C++实现的命令行参数管理

分类： C/C++语言 2012-11-20 15:34 1008人阅读 评论(0) 收藏 举报

在编写可运行程序时，经常需要输入除了可运行文件之外的其它的命令行参数，可以用传统的getopt函数来分析，本文基于面向对象，分析一种管理命令行参数方法 -- 来源于webrtc项目，在阅读过程中，大家分享一下。

一，传统命令行分析

```cpp
包含头文件：#include<unistd.h>
    int getopt(int argc,char * const argv[ ],const char * optstring);
    extern char *optarg;
    extern int optind, opterr, optopt;
```

二，命令行参数管理

假设命令行的输入格式的规则如下：

1. --flag 布尔类型。
2. --noflag 布尔类型。
3. --flag=value 等号周边没有空格。

2.1 参数的值封装---FlagValue

这个类对参数的值进行封装，如--prefix=/usr，作为一个命令行参数时，prefix为键，/usr为值。在参数中，在此定义值的类型为布尔、整型、浮点、字符串中的一种。

由于一个值在只能取四种的一种，所以此处用联合类型表示FlagValue。

```cpp
union FlagValue {
  static FlagValue New_BOOL(int b) {
    FlagValue v;
    v.b = (b != 0);
    return v;
  }

  static FlagValue New_INT(int i) {
    FlagValue v;
    v.i = i;
    return v;
  }
```

```
13.
14.   static FlagValue New_FLOAT(float f) {
15.     FlagValue v;
16.     v.f = f;
17.     return v;
18.   }
19.
20.   static FlagValue New_STRING(const char* s) {
21.     FlagValue v;
22.     v.s = s;
23.     return v;
24.   }
25.
26.   bool b;
27.   int i;
28.   double f;
29.   const char* s;
30. };
```

这个联合类型对命令行中键值对中的值进行封装，可以表示四种类型。

2.2 命令行中键值的表示 -- Flag

这个类是表示一对键值的抽象，包含下列元素：

1. 键值对。包括name和variable_表示键和值。如--prefix=/usr中，name的值为prefix，variable_为/usr的一个表示。
2. 链表维护域。Flag *next_用于指向下一个命令行参数。
3. comment_表示该参数的解释。
4. file表示和键值相关的外部文件。
5. default_表示默认情况下，就是用户没有输入该参数的情况下默认的值。
6. 定义友元类FlagList，因为FlagList需要访问Flag的next_域。

```
[cpp]

01. class Flag {
02.  public:
03.   enum Type { BOOL, INT, FLOAT, STRING };
04.
05.   // Internal use only.
06.   Flag(const char* file, const char* name, const char* comment,
07.        Type type, void* variable, FlagValue default_);
08.
09.   // General flag information
10.   const char* file() const  { return file_; }
11.   const char* name() const  { return name_; }
12.   const char* comment() const  { return comment_; }
13.
14.   // Flag type
15.   Type type() const  { return type_; }
16.
17.   // Flag variables
18.   bool* bool_variable() const {
19.     assert(type_ == BOOL);
```

```
20.     return &variable_->b;
21.   }
22.
23.   int* int_variable() const {
24.     assert(type_ == INT);
25.     return &variable_->i;
26.   }
27.
28.   double* float_variable() const {
29.     assert(type_ == FLOAT);
30.     return &variable_->f;
31.   }
32.
33.   const char** string_variable() const {
34.     assert(type_ == STRING);
35.     return &variable_->s;
36.   }
37.
38.   // Default values
39.   bool bool_default() const {
40.     assert(type_ == BOOL);
41.     return default_.b;
42.   }
43.
44.   int int_default() const {
45.     assert(type_ == INT);
46.     return default_.i;
47.   }
48.
49.   double float_default() const {
50.     assert(type_ == FLOAT);
51.     return default_.f;
52.   }
53.
54.   const char* string_default() const {
55.     assert(type_ == STRING);
56.     return default_.s;
57.   }
58.
59.   // Resets a flag to its default value
60.   void SetToDefault();
61.
62.   // Iteration support
63.   Flag* next() const  { return next_; }
64.
65.   // Prints flag information. The current flag value is only printed
66.   // if print_current_value is set.
67.   void Print(bool print_current_value);
68.
69.  private:
70.   const char* file_;
71.   const char* name_;
72.   const char* comment_;
73.
74.   Type type_;
75.   FlagValue* variable_;
76.   FlagValue default_;
77.
78.   Flag* next_;
79.
```

```
80.   friend class FlagList;  // accesses next_
81. };
```

