

Asterisk 补丁/代码向导

李 峰 (lifeng@telebyte.com.cn)

2007年10月16日

目 录

1	前 言	3
2	总的原则	3
2.1	声明函数和变量	3
2.2	使用内部API	4
2.3	代码格式	4
2.3.1	If 语句	5
2.3.2	Case 语句	5
2.3.3	没有括号的无镶嵌语句	5
2.3.4	Labels/goto 是可接受的	6
2.3.5	不要用未初始化的变量	6
2.3.6	不要 cast 'void *'	6
3	变量命名	6
3.1	全局变量	6
3.2	不要用不必要的类型定义 typedef)	6
3.2.1	如可能, 尽量用枚举(enum)来替代#define	7
4	串处理	7
5	使用函数	7
5.1	用 strsep 来解析串	8
5.2	创建通用代码!	8
6	指针和指针分配的处理问题	8
6.1	复引用和局部指针	8
6.2	尽量在指针参数前用 const 修饰符	8
6.3	不要建立你自己的链接表代码—重用!	8
6.4	请避免不需要的指针分配工作!	9
6.5	对结构进行分配	9
7	CLI 命令	9
8	新的拨号计划(dialplan)应用/函数	10
9	Doxygen API 文档标准	10
10	在提交代码前结束代码	11
10.1	再次查看你的代码	11
10.2	读补丁	11
10.3	听取建议	11
10.4	优化, 优化, 再优化	11

1 前言

我们希望你能为 Asterisk (开源的PBX项目)做出自己的贡献。由于 Asterisk 是一个庞大的和时效性很强的应用项目, 所以其代码需要满足一定的规范以便于大多数的参与开发的程序员在开发新的功能或维护代码时可以读懂以前的代码。代码也需要检查和测试, 以便于代码能按照总体构架和指导原则来工作, 并且代码需要非常好的文档来支持。

Asterisk 被 Digium 公司以双许可方式发布。为了使你的代码被代码库接受, 所有的非细微的改变都被声明允许 Digium 公司或其它公共地方发布。如需更多的信息, 请访问 <http://bugs.digium.com>

补丁将统一到 diff 格式 (diff -u), 从 asterisk 源代码的顶级目录。例如:

```
~/work$ diff -urN asterisk-base asterisk-new
```

如果你从一个新的 SVN 工作拷贝的基础上做了修改, 你可以使用下面的命令来建立补丁文件:

```
$ svn diff <mycodefile>.c
```

2 总的原则

- 所有的代码, 文件名, 函数名和注释必需用英语来写。
- 不要用 ”/* JMG 4/20/04 */” 这样的方式来写你的注释; 注释应解释代码完成什么功能, 不是什么时间改变的或由谁来改变。如果你贡献了大量的代码, 请确保你添加了 CREDITS 文件。
- 不要在代码中做不必要的空白字符变动。如果你要做改变, 将改变提交到代码跟踪者或作为另一个版本的补丁来出现。这个补丁中包只包括空格和格式改变。
- 不要使用 C++ 体的注释 (//) 。
- 总是试图满足你工作的现有的文件格式。
- 如果分配行内注释, 用空格代替制表键, 或 #define (这使你的注释甚至在其它制表尺寸下也能正常的显示)

2.1 声明函数和变量

- 不要在代码块内声明变量(例如, 就向最近的GNU编译器所支持的一样)因为这种方式不利于阅读代码并且这种方式不兼容于GCC 2.95和别的C编译器。
- 不想用于模块外的函数和变量必须声明为 static 。
- 当用 scanf 函数(或变量)读整形数字时, 不要用 '%i' 格式, 除非你指定希望允许非十进制的数值的输入, '%d' 总是更好的选择, 由于它默认在后台将前导0变成八进制数。
- 从输入得到的串量不要用于 *printf 函数指针的第一个参数。

2.2 使用内部API

请确保你明了现存于Asterisk中的增强代码移植性和产生更为安全的线程安全代码字符串和数据处理函数, 请检查 `utils.c/utils.h` 来得到更详细的信息。

2.3 代码格式

总的来说, Asterisk的代码格式的指导原则如下:

```
# indent -i4 -ts4 -br -brs -cdw -cli0 -ce -nbfd -npcs -nprs -npsl -saf -sai -saw foo.c
```

下面是这些参数的详细解释:

- i4: 缩进级别为4
- ts4: 制表位宽为4个字符
- br: 在if行上有花括号
- cdw: 当有while语句时, 用一对括号包含
- cli0: case 语句缩进位为0
- ce: else语句用一对括号包含
- nbfd: 不要将函数的 decl 参数断行
- npcs: 在调用函数名后没有空格
- nprs: 在括号后没有空格
- npsl: 不中断过程类型
- saf: 在for关键词后产生空格
- sai: 在if关键词后产生空格
- saw: 在while关键词后产生空格

