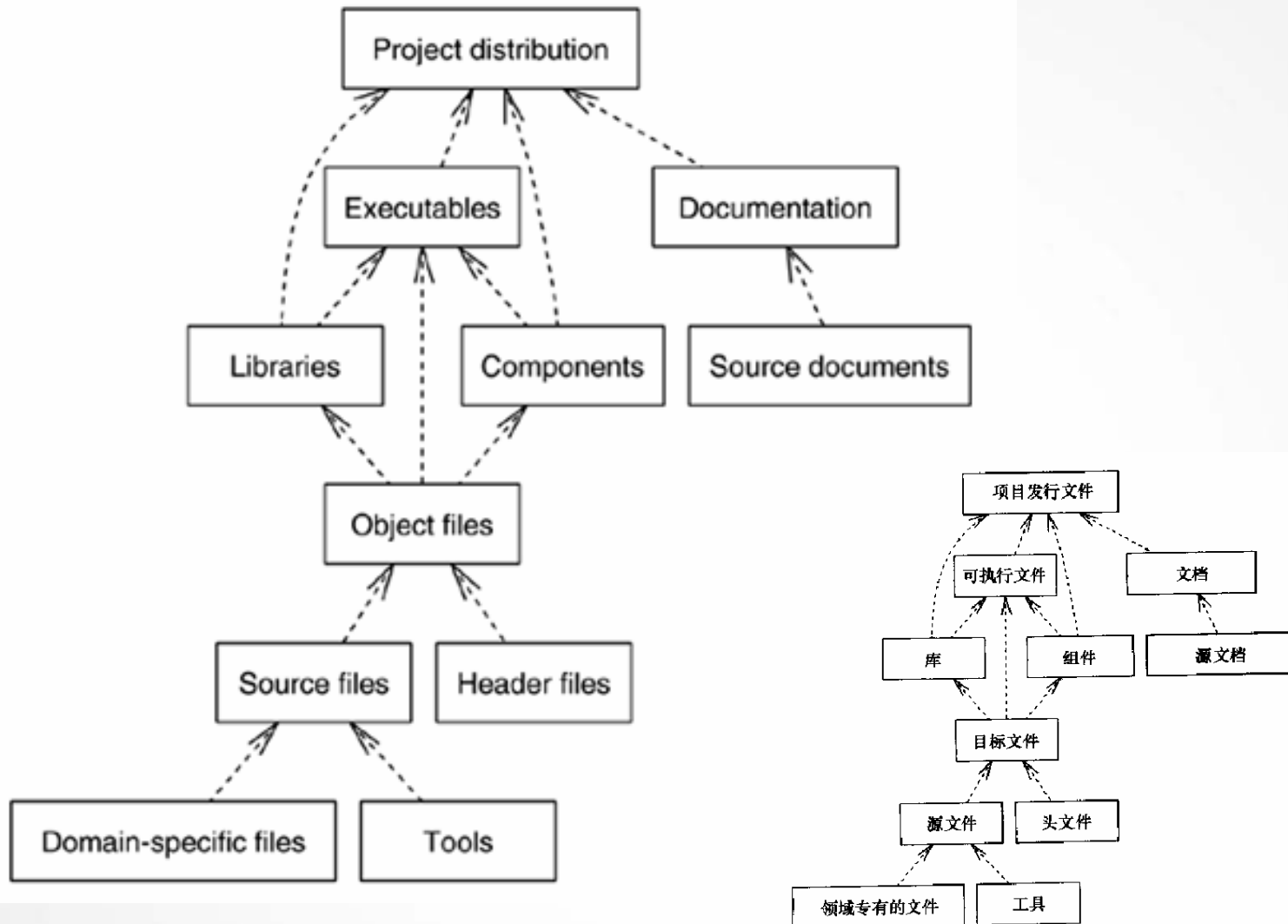
- 第一步是配置软件的选项，并精确地确定所要编译的内容。根据项目的配置，我们可以创建一个依赖关系图(dependency graph)。
- 大型项目一般涉及数万个不同的组件;其中的大部分组件都与其他组件有关。
- 依赖关系图说明各个项目组件的正确编译次序。编译过程中的某些部分常常不是使用标准的开发工具，比如编译器或链接器，来完成，而是需要使用专为特定项目开发的工具。
- 一个典型的例子是irake工具，可以用它来标准化跨} window系统的编译过程。项目专用的工具编译后，就能够与其他标准工具一同使用，预处理、编译和链接项目的可执行文件。同时，项目的文档经常同步地从‘， 源码”格式转换为最终的发行格式。
- 最后是将生成的执行文件和文档安装到目标系统，或准备用于大规模的部署或发行。典型的发行方法包括Red Hat Linux RPM文件， Windows Installer格式， 或者将适当的文件上传到web站点。