2.3 命令行键值链表--- FlagList

这个类维护一个全局的链表，链表中每一项都是命令行参数解析的结果，如：--prefix=/usr --localstatedir=/var/data 这就表示两个Flag对象，通过Flag对象的next域来关联。

这个类的属性和方法都是静态的，属性只有Flag* list_，用于维护命令行所有输入的参数所组成的链表。

主要方法如下：

SetFkagsFromCommandLine：解析根据命令行的输入，这里传入的是所有的命令行输入。

SplitArgument：解析命令行中具体的一个可以被解析的键值对。

```
[cpp]

01.  class FlagList {
02.  public:
03.   FlagList();
04.
05.   static Flag* list()  { return list_; }
06.
07.   static void Print(const char* file, bool print_current_value);
08.
09.   static Flag* Lookup(const char* name);
10.
11.   static void SplitArgument(const char* arg,
12.                             char* buffer, int buffer_size,
13.                             const char** name, const char** value,
14.                             bool* is_bool);
15.
16.   static int SetFlagsFromCommandLine(int* argc,
17.                                      const char** argv);
18.   static inline int SetFlagsFromCommandLine(int* argc,
19.                                             char** argv) {
20.     return SetFlagsFromCommandLine(argc, const_cast<const char**>(argv));
21.   }
22.
23.  private:
24.   static Flag* list_;
25. };
```

2.4 实现

先看在链表中，查找指定的参数，这个实现比较简单。

```cpp
Flag* FlagList::Lookup(const char* name) {
  Flag* f = list_;
  while (f != NULL && strcmp(name, f->name()) != 0)
    f = f->next();
  return f;
}
```

解析特定的命令行参数函数

```cpp
void FlagList::SplitArgument(const char* arg,
                             char* buffer, int buffer_size,
                             const char** name, const char** value,
                             bool* is_bool) {
 *name = NULL;
 *value = NULL;
 *is_bool = false;

 if (*arg == '-') {
   // find the begin of the flag name
   arg++;  // remove 1st '-'
   if (*arg == '-')
     arg++;  // remove 2nd '-'
   if (arg[0] == 'n' && arg[1] == 'o') {
     arg += 2;  // remove "no"
     *is_bool = true;
   }
   *name = arg;

   // find the end of the flag name
   while (*arg != '\0' && *arg != '=')
     arg++;

   // get the value if any
   if (*arg == '=') {
     // make a copy so we can NUL-terminate flag name
     int n = static_cast<int>(arg - *name);
     if (n >= buffer_size)
       Fatal(__FILE__, __LINE__, "CHECK(%s) failed", "n < buffer_size");
     memcpy(buffer, *name, n * sizeof(char));
     buffer[n] = '\0';
     *name = buffer;
     // get the value
     *value = arg + 1;
   }
 }

```

上面的函数是对诸如--prefix=/usr 、--prefix、--prefix /usr之类的命令行进行解析，如果某项输入不以-开头，则不被解析。

流程如下：

1. 如果以"-"开头，则去掉这个开头，接下来如果还是"-",还是把这个字符去掉。

2. 如果接下来的字符是no，则表示这是一个布尔类型输入，此时将arg的值赋值给name。
3. 然后一直跳过字符，直到遇到“=”号，或者arg的结尾。
4. 如果后面是“=”号，则后面的部分为name所对应的value值。

