第5章 C/C++线程专有存储（Thread-Specific Storage）：用于访问“per-Thread”状态的对象行为模式

Douglas C. Schmidt Nat Pryce Timothy H. Harrison

摘 要

在理论上，使应用多线程化可以改善性能（通过同时执行多个指令流），并简化程序结构（通过允许每个线程同步地、而不是反应式地或异步地执行）。而在实践中，由于获取和释放锁的开销，多线程应用常常并不比单线程应用执行得更好，甚至还会更糟。此外，由于避免条件竞争和死锁所需的复杂的并发控制协议，多线程应用常常难以编程。

本论文描述线程专有存储（Thread-Specific Storage）模式，该模式可减轻多线程性能和编程复杂性的若干问题。通过允许多个线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问带来锁定开销，线程专有存储模式可改善性能，并简化多线程应用。

5.1 意图

允许多个线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问带来锁定开销。

5.2 动机

5.2.1 上下文和压力

线程专有存储应被用于这样的多线程应用：它们经常访问那些*逻辑上*是全局的、而*物理上*是专有于每个线程的对象。例如，像UNIX和Win32这样的操作系统使用errno来向应用报告错误信息。当错误在系统调用中发生时，OS设置errno来报告问题、并返回文档化的失败状态。当应用检测到失败时，它检查errno来确定发生了何种类型的错误。

例如，考虑下面典型的C代码片段，它接收来自非阻塞TCP socket的缓冲区：

```
// One global errno per-process.
```

```
extern int errno;


void *worker (SOCKET socket)

{

// Read from the network connection

// and process the data until the connection

// is closed.

for (;;)

{

       char buffer[BUFSIZ];

       int result = recv (socket, buffer, BUFSIZ, 0);



       // Check to see if the recv() call failed.

       if (result == -1)

       {

              if (errno != EWOULDBLOCK)

                            // Record error result in thread-specific
                            data.

                            printf ("recv failed, errno = %d", errno);

       }

       else

              // Perform the work on success.

              process_buffer (buffer);

}

}
```

如果recv返回-1，代码检查errno是否等于 EWOULDBLOCK，如果不是（例如，如果errno = EINTR）就打印出错误消息；返回的不是-1代码就处理它接收到的缓冲区。

5.2.2 常见陷阱和缺陷

尽管上面所示的“全局错误变量”方法对于单线程应用工作得相当好，在多线程应用中却会发生微妙的问题。特别地，占先式多线程系统中的条件竞争会导致一个线程中的方法所设置的errno值被另一线程中的应用错误地解释。因而，如果多个线程同时执行worker函数，全局版本的errno就有可能会由于条件竞争而被不正确地设置。

例如，在图5-1中两个线程（T1和T2）可以在socket上执行recv调用。在此例中，T1的recv返回-1，并设置errno为EWOULDBLOCK，指示目前没有数据在socket上排队。但是在它检查这种情况之前，

T1被占先，T2运行。假设T2被中断，它设置errno为EINTR。如果T2随之又立即被占先，T1将会错误地假定它的recv调用被中断，并执行错误的动作。因而，这个程序是错误的和不可移植的，因为它的行为依赖于线程执行的顺序。

在这之下的问题是对全局errno值的设置和测试发生在两个步骤中：（1）recv调用设置该值和（2）应用测试该值。因此，“显而易见”的解决方案，即用互斥体包装errno并不能解决竞争状态，因为设置/测试涉及到多个操作（也就是，它不是原子的）。

解决此问题的一种途径是创建更为成熟的锁定协议。例如，recv调用可以在内部获取errno_mutex，并且必须由应用在recv返回、errno的值被测试后来释放它。但是，此方案并不合乎需要，因为应用可能会忘记释放锁，从而导致饥饿和死锁。而且，如果应用必须在每次库调用后检查错误状态，额外的锁定开销将会显著地降低性能，即使在没有使用多个线程的情况下也是如此。