项目依赖关系的定义与管理

- 编译过程中最为错综复杂的部分是项目依赖关系的定义与管理。
- 这些关系说明项目中不同部分之间的相互依赖关系，以及项目中各个部分进行编译的次序。
- 我们在图中列举了项目中一组典型的依赖关系。这个项目的发行文件依赖于执行文件和文档文件的存在；有时，项目专有的动态载入库和软件组件也常常是最终发生文件的一部分。
- 可执行文件依赖于目标文件、库和组件，因为可执行文件就是这些文件链接到一起产生的。库和组件也依赖于目标文件。目标文件依赖于相应源文件的编译，在使用G和G++等语言的情况下，还依赖于源文件中包括的头文件。
- 最后，一些源文件是从领域专有的文件中自动产生的(例如，Yacc语法文件经常用于创建一个用G语言编写的分析器)；因此，源文件依赖于领域专有的代码和相应的工具。
- 在我们的描述中，可以看到依赖关系与编译过程是如此紧密地链接在一起。在描述完具体的依赖关系图后，我们将讨论如何利用这种关系引导编译过程。

一组典型的项目依赖关系



Make 和makefile

- 在使用诸如make之类的工具时，由文件的各个构成部分构造文件需要遵循的依赖关系与规则在一个单独的文件中指定，一般命名为makefile或Makefile。
- make读取并处理该制作文件，之后发布适当的命令编译指定的目标。
- 制作文件中绝大部分内容是一个个的项，每个项描述目标、目标的依赖关系和从这些依赖关系生成目标的规则。


```
OBJS= modules.o $(MODULES) \  
main/libmain.a $(OSDIR)/libos.a \  
ap/libap.a  
...  
.SUFFIXES: .def  
.def.a:  
    emximp -o $@ $<  
[...]  
target_static: subdirs modules.o  
    <-- a  
    $(CC) -c $(INCLUDES) $(CFLAGS) buildmark.c  
    $(CC) $(CFLAGS) $(LDFLAGS) $(LDFLAGS_SHLIB_EXPORT) \  
        -o $(TARGET) buildmark.o $(OBJS) $(REGLIB) $(EXPATLIB) $(LIBS)  
[...]  
clean:  
    -rm -f $(TARGET) lib$(TARGET).* *.o    <-- b  
  
$(OBJS): Makefile subdirs  
  
# DO NOT REMOVE  
buildmark.o: buildmark.c include/ap_config.h include/ap_mmn.h \  
include/ap_config_auto.h os/unix/os.h include/ap_ctype.h \  
[...]
```