解析所有的命令行输入

```cpp
int FlagList::SetFlagsFromCommandLine(int* argc, const char** argv) {
  // parse arguments
  for (int i = 1; i < *argc; /* see below */) {
    int j = i;  // j > 0
    const char* arg = argv[i++];

    // split arg into flag components
    char buffer[1024];
    const char* name;
    const char* value;
    bool is_bool;
    SplitArgument(arg, buffer, sizeof buffer, &name, &value, &is_bool);

    if (name != NULL) {
      // lookup the flag
      Flag* flag = Lookup(name);
      if (flag == NULL) {
        fprintf(stderr, "Error: unrecognized flag %s\n", arg);
        return j;
      }

      // if we still need a flag value, use the next argument if available
      if (flag->type() != Flag::BOOL && value == NULL) {
        if (i < *argc) {
          value = argv[i++];
        } else {
          fprintf(stderr, "Error: missing value for flag %s of type %s\n",
            arg, Type2String(flag->type()));
          return j;
        }
      }

      // set the flag
      char empty[] = { '\0' };
      char* endp = empty;
      switch (flag->type()) {
        case Flag::BOOL:
          *flag->bool_variable() = !is_bool;
          break;
        case Flag::INT:
          *flag->int_variable() = strtol(value, &endp, 10);
          break;
        case Flag::FLOAT:
          *flag->float_variable() = strtod(value, &endp);
          break;
        case Flag::STRING:
          *flag->string_variable() = value;
          break;
      }
```

```
50.
51.       // handle errors
52.       if ((flag->type() == Flag::BOOL && value != NULL) ||
53.           (flag->type() != Flag::BOOL && is_bool) ||
54.           *endp != '\0') {
55.         fprintf(stderr, "Error: illegal value for flag %s of type %s\n",
56.           arg, Type2String(flag->type()));
57.         return j;
58.       }
59.
60.     }
61.   }
62.   return 0;
63. }
```