函数调用和参数用统一的空格分开,下面是代码的示例说明:

合理的情况: `foo(arg1, arg2);`
合理的情况: `foo(arg1, arg2); /* 可被接受, 但不是最合理的代码格式 */`
不合理的情况: `foo (arg1, arg2);`
不合理的情况: `foo(arg1, arg2);`
不合理的情况: `foo(arg1, arg2, arg3);`

不要对待关键词 (if, while, do, return) 象对函数一样;

对关键词和表达式之间要保留空格。对于‘return’, 甚至不要将表达式用括号括起来, 因为它们不需要。这些空白对于代码来说没有坏处:) 为了使代码更容易被阅读, 请按上面的规则使用空格, 例如:

```
for (str=foo;str;str=str->next)
```

相对于下面的语句来说, 更难于阅读。

```
for (str = foo; str; str = str->next)
```

下面是代码如何被格式化的示例。

```
Functions:  
int foo(int a, char *s)  
{  
return 0;  
}
```

2.3.1 If 语句

```
if (foo) {
    bar();
} else {
    blah();
}
```

2.3.2 Case 语句

```
switch (foo) {
case BAR:
    blah();
    break;
case OTHER:
    other();
    break;
}
```

2.3.3 没有括号的无镶嵌语句

如下面的示例代码:

```
for (x=0; x < 5; x++)
    if (foo)
if (bar)
    baz();
```

替换为:

```
for (x = 0; x < 5; x++) {
    if (foo) {
        if (bar)
    baz();
    }
}
```

不要象下面这样编译代码:

```
if (foo) {
    /* .... 50 lines of code ... */
} else {
    result = 0;
    return;
}
```

转而替换为最小化需要缩进的代码的行数, 这仅需通过缩进最小的条件语句‘if’来实现:

```
if !(foo) {
    result = 0;
    return;
}

.... 50 lines of code ....
```

当适当的使用这个技术，它使的函数更易读和更易维护，尤其是当这些有一个或两个对于函数必需成功的使函数执行的‘setup’操作时。

2.3.4 Lables/goto 是可接受的

正确的使用这个技术与需要的结果相关联的 Lable/goto 组合，以便于 error/failure 情况出现里能退出函数，这不是一件坏事！在这种情况下，我们鼓励使用 goto 语句，因为这时不用使用清除错误或失效情况的代码。

2.3.5 不要用未初始化的变量

请确保你不要使用未初始化的变量。如果你用了未初始化的变量，通常编译器会发出警告信息。但是不要按这种方式走的太远，并且不需要被初始化的变量是不需要这样的。如果第一次在函数中使用变量，这其实是存一个值于函数中，之后初始化这个变量是与指针无关的。并将在产生的二进制代码中产生特殊的目标码和数据。当你怀疑这个过程时，请相信编译器将诉你是否需要初始化变量。如果你没有收到警告信息，变量不需要被初始化。

2.3.6 不要 cast ‘void *’

不要将 cast ‘void *’ 转换到任何其它类型，或将任何其它类型转换为空指针‘void *’。暗含类型转换 从/源自 ‘void *’ 在 C 语言中经过特殊说明是明确允许的。这意味着下更函数的结果 malloc(), calloc(), alloca() 和相似的函数甚至不需要转换为指定的类型，并且当你传递一个指针到一个接受空指针的回呼函数，你不需要转换到其类型。

3 变量命名

3.1 全局变量

命名全局变量(或在一个很长的函数中命名局部变量时)你应当意识到100年后的外国人可能要读你的代码。所有的变量名都用小写，除了后面跟随外部 API 或规定正常情况用大写或混合大小写变量名；在这种情况下，为了便于理解最好在后面跟外部 API/分类说明。

对于打算做全局变量的变量，在命名时加以说明这个变量将用于全局，例如：

```
static char global_something[80]
```

3.2 不要用不必要的类型定义 typedef)

不要用 ‘typedef’ 仅为了表明类型的数量；这里没有实质的好处：

```
struct foo {
```

```
    int bar;
};
typedef foo_t struct foo;
```

实际, 根本不用 'variable type' 为后缀; 但更好的方法仅输入 'struct foo' 而不是 'foo_s'。

3.2.1 如可能, 尽量用枚举(enum)来替代#define

尽可能使用枚举而不是用 #defined 数字常量; 这允许结构化数字, 本地变量和函数参数用枚举类型来声明。如:

```
enum option {
    OPT_FOO = 1
    OPT_BAR = 2
    OPT_BAZ = 4
};

static enum option global_option;

static handle_option(const enum option opt)
{
    ...;
}
```