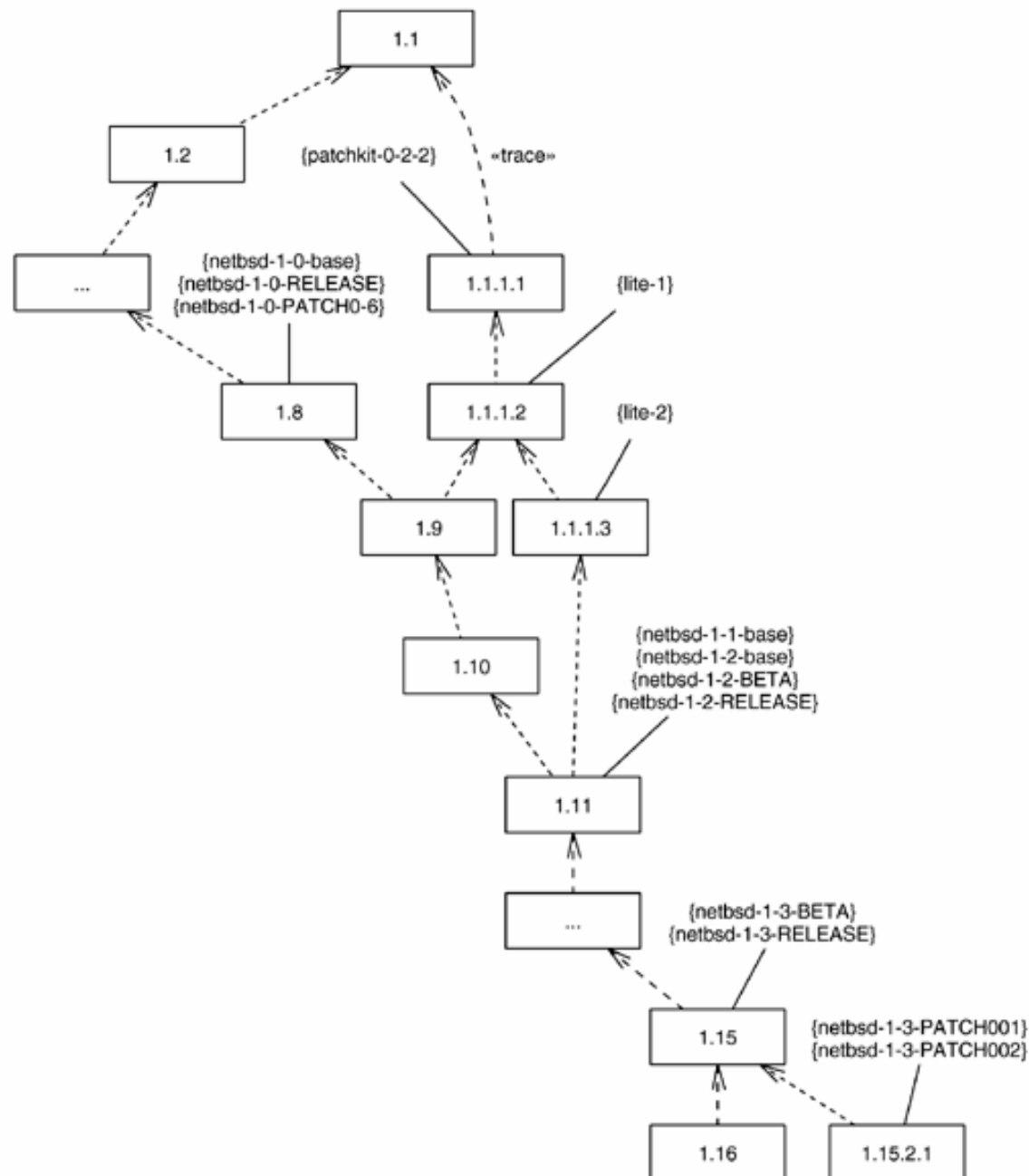
- 随着制作文件越来越大，它们变得难以阅读和推理。有一种方便的快捷方式，可以用来解开一个复杂的编译过程。可以使用-n开关预演make,检查大型编译过程中的各个步骤。在进行预演时，make将显示它要执行的命令，但不会真的执行它们。
- 从而，可以将make的输出重定向到一个文件中，在您喜欢的编辑器中打开该文件仔细地检查它。
- 在编译清单中查找字符串(代表性的工具、命令行参数、输入输出文件)的能力特别有用。
- 制作文件很少能在不同的操作系统间进行移植，因为它们经常包平台相关的编译命令。

4 配置(Configuration)

- 配置（configuration）可以控制的软件系统，允许开发者编译、维护和发展源代码的单一正式版本.只维护源代码的单一副本简单了更改和演化管理。通过使用适当的配置，相同的源代码体可以：
 - 创建拥有不同特性的产品；
 - 为不同的构架或报作条统构造产品；
 - 在不同的并发环峨下进行维护；
 - 与不同的库链接；
 - 快用运行拼间指定的配置选项来运行.