源码下载地址：http://dow nload.csdn.net/detail/zmxiangde_88/4789141

粘贴如下：

flags.h

```
[cpp]
01. #ifndef TALK_BASE_FLAGS_H__
02. #define TALK_BASE_FLAGS_H__
03.
04. #include <assert.h>
05.
06. #include "talk/base/checks.h"
07. #include "talk/base/common.h"
08.
09. // Internal use only.
10. union FlagValue {
11.   // Note: Because in C++ non-bool values are silently converted into
12.   // bool values ('bool b = "false";' results in b == true!), we pass
13.   // and int argument to New_BOOL as this appears to be safer - sigh.
14.   // In particular, it prevents the (not uncommon!) bug where a bool
15.   // flag is defined via: DEFINE_bool(flag, "false", "some comment");.
16.   static FlagValue New_BOOL(int b) {
17.     FlagValue v;
18.     v.b = (b != 0);
19.     return v;
20.   }
21.
22.   static FlagValue New_INT(int i) {
23.     FlagValue v;
24.     v.i = i;
25.     return v;
26.   }
27.
28.   static FlagValue New_FLOAT(float f) {
29.     FlagValue v;
30.     v.f = f;
31.     return v;
32.   }
33.
34.   static FlagValue New_STRING(const char* s) {
35.     FlagValue v;
36.     v.s = s;
```

```
37.     return v;
38.   }
39.
40.   bool b;
41.   int i;
42.   double f;
43.   const char* s;
44. };
45.
46.
47. // Each flag can be accessed programmatically via a Flag object.
48. class Flag {
49.  public:
50.   enum Type { BOOL, INT, FLOAT, STRING };
51.
52.   // Internal use only.
53.   Flag(const char* file, const char* name, const char* comment,
54.        Type type, void* variable, FlagValue default_);
55.
56.   // General flag information
57.   const char* file() const  { return file_; }
58.   const char* name() const  { return name_; }
59.   const char* comment() const  { return comment_; }
60.
61.   // Flag type
62.   Type type() const  { return type_; }
63.
64.   // Flag variables
65.   bool* bool_variable() const {
66.     assert(type_ == BOOL);
67.     return &variable_->b;
68.   }
69.
70.   int* int_variable() const {
71.     assert(type_ == INT);
72.     return &variable_->i;
73.   }
74.
75.   double* float_variable() const {
76.     assert(type_ == FLOAT);
77.     return &variable_->f;
78.   }
79.
80.   const char** string_variable() const {
81.     assert(type_ == STRING);
82.     return &variable_->s;
83.   }
84.
85.   // Default values
86.   bool bool_default() const {
87.     assert(type_ == BOOL);
88.     return default_.b;
89.   }
90.
91.   int int_default() const {
92.     assert(type_ == INT);
93.     return default_.i;
94.   }
95.
96.   double float_default() const {
```

```
97.     assert(type_ == FLOAT);
98.     return default_.f;
99.   }
100.
101.  const char* string_default() const {
102.    assert(type_ == STRING);
103.    return default_.s;
104.  }
105.
106.  // Resets a flag to its default value
107.  void SetToDefault();
108.
109.  // Iteration support
110.  Flag* next() const  { return next_; }
111.
112.  // Prints flag information. The current flag value is only printed
113.  // if print_current_value is set.
114.  void Print(bool print_current_value);
115.
116. private:
117.  const char* file_;
118.  const char* name_;
119.  const char* comment_;
120.
121.  Type type_;
122.  FlagValue* variable_;
123.  FlagValue default_;
124.
125.  Flag* next_;
126.
127.  friend class FlagList;  // accesses next_
128. };
129.
130.
131. // Internal use only.
132. #define DEFINE_FLAG(type, c_type, name, default, comment) \
133.   /* define and initialize the flag */                    \
134.   c_type FLAG_##name = (default);                          \
135.   /* register the flag */                                 \
136.   static Flag Flag_##name(__FILE__, #name, (comment),   \
137.                           Flag::type, &FLAG_##name,       \
138.                           FlagValue::New_##type(default))
139.
140.
141. // Internal use only.
142. #define DECLARE_FLAG(c_type, name)              \
143.   /* declare the external flag */               \
144.   extern c_type FLAG_##name
145.
146.
147. // Use the following macros to define a new flag:
148. #define DEFINE_bool(name, default, comment) \
149.   DEFINE_FLAG(BOOL, bool, name, default, comment)
150. #define DEFINE_int(name, default, comment) \