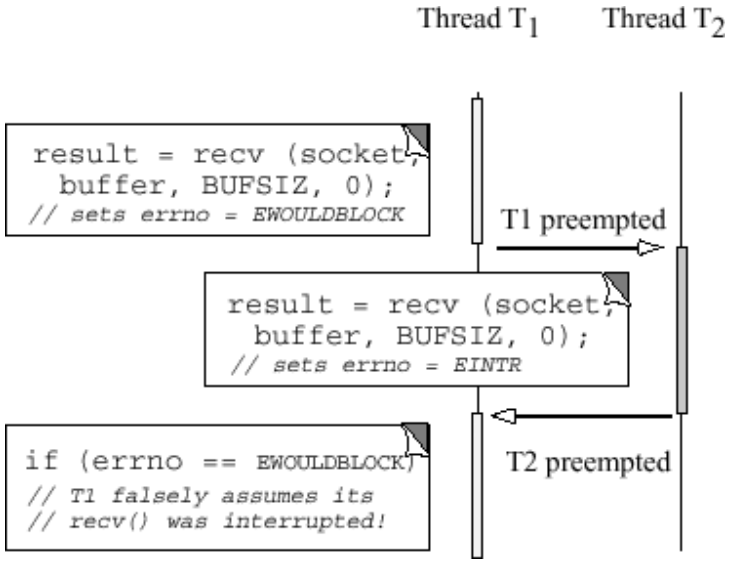

图5-1 多线程程序中的条件竞争

5.2.3 解决方案：线程专有存储

上面描述的陷阱和缺陷的一种通用解决方案是使用*线程专有存储*模式。该模式消除以下压力：

- **效率：**线程专有存储允许线程中的一系列方法原子地访问线程专有的对象，而又不会给每次访问带来锁定开销。
- **简化应用编程：**对于应用程序员来说，线程专有存储易于使用，因为系统开发者可以通过数据抽象或宏来使线程专有存储的使用在源码级完全透明化。
- **高度可移植：**线程专有存储在大多数多线程OS平台上都可用，并且可以在缺乏它的平台上（比如VxWorks）方便地实现。

因此，不管应用是运行在单线程还是多线程中，使用线程专有存储模式都不会带来额外开销，并且无需改动代码。例如，下面的代码演示errno是如何在Solaris 2.x上定义的：

```
// A thread-specific errno definition (typically

// defined in <sys/errno.h>).


#if defined (_REENTRANT)

// The _errno() function returns the

// thread-specific value of errno.

#define errno (*_errno())

#else

// Non-MT behavior is unchanged.

extern int errno;

#endif /* REENTRANT */


void *worker (SOCKET socket)

{

// Exactly the same implementation shown above.

}
```

如果_REENTRANT标志被设置，errno符号就被定义为调用名为_errno的助手函数的宏，此函数返回一个指向errno的线程专有值的指针。该指针被宏去除引用，以使它能够任意出现在赋值运算的左边或右边。

5.3 适用性

应用有以下特性时可使用线程专有存储：

- 应用最初的编写假定了单线程控制，并正在被移植到多线程环境，而又*不能*改变现有API；或是
- 应用含有多个占先式线程控制，可以任意的调度顺序并发执行；以及
- 每个线程控制调用一系列方法，这些方法共享只对该线程来说是公用的数据；以及
- 在每个线程中被对象共享的数据必须通过一个全局可见的访问点来访问；该访问点“逻辑地”与其他线程共享，但在“物理上” 对于每个线程却是唯一的；以及
- 数据在方法间隐式地传递，而不是经由参数显式地传递。

理解上面描述的特性对于使用（或不使用）线程专有存储模式来说是至关紧要的。例如，UNIX errno变量是一个数据例子：（1）逻辑上全局，但是物理上线程专有，以及（2）在方法间隐式地传递。

当应用有以下特性时，*不要*使用线程专有存储模式：

- 多个线程为单个任务协同工作，该任务需要并发访问共享数据。例如，多线程应用可以对在内存中的数据库并发地进行读写。在这样的情况下，线程必须共享不是线程专有的记录和表。如果使用线程专有存储来存储此数据库，线程就不能共享这些数据。因而，对数据库记录的访问必须通过同步原语（例如，互斥体）来控制，以使线程能在共享数据上协作。
- 维护物理*和*逻辑上都分离的数据要更为直观和高效。例如，通过将数据作为参数显式地传递给所有方法，有可能使线程访问仅在每个线程中可见的数据。在这样的情况下，线程专有存储模式有可能是不必要的。

5.4 结构和参与者

图5-2演示线程专有存储中的以下参与者的结构：

应用线程（Application Thread）

- 应用线程使用TS Object Proxy（TS对象代理）来访问驻留在线程专有存储中的TS Object。如5.9所示，线程专有存储模式的实现可以使用“*灵巧指针*”（smart pointer）来隐藏TS Object Proxy，以使应用看起来像是在直接访问TS Object。