5 修订控制(Revision Control)

- 我们可以将系统的源代码想像成在空间和时间两个方向上延伸。
- 代码，组织成文件和目录的形式，占据空间，同时，同一代码还随着时间的推移不断演化。
- 修订控制系统可以跟踪代码的演化，标记重大的事件，并记录更改背后的原原由，允许我们查看和控制时间要素。



- 在深入研究如何应对修订控制系统控制下的项目之前，最好先了解一下，在典型的开发过程中如何跟踪修订的概况。
- 修订控制系统将与文件相关的所有历史信息保存到一个中心储存库中。
- 每个文件的历史都保存在一系列可辨别的版本中，一般使用自动生成的版本号.还经常使用开发人员为标记系统开发过程中某个里程碑面选定的关联标记。
- 在项目发布之前，版本的里程碑用来标记开发的目标

- 例如，提供指进性的更改和bug的修复。当开发人员在文件的某个版本上完成工作后，他或她会将文件交付(commit)或检入(check in)存储库。
- 修订控制系统保存了文件所有版本的全部细节(大多数系统通过存储版本之间的差异来节省空间)，同样还有每次交付操作的元数据(一般是日期、时间、开发人员的姓名和对更改的注释)。
- 某些开发模型允许开发人员在使用文件时将它锁定(lock)，从而禁止同时更改。其他的系统则提供一种协调方式，当冲突发生时，可以协调互相冲突的更改。有时，针对给定文件的开发工作可能被分为两个不同的分支(branch)(例如，一个稳定分支和一个维护分支);后期，两个分支可能会再次结合。
- 有些系统由大量的文件组成，为了协调整个系统的发布工作，许多系统提供一种标记方式，用一个标识系统版本的符号名称，标识属于该系统的一系列文件。
- 从而，我们能够确认组成完整系统的每个文件的版本。

修订控制系统的文件标记

RCS and CVS

<code>\$Author\$</code>	Login name of the user who checked in the revision	检入修订文件的用户的登录名
<code>\$Date\$</code>	Date and time the revision was checked in	修订文件检入的日期和时间
<code>\$Header\$</code>	Path name, revision, date, time, author, state, locker	路径名、修订文件、日期、时间、作者、状态、
<code>\$Id\$</code>	Same as <code>\$Header\$</code> ; file name without a path	同 <code>\$Header\$</code> ; 只给出文件名, 不带路径
<code>\$Locker\$</code>	Login name of the user who locked the revision	锁定修订文件的用户的登录名
<code>\$Log\$</code>	Messages supplied at check-in, preceded by a header	检入时提供的信息, 前面有一个标题
<code>\$Name\$</code>	Symbolic name used to check out the revision	用来检出修订文件的符号名称
<code>\$RCSfile\$</code>	Name of the RCS file without a path	不带路径的 RCS 文件名
<code>\$Revision\$</code>	Revision number assigned to the revision	赋予修订文件的修订号
<code>\$Source\$</code>	Full path name of the RCS file	RCS 文件的全路径名
<code>\$State\$</code>	State assigned to the revision	修订文件所处的状态

修订控制系统的文件标记

SCCS

<code>%M%</code>	Module name
<code>%R%</code>	Release
<code>%L%</code>	Level
<code>%B%</code>	Branch
<code>%S%</code>	Sequence
<code>%I%</code>	Release, level, branch, and sequence
<code>%D%</code> / <code>%H%</code>	Date of the current get
<code>%T%</code>	Time of the current get
<code>%E%</code> / <code>%G%</code>	Date the newest applied delta was created
<code>%U%</code>	Time the newest applied delta was created
<code>%F%</code> / <code>%P%</code>	SCCS file/full path name
<code>%Y%</code>	Module type
<code>%C%</code>	Current line number in the file
<code>%Z%</code>	The four-character string @(#) used by <i>what</i>
<code>%W%</code>	Shorthand for <code>%Z%%M%tab%i%</code>
<code>%A%</code>	Shorthand for <code>%Z%%Y% %M% %I%%Z%</code>