151.   DEFINE_FLAG(INT, int, name, default, comment)
152. #define DEFINE_float(name, default, comment) \
153.   DEFINE_FLAG(FLOAT, double, name, default, comment)
154. #define DEFINE_string(name, default, comment) \
155.   DEFINE_FLAG(STRING, const char*, name, default, comment)
156.
```

```
157.
158.  // Use the following macros to declare a flag defined elsewhere:
159.  #define DECLARE_bool(name)  DECLARE_FLAG(bool, name)
160.  #define DECLARE_int(name)  DECLARE_FLAG(int, name)
161.  #define DECLARE_float(name)  DECLARE_FLAG(double, name)
162.  #define DECLARE_string(name)  DECLARE_FLAG(const char*, name)
163.
164.
165.  // The global list of all flags.
166.  class FlagList {
167.   public:
168.    FlagList();
169.
170.    // The NULL-terminated list of all flags. Traverse with Flag::next().
171.    static Flag* list()  { return list_; }
172.
173.    // If file != NULL, prints information for all flags defined in file;
174.    // otherwise prints information for all flags in all files. The current
175.    // flag value is only printed if print_current_value is set.
176.    static void Print(const char* file, bool print_current_value);
177.
178.    // Lookup a flag by name. Returns the matching flag or NULL.
179.    static Flag* Lookup(const char* name);
180.
181.    // Helper function to parse flags: Takes an argument arg and splits it into
182.    // a flag name and flag value (or NULL if they are missing). is_bool is set
183.    // if the arg started with "-no" or "--no". The buffer may be used to NUL-
184.    // terminate the name, it must be large enough to hold any possible name.
185.    static void SplitArgument(const char* arg,
186.                              char* buffer, int buffer_size,
187.                              const char** name, const char** value,
188.                              bool* is_bool);
189.
190.    // Set the flag values by parsing the command line. If remove_flags
191.    // is set, the flags and associated values are removed from (argc,
192.    // argv). Returns 0 if no error occurred. Otherwise, returns the
193.    // argv index > 0 for the argument where an error occurred. In that
194.    // case, (argc, argv) will remain unchanged indepdendent of the
195.    // remove_flags value, and no assumptions about flag settings should
196.    // be made.
197.    //
198.    // The following syntax for flags is accepted (both '-' and '--' are ok):
199.    //
200.    //   --flag        (bool flags only)
201.    //   --noflag      (bool flags only)
202.    //   --flag=value  (non-bool flags only, no spaces around '=')
203.    //   --flag value  (non-bool flags only)
204.    static int SetFlagsFromCommandLine(int* argc,
205.                                       const char** argv,
206.                                       bool remove_flags);
207.    static inline int SetFlagsFromCommandLine(int* argc,
208.                                              char** argv,
209.                                              bool remove_flags) {
210.      return SetFlagsFromCommandLine(argc, const_cast<const char**>(argv),
211.                                     remove_flags);
212.    }
213.
214.    // Registers a new flag. Called during program initialization. Not
215.    // thread-safe.
216.    static void Register(Flag* flag);
```

```
217.
218.  private:
219.   static Flag* list_;
220. };
221.
222. #ifdef WIN32
223. // A helper class to translate Windows command line arguments into UTF8,
224. // which then allows us to just pass them to the flags system.
225. // This encapsulates all the work of getting the command line and translating
226. // it to an array of 8-bit strings; all you have to do is create one of these,
227. // and then call argc() and argv().
228. class WindowsCommandLineArguments {
229.  public:
230.   WindowsCommandLineArguments();
231.   ~WindowsCommandLineArguments();
232.
233.   int argc() { return argc_; }
234.   char **argv() { return argv_; }
235.  private:
236.   int argc_;
237.   char **argv_;
238.
239.  private:
240.   DISALLOW_EVIL_CONSTRUCTORS(WindowsCommandLineArguments);
241. };
242. #endif  // WIN32
243.
244.
245. #endif  // SHARED_COMMANDLINEFLAGS_FLAGS_H__
```