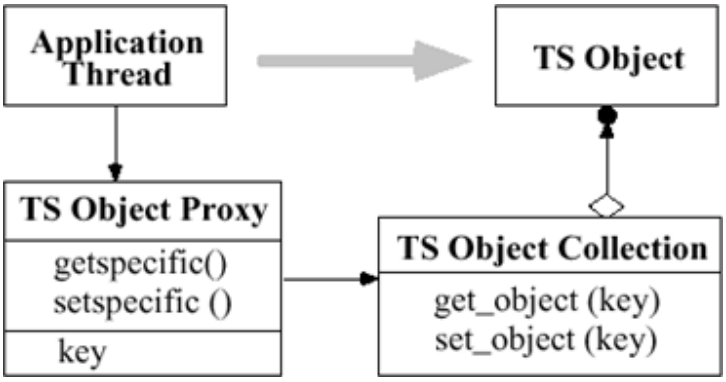

图5-2 线程专有存储模式中的参与者的结构

线程专有对象代理（TS Object Proxy）（errno宏）

- TS Object Proxy定义TS Object的接口。它负责通过getspecific和setspecific方法来为每个应用线程提供对一个唯一的对象的访问。例如，在5.2的错误处理例子中，errno TS Object是一个int。
  一个TS Object Proxy的实例负责一种类型的对象，也就是，它为所有访问代理的线程而充当对线

程专有TS Object进行访问的中介。例如，多个线程可以使用同一个TS Object Proxy来访问线程专有的errno值。代理所存储的key（专有钥）值是由TS Object Collection（TS对象集合）在代理被创建时分配的，并由getspecific和setspecific方法传递给集合。
TS Object Proxy的目的是隐藏key和TS Object Collection。没有代理，Application Thread必须显式地获取集合和使用专有钥。如5.9所示，线程专有存储的大多数细节可以通过TS Object Proxy的灵巧指针来完全隐藏。

线程专有对象（TS Object）（* _errno（）value）

- TS Object是特定线程的线程专有对象的实例。例如，线程专有的errno是int类型的对象。它由TS Object Collection管理，并且只能通过TS Object Proxy来访问。

线程专有对象集合（TS Object Collection）

- 在复杂的多线程应用中，线程的errno值可以是驻留在线程专有存储中的许多类型的数据中的一种。因而，要获取线程专有错误数据的线程，必须使用一个key。这个key必须与errno相关联，以允许线程访问TS Object Collection中的正确条目。
TS Object Collection含有一组与特定线程相关联的所有线程专有对象，也就是，每个线程都有唯一的TS Object Collection。TS Object Collection将key映射到线程专有的TS Object。通过get_object(key)和set_object(key)方法，TS Object Proxy使用key来从TS对象集合中获取一个特定的TS对象。

5.5 协作

图5-3中的交互图演示线程专有存储模式中的参与者之间的以下协作：

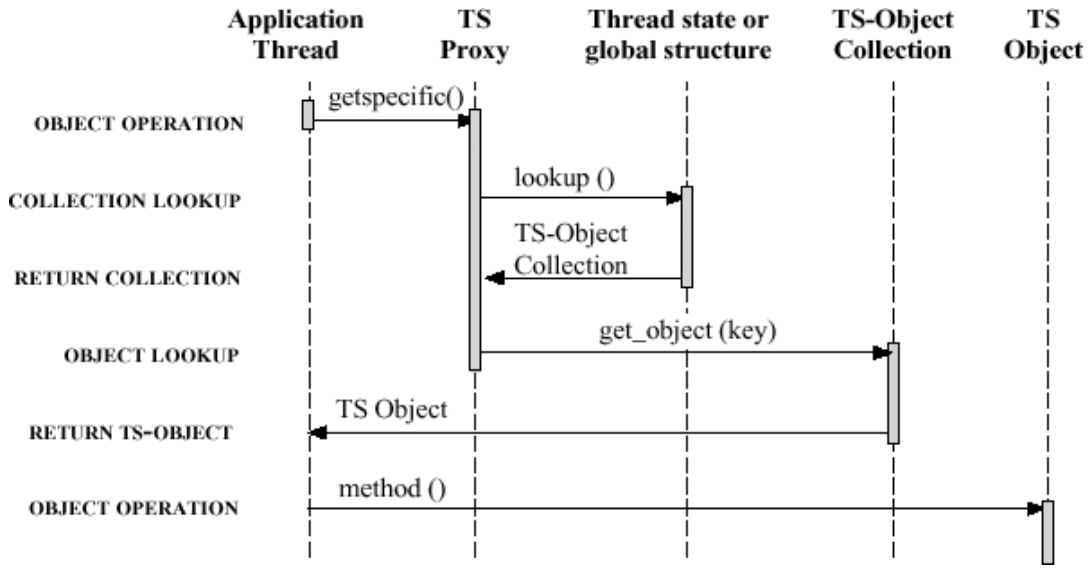

图5-3 线程专有存储模式中的参与者之间的交互

- **定位**TS Object Collection：每个Application Thread中的方法在TS Object Proxy上调用getspecific和setspecific方法来访问TS Object Collection，后者存储在线程中、或在通过线程ID进行索引的全局结构中。

- **从线程专有存储中获取**TS Object：一旦TS Object Collection被定位，TS Object Proxy就使用它的key来从集合中获取恰当的TS Object。