修订控制系统的文件标记

Visual Source Safe (VSS)

<code>\$Archive:\$</code>	VSS archive file location
<code>\$Author:\$</code>	User who last changed the file
<code>\$Date:\$</code>	Date and time of the last check-in
<code>\$Header:\$</code>	Shorthand for <code>Logfile</code> , <code>Revision</code> , <code>Date</code> , <code>Author</code>
<code>\$History:\$</code>	File history, in VSS format
<code>\$JustDate:\$</code>	Date, without the time addendum
<code>\$Log:\$</code>	File history, in RCS format
<code>\$Logfile:\$</code>	Same as <code>\$Archive:\$</code>
<code>\$Modtime:\$</code>	Date and time of the last modification
<code>\$NoKeywords:\$</code>	No keyword expansion for keywords that follow
<code>\$Revision:\$</code>	VSS version number
<code>\$Workfile:\$</code>	File name

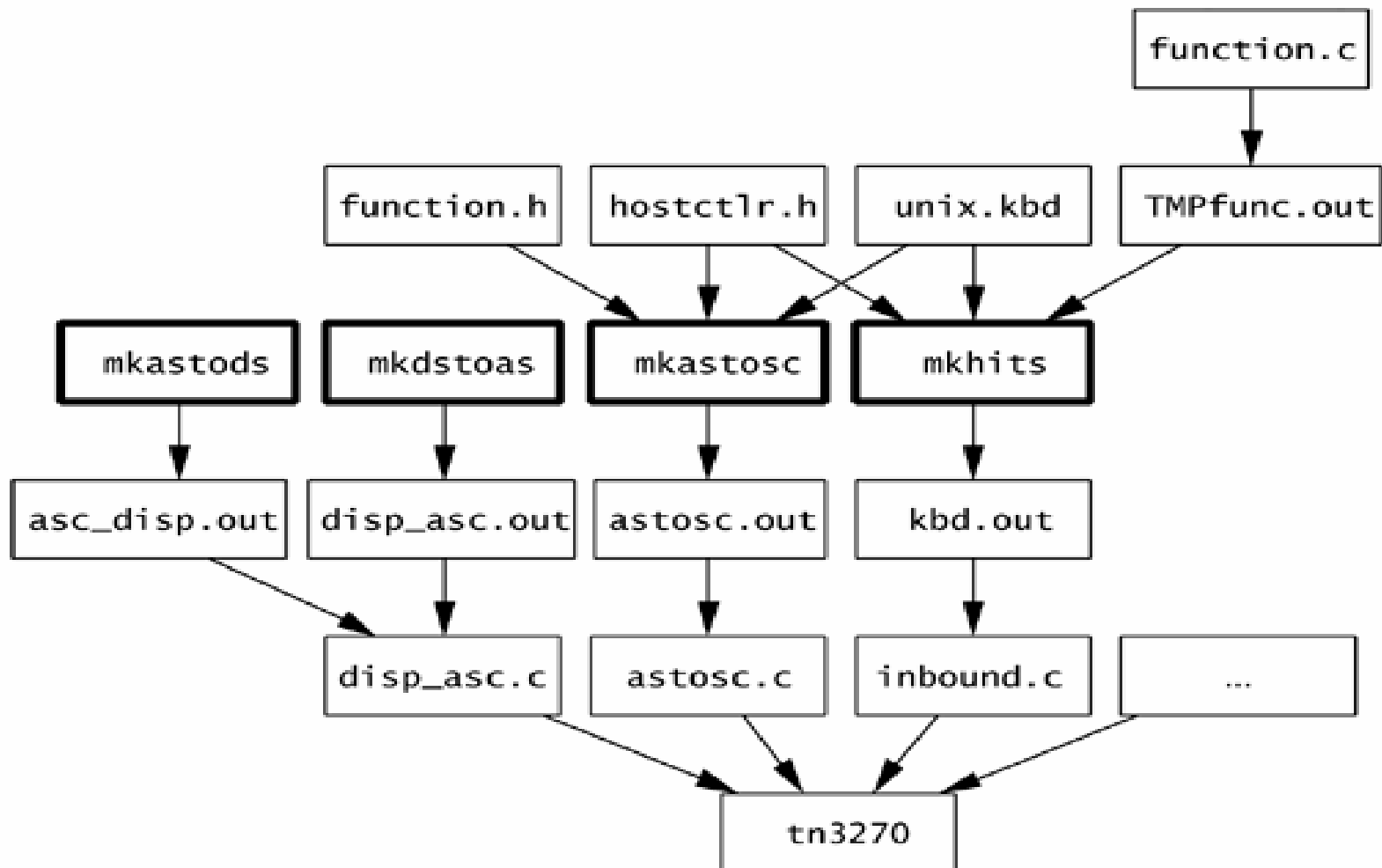
6 项目的专有工具(Project-Specific Tools)