flags.cc

```
[cpp]

01. #include <stdio.h>
02. #include <stdlib.h>
03. #include <string.h>
04.
05.
06. #ifdef WIN32
07. #include "talk/base/win32.h"
08. #include <shellapi.h>
09. #endif
10.
11. #include "talk/base/flags.h"
12.
13.
14. // -----------------------------------------------------------------------------
15. // Implementation of Flag
16.
17. Flag::Flag(const char* file, const char* name, const char* comment,
18.            Type type, void* variable, FlagValue default__)
19.     : file_(file),
20.       name_(name),
21.       comment_(comment),
22.       type_(type),
23.       variable_(reinterpret_cast<FlagValue*>(variable)),
24.       default_(default__) {
```

```
25.   FlagList::Register(this);
26. }
27.
28.
29. void Flag::SetToDefault() {
30.   // Note that we cannot simply do '*variable_ = default_;' since
31.   // flag variables are not really of type FlagValue and thus may
32.   // be smaller! The FlagValue union is simply 'overlayed' on top
33.   // of a flag variable for convenient access. Since union members
34.   // are guarantee to be aligned at the beginning, this works.
35.   switch (type_) {
36.     case Flag::BOOL:
37.       variable_->b = default_.b;
38.       return;
39.     case Flag::INT:
40.       variable_->i = default_.i;
41.       return;
42.     case Flag::FLOAT:
43.       variable_->f = default_.f;
44.       return;
45.     case Flag::STRING:
46.       variable_->s = default_.s;
47.       return;
48.   }
49.   UNREACHABLE();
50. }
51.
52.
53. static const char* Type2String(Flag::Type type) {
54.   switch (type) {
55.     case Flag::BOOL: return "bool";
56.     case Flag::INT: return "int";
57.     case Flag::FLOAT: return "float";
58.     case Flag::STRING: return "string";
59.   }
60.   UNREACHABLE();
61.   return NULL;
62. }
63.
64.
65. static void PrintFlagValue(Flag::Type type, FlagValue* p) {
66.   switch (type) {
67.     case Flag::BOOL:
68.       printf("%s", (p->b ? "true" : "false"));
69.       return;
70.     case Flag::INT:
71.       printf("%d", p->i);
72.       return;
73.     case Flag::FLOAT:
74.       printf("%f", p->f);
75.       return;
76.     case Flag::STRING:
77.       printf("%s", p->s);
78.       return;
79.   }
80.   UNREACHABLE();
81. }
82.
83.
84. void Flag::Print(bool print_current_value) {
```

```
85.   printf("  --%s (%s)  type: %s  default: ", name_, comment_,
86.          Type2String(type_));
87.   PrintFlagValue(type_, &default_);
88.   if (print_current_value) {
89.     printf("  current value: ");
90.     PrintFlagValue(type_, variable_);
91.   }
92.   printf("\n");
93. }
94.
95.
96. // ---------------------------------------------------------------------------
97. // Implementation of FlagList
98.
99. Flag* FlagList::list_ = NULL;
100.
101.
102. FlagList::FlagList() {
103.   list_ = NULL;
104. }
105.
106. void FlagList::Print(const char* file, bool print_current_value) {
107.   // Since flag registration is likely by file (= C++ file),
108.   // we don't need to sort by file and still get grouped output.
109.   const char* current = NULL;
110.   for (Flag* f = list_; f != NULL; f = f->next()) {
111.     if (file == NULL || file == f->file()) {
112.       if (current != f->file()) {
113.         printf("Flags from %s:\n", f->file());
114.         current = f->file();
115.       }
116.       f->Print(print_current_value);
117.     }
118.   }
119. }
120.
121.
122. Flag* FlagList::Lookup(const char* name) {
123.   Flag* f = list_;
124.   while (f != NULL && strcmp(name, f->name()) != 0)
125.     f = f->next();
126.   return f;
127. }
128.
129.
130. void FlagList::SplitArgument(const char* arg,
131.                              char* buffer, int buffer_size,
132.                              const char** name, const char** value,
133.                              bool* is_bool) {
134.   *name = NULL;
135.   *value = NULL;
136.   *is_bool = false;
137.
138.   if (*arg == '-') {
139.     // find the begin of the flag name
140.     arg++;  // remove 1st '-'
141.     if (*arg == '-')
142.       arg++;  // remove 2nd '-'
143.     if (arg[0] == 'n' && arg[1] == 'o') {
144.       arg += 2;  // remove "no"
```

```
145.       *is_bool = true;
146.     }
147.     *name = arg;
148.
149.     // find the end of the flag name
150.     while (*arg != '\0' && *arg != '=')
151.       arg++;
152.
153.     // get the value if any
154.     if (*arg == '=') {
155.       // make a copy so we can NUL-terminate flag name
156.       int n = static_cast<int>(arg - *name);
157.       if (n >= buffer_size)
158.         Fatal(__FILE__, __LINE__, "CHECK(%s) failed", "n < buffer_size");
159.       memcpy(buffer, *name, n * sizeof(char));
160.       buffer[n] = '\0';
161.       *name = buffer;
162.       // get the value