- **设置/获取**TS Object**的状态**：在这里，应用线程使用普通的C++方法调用来在TS Object上进行操作。无需进行锁定，因为对象通过一个指针来引用，而该指针仅能在调用线程中进行访问。

5.6 效果

5.6.1 好处

使用线程专有存储有若干好处，包括：

效率：线程专有存储可实现成无需对线程专有数据进行锁定。例如，通过将errno放入线程专有存储中，每个线程都可以可靠地设置和测试该线程中的方法的完成状态，而无需使用复杂的同步协议。这排除了线程中共享数据的锁定开销，比起获取和释放互斥体要更为迅捷[1]。

易于使用：对于应用程序员来说，线程专有存储使用起来很简单，因为系统开发者可以通过数据抽象或宏来使线程专有存储的使用在源码级完全透明化。

5.6.2 缺点

使用线程专有存储还有着以下缺点：

它鼓励了（线程安全的）全局变量的使用：许多应用不要求多个线程通过公用访问点来访问线程专有的数据。如果是这样，数据的存储应使只有拥有该数据的线程可对它进行访问。例如，考虑一个网络服务器，它使用工作者线程池来处理来自客户的请求。这些线程可能会记录所执行服务的数量和类型。这个日志机制可以作为使用线程专有存储的全局Logger对象来访问。但是，更简单的方法是将每个工作者线

程表示为主动对象[2]，并在其内部存储Logger的实例。在这样的情况下，只要将Logger作为参数传递给主动对象中的所有函数，对Logger的访问就不会产生额外开销。

它隐藏了系统的结构：线程专有存储的使用隐藏了应用中的对象之间的关系，可能会导致应用更难被理解。如附录A.2所描述的，在某些情况下，显式地表现对象间的关系可以消除对线程专有存储的需要。

5.7 实现

线程专有存储模式可以通过多种途径来实现。这一部分解释实现该模式所需的每一步骤。这些步骤被总结如下：

1. **实现**TS Object Collection：如果OS不提供线程专有存储的实现，它可以使用任何一种可用以维护TS Object Collection中数据结构的一致性的机制来实现。
2. **封装线程专有存储的细节：**如5.8所示，线程专有存储的接口通常是弱类型和易错的。因而，一旦有线程专有存储的实现允许，就使用C++编程语言特性（比如模板和重载）来将线程专有存储的低级细节隐藏在OO API后面。

这一部分的余下部分描述怎样实现低级的线程专有存储API。5.8提供了完整的示例代码，5.9检查通过C++包装来封装低级的线程专有存储API的若干方法。

5.7.1 实现TS Object Collection

图5-2中所示的TS Object Collection含有所有属于某个特定线程的TS Object。该集合可以使用指向TS Object的指针表来实现，这些对象通过key进行索引。线程必须通过key来在访问线程专有对象之前定位TS Object Collection。因此，第一个设计挑战就是决定怎样定位和存储TS Object Collection。

TS对象集合可通过以下两种方式存储：（1）外在于所有线程，或（2）内在于每一线程。对每种方法的描述和评估如下：

1. **外在于所有线程：**该方法定义每个线程的ID到它的TS对象集合表（如图5-4所示）的一个全局映射。定位适当的集合可能需要使用读者/作者锁来防止竞争状态。但是一旦集合被定位，就不再需要另外的锁定了，因为只有一个线程可在其TS Object Collection中活动。