- 大型项目经常拥有独特的问题，并且拥有足够的资源构造专门的工具。
- 定制编译工具用在软件开发过程的许多方面，包括配置、编译过程管理、代码的生成、测试和文档编制。
- 在下面的段落中，我们将针对每项任务，提供一些具有代表性的工具。

项目专用工具

- 到目前为止，项目专用工具的最通常的用途是生成特定的代码。因此，用工具动态地创建软件代码是编译过程的一部分。大多数情况下，这些代码使用的语言与系统的其他部分相同(例如，c)，并且大都比较简单，一般由查找表或简单的switch语句组成。使用这些工具一般是为了在编译期间构造代码，以避免在运行期间执行等同的操作所带来的开销。这种方式下生成的代码常常更高效，并且消耗较少的内存。代码生成工具的输入可以是另外的文件(以文本形式表达所要生成代码的规格说明)，简单的领域专用语言，甚至系统源代码的组成部分。

用在生成IBM 327终端仿真器中的工具



配置文件

```
struct hits hits[] = {  
    { 0, { {undefined}, {undefined}, {undefined},  
    {undefined}  
    } },  
    [...]  
    { 70, { /* 0x05 */  
        { FCN_ATTEN },  
        { undefined },  
        { FCN_AID, AID_TREQ },  
        { undefined },  
    } },  
    { 65, { /* 0x06 */  
        { FCN_AID, AID_CLEAR },  
        { undefined },  
        { FCN_TEST },  
        { undefined },  
    } },  
    } },
```

7 测试

- 设计良好的项目，都会预先为测试系统的全部或部分功能提供相应的措施。
- 这些措施可能隶属于一份经过深思熟虑，用来验证系统运作的计划，也可能是系统的开发者在实现系统的过程中实施的非正式测试活动的残余。
- 作为源代码阅读活动的一部分，我们首先应该能够识别并推理测试代码和测试用例，然后使用这些测试产物帮助理解其余的代码。

- 最简单的测试代码是一条生成日志(logging)或调试(debugging)输出的语句。开发人员一般用这种语句测试程序的运作是否与他们的预期相同。程序的调试输出可以帮助我们理解程序控制流程和数据元素的关键部分。此外，跟踪语句所在的地点一般也是算法运行的重要部分。大多数程序中，调试信息一般输入到程序的标准输出流或文件中。

例：DEBUG宏

```
#ifdef DEBUG
    if (trace == NULL)
        trace = fopen("bgtrace", "w");
    fprintf(trace, "\nRoll: %d %d%s\n", D0, D1, race ? " (race)" : "");
    fflush(trace);
#endif
```

- 请注意，在上面的代码中，如何用(DEBUG)宏来控制特定的代码是否编译到最终可执行文件中。系统的产品版本一般在编译时不会定义(DEBUG)宏，从而略去了跟踪代码和它生成的输出。

调试级别(debug level)

```
if (debug > 4)
    printf("systime: offset %s\n", lfptoa(now, 6));
```