注意: 编译器将不强制你作为函数的参数从枚举中传递实体(entry);这种推荐的格式只是为了使代码更清楚和自文档化。另外, 当用 switch/case 代码块来分枝处理 enum 值时, 编译器将警告你如果你忘了处理一个或多个 enum 值, 但这些值在这里是可处理的。

4 串处理

不要用 strncpy 来拷贝整个字符串;它不能担保你所处理的字符串的缓存以空结束。使用 ast_copy_string 来替换前面提到的函数。这个函数也更为有效的处理(并允许传递实际的缓冲尺寸, 这使的代码更为清楚)

不要使用 ast_copy_string (或任何长度限制的拷贝函数)来拷贝固定长度的串到缓冲(在编译时长度固定), 如果缓冲是完成拷贝工作的函数所分配, 在这种情况下, 你知道固定长度的串是否有足够的缓存空间来保存数据, 如果没有足够的空间, 你的代码将不能正常的运行! 使用 strcpy() 来处理固定长度的串的情况, 如果你仅想保存单字符串于缓存中, 你可以直接设置缓存的最初两个字符。如果你想清空缓存, 只需要在缓存中的第一位中存一个空字符('\0');其它的都不需要, 并且其它的任何方法都是无用的。

另外, 如果在函数中的先前的操作已经决定了你想放入缓存中的字符串的尺寸(甚至于你都没有分配缓存的尺寸),你可直接使用 strcpy(), 由于它可以被内联处理并优化和简化操作, 而不象 ast_copy_string()。

5 使用函数

当生成一个应用时, 如果你想解析输入的数据串, 总是使用 ast_strdupa(data) 到一个本地指针。

```
if (data)
    mydata = ast_strdupa(data);
```

分开发参数到拨号计划 (dialplan) 和函数使用 `ast_app_separate_args()` 来分离参数到你的应用中，一旦你建立一个本地串的拷贝。

5.1 用 `strsep` 来解析串

当可能时使用 `strsep()` 来解析串；这里不用担心 ‘reentrancy’ 当用 `strtok()` 时，并且甚至于更改原来的串(这在 `man` 手册中有相关的警告信息)，在大多数情况下，这是正是你想得到的！

5.2 创建通用代码!

如果你多次进行一样或相似的操作，你可以建立一个函数或宏。

请确保你没有复制任何能在 API 调用中找到的函数。如果你复制了能在其它静态函数的功能，请考虑建立一个能共享的函数的价值。

6 指针和指针分配的处理问题

6.1 复引用和局部指针

总的来说复引用和局部化指针对处理你一个通道中与当前线程不相关的通道成员和你不希望锁定的成员。

```
channame = ast_strdupa(otherchan->name);
```

6.2 尽量在指针参数前用 `const` 修饰符

对于你不想修改的函数使用 `const` 于指针参数前，由于这允许编译器进行某些优化。总的来说，使用 ‘`const`’ 于你不打算直接更改的任何参数前都是不错的选项，当你用一种你不想要的方式来使用参数变量时，由于这样可以捕捉到你代码中的逻辑/输入错误，所以这是一个不错的选择。

6.3 不要建立你自己的链接表代码—重用!

作为这个指针通常的示例，尽量使用要以锁定的链接表 (linked-list) 宏，这个宏可以在

```
/include/asterisk/linkedlists.h
```

中找到。它们都是有效的，易用并且提供管理单链表元素时所需的操作(如有些东西丢失时，必须让我们知道!)。仅因为你看其它的开放源代码的代码树中列表执行无原因的继续使用其代码的拷贝.....

在 `/asterisk/strings.h` 文件中和 `asterisk/time.h` 文件中有许多通用的串操作和时间相关的操作函数。

6.4 请避免不需要的指针分配工作!