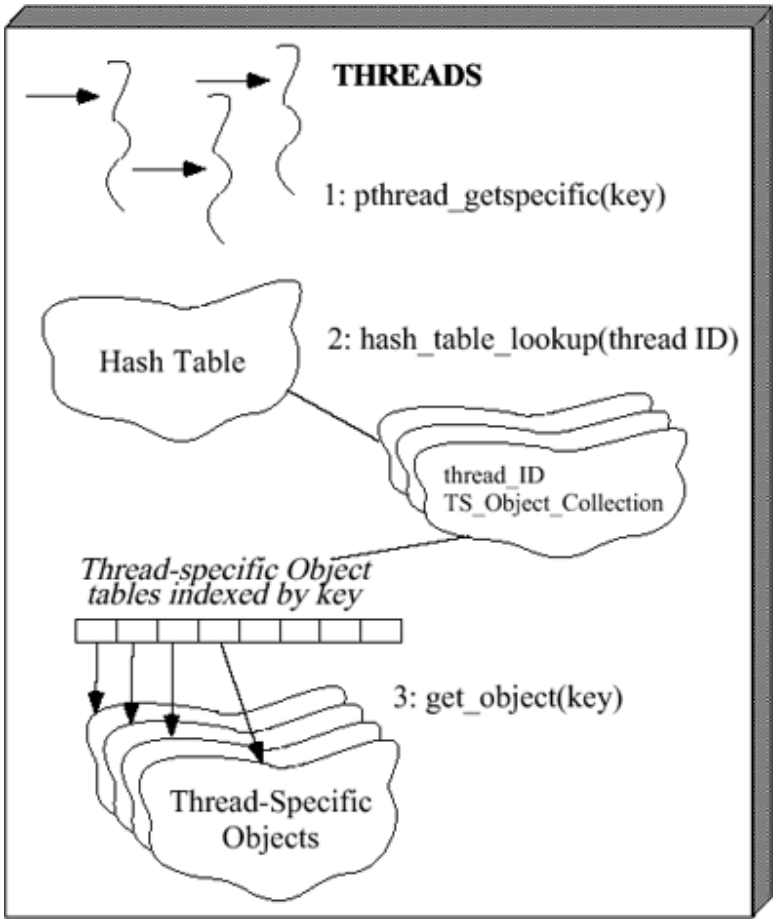

图5-4 线程专有存储的外部实现

2. **内在于每个线程：**该方法要求进程中的每个线程与它的其他内部状态（比如运行时线程栈、程序计数器、通用寄存器和线程ID）一起存储一个TS对象集合。当线程访问线程专有对象时，通过把相应的key作为线程内部的TS Object Collection的索引来获取该对象（如图5-5所示）。该方法不需要额外的锁定。

在外部和内部实现方案之间选择需要开发者进行以下权衡：

定长 vs. **变长**TS Object Collection：对于外部和内部两种实现，如果线程专有钥的范围相对较小，TS Object Collection可作为定长数组存储。例如，POSIX Pthread标准定义了专有钥的最小数目_POSIX_THREAD_KEYS_MAX，遵从标准的实现必须对其加以支持。如图5-5所示，如果长度是固定的（例如，128个专有钥，这是POSIX的缺省值），通过简单地使用对象的专有钥来在TS Object Collection数组中检索，查找时间可为O(1)。

但是，线程专有钥的范围也可以很大。例如，Solaris线程对专有钥的数目没有预定义的限制。因此，Solaris使用了一种变长数据结构，这可能会增加管理TS Object Collection所需的时间。

定长 vs. **变长的线程**ID**到**TS Object Collection**的映射：**线程ID的范围可以从很小到很大的值。这对于内部实现来说没有任何问题，因为线程ID隐含地与相应的包含在线程状态中的TS Object Collection关联在一起。

但是，对于外部实现，要使每个可能的线程ID值都在定长数组中有相应条目可能是不现实的。相反，让线程使用一个动态数据结构来将线程ID映射到TS Object Collection，在空间上要更为经济。例如，一种方法是在线程ID上使用哈希函数，以获得在哈希表桶中的一个偏移；在哈希表桶中含有一系列二元组，将线程ID映射到它们相应的TS Object Collection（如图5-4所示）。

全局 vs. 局部TS对象集合：内部方法*局部地存储*TS Object Collection，而外部方法*全局地存储*它们。取决于外部表的实现，全局定位可允许线程访问其他线程的TS Object Collection。尽管这看起来废止了*线程专有存储*的整个出发点，如果线程专有存储实现通过回收使用无用的专有钥来提供自动的垃圾收集的话，这还是有用的。该特性对于限制专有钥数目为较小值的实现特别重要（例如，Windows NT有着每个进程64个专有钥的限制）。

但是，使用外部表增加了每个线程专有对象的访问时间，因为如果可全局访问的表被修改的话（例如，创建新专有钥时），需要使用同步机制（比如读者/作者锁）来避免竞争状态。在另一方面，将TS Object Collection局部地保持在每个线程的状态中需要每个线程有更多的存储，而总的内存消耗也不会减少。