- 太多的跟踪信息可能会掩盖至关重要的信息，或者会将程序的运行速度降到根本没办法使用的程度。
- 程序中常常定义一个称为调试级别(debug level)的整数，它用来过滤生成的消息。

断言(assertion)

- 更为主动的方案是，在程序代码中测试各种在程序执行时应该为true的条件。这些条件，称为断言(assertion)，基本上是一些语句，这些语句指定当程序以开发者预想的方式工作时期望得到的结果。
- 许多语言和库都能够在对程序结构影响最小的情况下，测试断言，并在断言失败时，自动生成调试输出(常常还会终止程序)。
- 可以用断言来检验算法运作的步骤、函数接收的参数、程序的控制流程、底层硬件的属性和测试用例的结果。C和C++库都使用assert进行断言。

断言举例

- 断言还经常用来检查函数接收的参数的值。

```
if (dp != NULL)
    break;
/* uh_oh... we couldn't find a subexpression-level match */
assert(g->backrefs); /* must be back references doing it */
assert(g->nplus == 0 || m->lastpos != NULL);
```

- 类似的断言有时还用来检验函数的返回值。

```
Void pmap_zero_page(register vm_offset_t phys)
{
    [...]
    assert(phys != vm_page_fictitious_addr);
```

测试框架JUnit

- Java程序常常使用KentBeck和Erich Gamma的JUnit框架，将断言组织成完整的测试用例。
- JUnit支持测试数据的初始化、测试用例的定义、测试套件的组织和测试结果的收集，它还提供一个执行测试用例的GUI工具TestRunner。

回归测试

- 更有效率的测试形式是将程序的结果与已知正确的输出进行测试。
- 回归测试—通常基于早期经过人工检验的运行结果，经常在程序的实现被修复后执行，以确保程序中的各个部分没有被偶然地破坏。
- 回归测试一般由一个执行测试用例的测试马具、每个测试用例的输入、期望的结果和顺序执行所有用例的框架组成。execve完成Unix系统调用的回归测试就是一个典型的例子。下面这个微小的驱动程序可以用来执行不同类型的程序，并打印出结果。

```
Int main(int argc, char *argv[])  
{ [...]  
    if (execve(argv[1], &argv[1], NULL) == -1) {  
        printf("%s\n", strerror(errno));  
        exit(1);  
    }  
}
```

在阅读源代码时，要记住：

- 测试用例可以部分地代替函数规格说明。
- 另外，可以使用测试用例的输入数据对源代码序列进行预演。

本章小结

- 1 认识代码阅读.....
- 2 代码阅读的工具.....
- 3 代码阅读的方法与技巧.....
- 4 大型项目代码阅读.....
- 5 课程总结 分享感言.....

5

课程总结 分享感言

5.1

课程总结

5.2

分享感言

5

课程总结 分享感言

5.1

课程总结

5.2

分享感言

- 1) 阅读features。以此来搞清楚该项目有哪些特性；
- 2) 思考。想想如果自己来做有这些features的项目该如何构架；
- 3) 下载并安装demo或sample。通过demo或sample直观地感受这个项目；
- 4) 搜集能得到的doc，尽快地掌握如何使用这个项目；
- 5) 如果有介绍项目架构的文档，通过它了解项目的总体架构，如果没有，通过api-doc了解源码包的结构；
- 6) 分两遍来阅读源码。第一遍以应用为线索，以总体结构为基础，阅读在应用中使用到的类和方法，但不用过深挖掘细节，对于嵌套调用，只用通过函数名了解最上层函数的意义，这一遍的目的在于把大致结构了然于心。第二遍就是阅读类和方法的实现细节，以第一遍的阅读为基础，带着疑问去阅读那些自己难以实现的模块。
- 7) 总结。回味这个项目设计上的精妙，用到了哪些设计模式，能在哪些领域可以借鉴等等。

5

课程总结 分享感言

5.1

课程总结

5.2

分享感言

分享感言



龙旗控股
Longcheer Holdings



谢 谢

www.longcheertel.com