避免不需要的 `malloc()`, `strdup()` 调用。如果仅在你的函数中需要其值, 你可以使用 `ast_strdupa()` 或在堆栈中声明结构并将指针传递给它。然而, 请注意千万不要调用 `alloca()`, `ast_strdupa()` 或相似的函数于你正调用的函数的参数列表中; 这可能引起非常奇怪的堆栈安排和产生不希望的行为。

6.5 对结构进行分配

当对一个结构时行分配和清空时, 请试用以下的代码:

```
struct foo *tmp;

...

tmp = malloc(sizeof(*tmp));
if (tmp)
memset(tmp, 0, sizeof(*tmp));
```

这样做也排除了 'struct foo' 标识符的复制, 这确保了代码更容易被读, 同时也确保了它的拷贝一粘贴时不需要更多的编辑工作, 事实上, 你甚至可以这样来用:

```
struct foo * tmp;

...

tmp = calloc(1, sizeof(*tmp));
```

这将在一步操作中清零和分配相应的内存。

7 CLI 命令

新的 CLI 命令将使用模块的名字来命名, 之后跟随一个动词, 再之后是命令所需的任何参数。例如:

```
*CLI> iax2 show peer <peername>
```

不是

```
*CLI> show iax2 peer <peername>
```

8 新的拨号计划(dialplan)应用/函数

这里有两种向 Asterisk 的拨号计划添加功能方法：应用和函数。应用(在 apps/ 目录下可找到)是与一个通道进行交互/或用户用其它的明显的方法进行交互的代码的集合。函数(可以通过模块的类型产生)是当被提供的功能是简单时使用 ... `getting/retrieving` 一个值, 例如, 当一个操作无法和一个通道相关联时, 函数也被使用(如数值计算和串操作).应用通过调用 `ast_register_application` 函数来注册自己; 相关的代码请查看 `apps/app_skel.c` 文件中相关的示例。

函数通用调用 `ast_custom_function_register` 函数来完成对自己的注册工作。

9 Doxygen API 文档标准

当写一个 Asterisk API 文档时, 请按下面的格式来书写。不要使用 java 文档格式。

```
/*!
 * \brief Do interesting stuff.
 * \param thing1 interesting parameter 1.
 * \param thing2 interesting parameter 2.
 *
 * This function does some interesting stuff.
 *
 * \return zero on success, -1 on error.
 */
int ast_interesting_stuff(int thing1, int thing2)
{
    return 0;
}
```

注意使用 `\param`, `\brief`, 和 `\return` 结构。这里的说明将用于说明函数中相应代码部分的文档。同时注意在最后一个 `\param` 后的空行。所有的 doxygen 语句必须在 `/*! */` 之间, 如果函数或结构不需要一个特殊的说明, 说明可以不要。

请确保检查 doxygen 手册并且在字面上使用 `\a`, `\code`, `\c`, `\b`, `\note`, `\li` 和 `\e` 如何作为属性值来更改。

当写文档时, 在模块中的一个 'static' 函数或内部结构, 使用 `\internal` 更改标识来确保文档的结果如你所需的显示成 `\for internal use only`

结构的文档必须写成如下的格式:

```
/*!
 * \brief A very interesting structure.
 */
struct interesting_struct
{
    /*! \brief A data member. */
    int member1;

    int member2; /*!< \brief Another data member. */
}
```

注意 `/*! */` 块文档结构立即跟随着结构体，除非他们被写成 `/*!< */`，在这种情况下，他们的文档容器先行处理它。

10 在提交代码前结束代码

10.1 再次查看你的代码

当你完成你想要的功能，使用其它几种方法来优化代码。

10.2 读补丁

在提交一个补丁文件前，*读*补丁文件的所有代码，以确保你所希望的代码都在补丁文件中，并且这里没有你不希望的奇怪的更改发生。在你的开发期间，Asterisk的部分代码可能已发生改变，请确保你提交前使用的是最新的 SVN 版本。

10.3 听取建议

如果你被请求更改你的补丁文件，这是一个非常好的机会来更改你以前的 bug，在这个过程中，要更仔细的检查代码。同时注意 bug 统帅或新的开发者都是人，他们也可能犯错误：)

10.4 优化，优化，再优化

如果你想重用一个计算值，用保存你所需的变量值的方法而不是不停的重新计算所需的值。这可以防止你另外的计算中出现错误，如果你的公式有错误，要使其易于更正，这可能不能帮助优化代码，但至少可以帮助提高可靠性。

下面仅是一个示例，(所以不要过多的分析它，这会很丢面子):

```
const char *prefix = "pre";
const char *postfix = "post";
char *newname;
char *name = "data";

if (name && (newname = alloca(strlen(name) + strlen(prefix) + strlen(postfix) + 3)))
    snprintf(newname, strlen(name) + strlen(prefix) + strlen(postfix) + 3, "%s/%s/%s", prefix, name, postfix);

... vs this alternative;

const char *prefix = "pre";
const char *postfix = "post";
char *newname;
char *name = "data";
int len = 0;

if(name && (len = strlen(name) + strlen(prefix) + strlen(postfix) + 3) && (newname = alloca(len)))
    snprintf(newname, len, "%s/%s/%s", prefix, name, postfix);
```