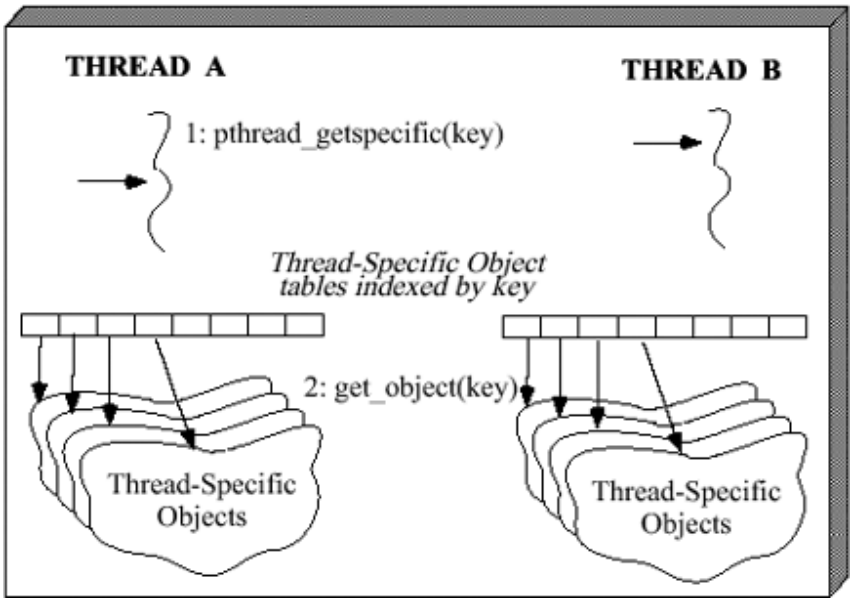

图5-5 线程专有存储的内部实现

5.8 示例代码

5.8.1 实现POSIX Pthreads线程专有存储API

下面的代码演示在使用专有钥定长数组将TS Object“内部地”存储在线程中时，线程专有存储是怎样实现的。这个例子是从POSIX Pthreads[4]的一个公开可用的实现[3]改编而来。

下面所示的thread_state结构含有线程的状态：

```
struct thread_state

{

// The thread-specific error number.

int errno_;


// Thread-specific data values.

void *key_[_POSIX_THREAD_KEYS_MAX];


// ... Other thread state.

};
```

除了errno和线程专有存储指针的数组，该结构还包括了一个指向线程的栈和空间的指针；这些栈和空间用于存储在上下文切换过程中存储/恢复的数据（例如，程序计数器）。

对于一个特定的线程专有对象，所有线程用同一个专有钥值来设置或取回线程专有的值。例如，如果Logger对象正在被登记以跟踪线程专有的日志属性，线程专有的Logger代理将分配得到某个专有钥值N。所有线程都将使用这个值N来访问它们的线程专有日志对象。目前正在使用的专有钥的总数目可相对于所有线程全局地存储。如下所示：

```
typedef int pthread_key_t;


// All threads share the same key counter.

static pthread_key_t total_keys_ = 0;
```

每次有新的线程专有钥被请求时，total_keys_count都会自动增长，如下面的pthread_key_create函数所示：

```
// Create a new global key and specify

// a "destructor" function callback.

Int pthread_key_create (pthread_key_t *key,

void (*thread_exit_hook) (void *))

{
```

```
if (total_keys_ >= _POSIX_THREAD_KEYS_MAX)

{

       // pthread_self() refers to the context of the

       // currently active thread.

       pthread_self ()->errno_ = ENOMEM;

       return -1;

}


thread_exit_hook_[total_keys_] = thread_exit_hook;

*key = total_keys_++;

return 0;

}
```

pthread_key_create函数分配一个新的专有钥值，唯一地标识一个线程专有数据对象。此外，它还允许应用将一个thread_exit_hook与一个专有钥关联。该挂钩（hook）是一个函数指针，会被自动调用，当（1）线程退出时，以及（2）有线程专有对象登记专有钥时。指向“线程退出挂钩”的函数指针的数组可被全局地存储。如下所示：

```
// Exit hooks to cleanup thread-specific keys.

static void (*thread_exit_hook_[_POSIX_THREAD_KEYS_MAX]) (void);
```

下面的pthread_exit函数演示线程退出挂钩函数是怎样在pthread_exit的实现中被调用的：

```
// Terminate the thread and call thread exit hooks.

void pthread_exit (void *status)

{

// ...

for (i = 0; i < total_keys; i++)

       if (pthread_self ()->key_[i] && thread_exit_hook_[i])

              // Indirect pointer to function call.

              (*thread_exit_hook_[i]) (pthread_self ()->key_[i]);


// ...

}
```

应用可为各个线程专有数据对象登记不同的函数，但是对于每个对象、会为每个线程调用同一个函数。登记动态分配的线程专有存储对象是一种常见的使用方法。因此，线程退出挂钩通常看起来是这样的：

```
static void cleanup_tss_Logger (void *ptr)
{
// This cast is necessary to invoke
// the destructor (if it exists).
delete (Logger *) ptr;
}
```

该函数释放一个动态分配的Logger对象。

pthread_setspecific函数为调用线程将value绑定到给定的key：

```
// Associate a value with a data key
// for the calling thread.
int pthread_setspecific (int key, void *value)
{
if (key < 0 || key >= total_keys) {
       pthread_self ()->errno_ = EINVAL;
       return -1;
}

pthread_self ()->key_[key] = value;
return 0;
}
```

同样地，pthread_getspecific为调用线程把绑定到给定Key的数据存储到value中：

```
// Retrieve a value from a data key
// for the calling thread.
int pthread_getspecific (int key, void **value)
{
if (key < 0 || key >= total_keys)
{
```

```
        pthread_self ()->errno_ = EINVAL;
        return -1;
}


*value = pthread_self ()->key_[key];
return 0;
}
```

