G++ 2.91.57，cygnus\cygwin-b20\include\g++**type_traits.h** 完整列表

```
/*
 *
 * Copyright (c) 1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __TYPE_TRAITS_H
#define __TYPE_TRAITS_H

#ifndef __STL_CONFIG_H
#include <stl_config.h>
#endif

/*
This header file provides a framework for allowing compile time dispatch
based on type attributes. This is useful when writing template code.
For example, when making a copy of an array of an unknown type, it helps
to know if the type has a trivial copy constructor or not, to help decide
if a memcpy can be used.

The class template __type_traits provides a series of typedefs each of
which is either __true_type or __false_type. The argument to
__type_traits can be any type. The typedefs within this template will
attain their correct values by one of these means:
    1. The general instantiation contain conservative values which work
       for all types.
    2. Specializations may be declared to make distinctions between types.
    3. Some compilers (such as the Silicon Graphics N32 and N64 compilers)
       will automatically provide the appropriate specializations for all
       types.

EXAMPLE:

//Copy an array of elements which have non-trivial copy constructors
template <class T> void copy(T* source,T* destination,int n,__false_type);
//Copy an array of elements which have trivial copy constructors. Use memcpy.
template <class T> void copy(T* source,T* destination,int n,__true_type);

//Copy an array of any type by using the most efficient copy mechanism
```

```
template <class T> inline void copy(T* source,T* destination,int n) {
   copy(source,destination,n,typename
       __type_traits<T>::has_trivial_copy_constructor());
}
*/


struct __true_type {
};

struct __false_type {
};

template <class type>
struct __type_traits {
   typedef __true_type    this_dummy_member_must_be_first;
                /* Do not remove this member. It informs a compiler which
                   automatically specializes __type_traits that this
                   __type_traits template is special. It just makes sure that
                   things work if an implementation is using a template
                   called __type_traits for something unrelated. */

   /* The following restrictions should be observed for the sake of
      compilers which automatically produce type specific specializations
      of this class:
          - You may reorder the members below if you wish
          - You may remove any of the members below if you wish
          - You must not rename members without making the corresponding
            name change in the compiler
          - Members you add will be treated like regular members unless
            you add the appropriate support in the compiler. */


   typedef __false_type   has_trivial_default_constructor;
   typedef __false_type   has_trivial_copy_constructor;
   typedef __false_type   has_trivial_assignment_operator;
   typedef __false_type   has_trivial_destructor;
   typedef __false_type   is_POD_type;
};



// Provide some specializations.  This is harmless for compilers that
//  have built-in __types_traits support, and essential for compilers
//  that don't.

__STL_TEMPLATE_NULL struct __type_traits<char> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
```

```
   typedef __true_type   has_trivial_assignment_operator;
   typedef __true_type   has_trivial_destructor;
   typedef __true_type   is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<signed char> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
   typedef __true_type   has_trivial_assignment_operator;
   typedef __true_type   has_trivial_destructor;
   typedef __true_type   is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned char> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
   typedef __true_type   has_trivial_assignment_operator;
   typedef __true_type   has_trivial_destructor;
   typedef __true_type   is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<short> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
   typedef __true_type   has_trivial_assignment_operator;
   typedef __true_type   has_trivial_destructor;
   typedef __true_type   is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned short> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
   typedef __true_type   has_trivial_assignment_operator;
   typedef __true_type   has_trivial_destructor;
   typedef __true_type   is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<int> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
   typedef __true_type   has_trivial_assignment_operator;
   typedef __true_type   has_trivial_destructor;
   typedef __true_type   is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned int> {
   typedef __true_type   has_trivial_default_constructor;
   typedef __true_type   has_trivial_copy_constructor;
   typedef __true_type   has_trivial_assignment_operator;
```

```
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<long> {
   typedef __true_type    has_trivial_default_constructor;
   typedef __true_type    has_trivial_copy_constructor;
   typedef __true_type    has_trivial_assignment_operator;
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<unsigned long> {
   typedef __true_type    has_trivial_default_constructor;
   typedef __true_type    has_trivial_copy_constructor;
   typedef __true_type    has_trivial_assignment_operator;
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<float> {
   typedef __true_type    has_trivial_default_constructor;
   typedef __true_type    has_trivial_copy_constructor;
   typedef __true_type    has_trivial_assignment_operator;
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<double> {
   typedef __true_type    has_trivial_default_constructor;
   typedef __true_type    has_trivial_copy_constructor;
   typedef __true_type    has_trivial_assignment_operator;
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;
};

__STL_TEMPLATE_NULL struct __type_traits<long double> {
   typedef __true_type    has_trivial_default_constructor;
   typedef __true_type    has_trivial_copy_constructor;
   typedef __true_type    has_trivial_assignment_operator;
   typedef __true_type    has_trivial_destructor;
   typedef __true_type    is_POD_type;
};

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class T>
struct __type_traits<T*> {
   typedef __true_type    has_trivial_default_constructor;
```

```
  typedef __true_type   has_trivial_copy_constructor;
  typedef __true_type   has_trivial_assignment_operator;
  typedef __true_type   has_trivial_destructor;
  typedef __true_type   is_POD_type;
};

#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */

struct __type_traits<char*> {
  typedef __true_type   has_trivial_default_constructor;
  typedef __true_type   has_trivial_copy_constructor;
  typedef __true_type   has_trivial_assignment_operator;
  typedef __true_type   has_trivial_destructor;
  typedef __true_type   is_POD_type;
};

struct __type_traits<signed char*> {
  typedef __true_type   has_trivial_default_constructor;
  typedef __true_type   has_trivial_copy_constructor;
  typedef __true_type   has_trivial_assignment_operator;
  typedef __true_type   has_trivial_destructor;
  typedef __true_type   is_POD_type;
};

struct __type_traits<unsigned char*> {
  typedef __true_type   has_trivial_default_constructor;
  typedef __true_type   has_trivial_copy_constructor;
  typedef __true_type   has_trivial_assignment_operator;
  typedef __true_type   has_trivial_destructor;
  typedef __true_type   is_POD_type;
};

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */


#endif /* __TYPE_TRAITS_H */

// Local Variables:
// mode:C++
// End:
```