163.       *value = arg + 1;
164.     }
165.   }
166. }
167.
168.
169. int FlagList::SetFlagsFromCommandLine(int* argc, const char** argv,
170.                                       bool remove_flags) {
171.   // parse arguments
172.   for (int i = 1; i < *argc; /* see below */) {
173.     int j = i;  // j > 0
174.     const char* arg = argv[i++];
175.
176.     // split arg into flag components
177.     char buffer[1024];
178.     const char* name;
179.     const char* value;
180.     bool is_bool;
181.     SplitArgument(arg, buffer, sizeof buffer, &name, &value, &is_bool);
182.
183.     if (name != NULL) {
184.       // lookup the flag
185.       Flag* flag = Lookup(name);
186.       if (flag == NULL) {
187.         fprintf(stderr, "Error: unrecognized flag %s\n", arg);
188.         return j;
189.       }
190.
191.       // if we still need a flag value, use the next argument if available
192.       if (flag->type() != Flag::BOOL && value == NULL) {
193.         if (i < *argc) {
194.           value = argv[i++];
195.         } else {
196.           fprintf(stderr, "Error: missing value for flag %s of type %s\n",
197.             arg, Type2String(flag->type()));
198.           return j;
199.         }
200.       }
201.
202.       // set the flag
203.       char empty[] = { '\0' };
204.       char* endp = empty;
```

```
205.       switch (flag->type()) {
206.         case Flag::BOOL:
207.           *flag->bool_variable() = !is_bool;
208.           break;
209.         case Flag::INT:
210.           *flag->int_variable() = strtol(value, &endp, 10);
211.           break;
212.         case Flag::FLOAT:
213.           *flag->float_variable() = strtod(value, &endp);
214.           break;
215.         case Flag::STRING:
216.           *flag->string_variable() = value;
217.           break;
218.       }
219.
220.       // handle errors
221.       if ((flag->type() == Flag::BOOL && value != NULL) ||
222.           (flag->type() != Flag::BOOL && is_bool) ||
223.           *endp != '\0') {
224.         fprintf(stderr, "Error: illegal value for flag %s of type %s\n",
225.           arg, Type2String(flag->type()));
226.         return j;
227.       }
228.
229.       // remove the flag & value from the command
230.       if (remove_flags)
231.         while (j < i)
232.           argv[j++] = NULL;
233.     }
234.   }
235.
236.   // shrink the argument list
237.   if (remove_flags) {
238.     int j = 1;
239.     for (int i = 1; i < *argc; i++) {
240.       if (argv[i] != NULL)
241.         argv[j++] = argv[i];
242.     }
243.     *argc = j;
244.   }
245.
246.   // parsed all flags successfully
247.   return 0;
248. }
249.
250. void FlagList::Register(Flag* flag) {
251.   assert(flag != NULL && strlen(flag->name()) > 0);
252.   if (Lookup(flag->name()) != NULL)
253.     Fatal(flag->file(), 0, "flag %s declared twice", flag->name());
254.   flag->next_ = list_;
255.   list_ = flag;
256. }
257.
258. #ifdef WIN32
259. WindowsCommandLineArguments::WindowsCommandLineArguments() {
260.   // start by getting the command line.
261.   LPTSTR command_line = ::GetCommandLine();
262.    // now, convert it to a list of wide char strings.
263.   LPWSTR *wide_argv = ::CommandLineToArgvW(command_line, &argc_);
264.   // now allocate an array big enough to hold that many string pointers.
```

```
265.    argv_ = new char*[argc_];
266.
267.    // iterate over the returned wide strings;
268.    for(int i = 0; i < argc_; ++i) {
269.      std::string s = talk_base::ToUtf8(wide_argv[i], wcslen(wide_argv[i]));
270.      char *buffer = new char[s.length() + 1];
271.      talk_base::strcpyn(buffer, s.length() + 1, s.c_str());
272.
273.      // make sure the argv array has the right string at this point.
274.      argv_[i] = buffer;
275.    }
276.    LocalFree(wide_argv);
277.  }
278.
279.  WindowsCommandLineArguments::~WindowsCommandLineArguments() {
280.    // need to free each string in the array, and then the array.
281.    for(int i = 0; i < argc_; i++) {
282.      delete[] argv_[i];
283.    }
284.
285.    delete[] argv_;
286.  }
287.  #endif  // WIN32
```

上一篇 No package 'gtk+-2.0' found的错误

下一篇 assert宏的深入学习

Senior Ja... /Software E...
上海瀚纳仕人才管理咨询有限公 | 15-30K/月 我要跳槽

Flex Consultant Specialist
上海瀚纳仕人才管理咨询有限公 | 20-35K/月 我要跳槽

赴日软件开发工程师
沈阳龙信科技有限公司 | 15-25K/月 我要跳槽

Software .NET (C#/ASP/VB) Developer
上海瀚纳仕人才管理咨询有限公 | 15-26K/月 我要跳槽

用 户 名： u012288867

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN
Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5
Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML
components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone
CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl
Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala
Django Bootstrap