因为数据存储在每个线程的内部状态中，这些函数不需要任何额外的锁来访问线程专有数据。

5.8.2 在应用中使用线程专有存储

下面的例子演示怎样在可从多于一个线程调用的一个C函数中，使用来自POSIX Pthread规范的线程专有存储API，而又*无须*显式地调用初始化函数：

```
// Local to the implementation.
static pthread_mutex_t keylock = PTHREAD_MUTEX_INITIALIZER;
static pthread_key_t key;
static int once = 0;


void *func (void)
{
void *ptr = 0;


// Use the Double-Checked Locking pattern
// (described further below) to serialize
// key creation without forcing each access
// to be locked.


if (once == 0)
{
        pthread_mutex_lock (&keylock);
        if (once == 0)
        {
                // Register the free(3C) function
```

```
                // to deallocation TSS memory when
                // the thread goes out of scope.
                pthread_key_create (&key, free);
                once = 1;
        }
        pthread_mutex_unlock (&keylock);
}

pthread_getspecific (key, (void **) &ptr);
if (ptr == 0)
{
        ptr = malloc (SIZE);
        pthread_setspecific (key, ptr);
}

return ptr;
}
```

5.8.3 评估

上面的解决方案直接在应用代码中调用线程专有库函数（比如pthread_getspecific和pthread_setspecific）。但是这些直接用C编写的API有以下局限：

- **不可移植：**POSIX Pthreads、Solaris线程和Win32线程非常类似。但是，Win32线程的语义有着微妙的不同，因为它们不提供一种可靠的方法来在线程退出时清理在线程专有存储中分配的对象。而且，在Solaris线程中没有API用于删除专有钥。这使得开发者难以在UNIX和Win32平台间编写可移植代码。
- **难以使用：**尽管省略了错误检查，在5.8.2中的func例子所演示的锁定操作仍然是复杂而又不直观的。该代码是“双重检查锁定模式”（Double-Checked Locking pattern）[5]的C实现。将这个C实现与5.9.2.1中的C++版本相比较，以观察使用C++包装所带来的更多的简单性、清晰性和类型安全性是富有启发意义的。
- **非类型安全的：**POSIX Pthreads、Solaris和Win32线程专有存储接口将指向线程专有对象的指针作为void *存储。尽管这种方法很灵活，它很容易造成错误，因为void *消除了类型安全性。

5.9 变种

5.8演示了怎样通过POSIX pthread接口实现和使用线程专有存储模式。但是，所得到的解决方案不可移植、难以使用，且不是类型安全的。为了克服这些局限，可以开发另外的类和C++包装来健壮地以一种类型安全的方式编写线程专有存储。

这一部分演示怎样使用C++包装来封装POSIX Pthreads、Solaris或Win32线程提供的低级线程专有存储机制。5.9.1描述怎样通过硬编码的C++包装来封装POSIX Pthread库接口，5.9.2描述一种使用C++模板包装的更为通用的解决方案。用于每种可选方法的例子是5.6.2描述的Logger抽象的一个变种。

5.9.1 硬编码的C++包装

使一个类的所有实例成为线程专有的一种方法是直接使用线程专有库例程。实现该方法所需的步骤描述如下。错误检查已被省略到最少以节约空间。

5.9.1.1 定义线程专有状态信息

第一步是决定必须在线程专有存储中存取的对象状态信息。例如，Logger可能有以下状态：

```
class Logger_State

{

public:

int errno_;

// Error number.


int line_num_;

// Line where the error occurred.


// ...

};
```

每个线程都将拥有它自己的一份这些状态信息的拷贝。

5.9.1.2 定义外部类接口

下一步是定义被所有应用线程使用的外部类接口。下面的Logger外部类接口看起来就像是一个平常的非线程专有的C++类：

```
class Logger

{

public:

// Set/get the error number.

int errno (void);

void errno (int);


// Set/get the line number.

int line_num (void);

void line_num (int);


// ...

};
```

5.9.1.3 定义线程专有的助手函数

该步骤使用线程库提供的线程专有存储函数来定义一个助手函数，它返回一个指向适当的线程专有存储的指针。这个助手函数通常执行以下步骤：

1. **专有钥初始化：**为每个线程专有对象初始化一个专有钥，并使用此专有钥来存/取一个指向动态分配内存的线程专有指针；此内存含有内部结构的实例。该代码可以被实现如下：

```
class Logger

{

public:

// ... Same as above ...


protected:

Logger_State *get_tss_state (void);


// Key for the thread-specific error data.

pthread_key_t key_;


// "First time in" flag.
```

```
int once_;
};

Logger_State *Logger::get_tss_state (void)
{
// Check to see if this is the first time in
// and if so, allocate the key (this code
// doesn't protect against multi-threaded
// race conditions...).
if (once_ == 0)
{
pthread_key_create (this->key_, free);
once_ = 1;
}

Logger_State *state_ptr;

// Get the state data from thread-specific
// storage. Note that no locks are required...
pthread_getspecific (this->key_, (void **) &state_ptr);

if (state_ptr == 0)
{
state_ptr = new Logger_State;
pthread_setspecific (this->key_, (void *) state_ptr);
}

// Return the pointer to thread-specific storage.
return state_ptr;
};
```

2. **获取指向线程专有存储对象的指针：**外部接口中的每个方法都调用get_tss_state助手函数来获取指向驻留在线程专有存储中的Logger_State对象的指针。如下所示：

```
int Logger::errno (void)
{
```

```
        return this->get_tss_state ()->errno_;

        }
```

3. **执行正常操作：**一旦外部接口方法有了该指针，应用就可以对线程专有对象进行操作，就如同它是平常的（也就是，非线程专有的）C++对象：

```
Logger logger;


int recv_msg (HANDLE socket, char *buffer, size_t bufsiz)

{

if (recv (socket, buffer, bufsiz, 0) == -1)

{

        logger->errno () = errno;

        return -1;

}

// ...

}


int main (void)

{

// ...

if (recv_msg (socket, buffer, BUFSIZ) == -1

        && logger->errno () == EWOULDBLOCK)

        // ...

}
```

5.9.1.4 对硬编码包装的评价

使用硬编码包装的优点是它将应用与线程专有库函数的知识屏蔽开来。该方法的缺点是它不能促进复用性、可移植性，或是灵活性。特别地，对于每个线程专有类，开发者都需要在类中重新实现线程专有助手方法。

而且，如果应用被移植到有着不同的线程专有存储API的平台上，就必须改变在每个线程专有类中的代码，以使用新的线程库。此外，直接改变线程专有类使得程序难以变更线程策略。例如，将一个线程专有类变为全局类需要对代码进行侵入性的变动，从而降低了灵活性和复用性。特别地，每次对对象内部的状态信息的访问都将要求改变用于从线程专有存储取回该状态的助手方法。

5.9.2 C++模板包装

一种更为可复用、可移植和灵活的方法是实现TS Object Proxy模板，负责所有的线程专有方法。该方法允许类与线程专有存储怎样实现的知识去耦合。通过定义称为TSS的代理类，该解决方案改善了代码的可复用性、可移植性和灵活性。如下所示，该类是一个模板，通过其对象驻留在线程专有存储中的类来参数化：

```
// TS Proxy template

template <class TYPE>

class TSS

{

public:

// Constructor.

TSS (void);


// Destructor

?TSS (void);


// Use the C++ "smart pointer" operator to

// access the thread-specific TYPE object.

TYPE *operator-> ();


private:

// Key for the thread-specific error data.

pthread_key_t key_;


// "First time in" flag.

int once_;


// Avoid race conditions during initialization.

Thread_Mutex keylock_;


// Cleanup hook that deletes dynamically

// allocated memory.

static void cleanup_hook (void *ptr);

};
```

该类中的方法描述如下。和前面一样，错误检查已被省略到最少以节约空间。

5.9.2.1 C++委托操作符（Delegation Operator）

通过重载C++委托操作符（操作符->），应用可以调用TSS代理上的方法，就好像是在调用目标类一样。在此实现中使用的C++委托操作符控制所有对类TYPE的线程专有对象的访问。操作符->方法受到了来自C++编译器的特殊对待。如5.9.2.3所述，它先从线程专有存储那里获取一个指向适当的TYPE的指针，随后就重新委托原来在其上调用的方法。

TSS类中的大多数工作都在下面所示的操作符->方法中执行：

```
template <class TYPE> TYPE *
TSS<TYPE>::operator-> ()
{
TYPE *tss_data = 0;

// Use the Double-Checked Locking pattern to
// avoid locking except during initialization.
// First check.
if (this->once_ == 0)
{
       // Ensure that we are serialized (constructor
       // of Guard acquires the lock).
       Guard <Thread_Mutex> guard (this->keylock_);

       // Double check
       if (this->once_ == 0)
       {
              pthread_key_create (&this->key_, &this->cleanup_hook);

              // *Must* come last so that other threads
              // don't use the key until it's created.
              this->once_ = 1;
       }
```

```
        // Guard destructor releases the lock.
}

// Get the data from thread-specific storage.
// Note that no locks are required here...
pthread_getspecific (this->key_, (void **) &tss_data);

// Check to see if this is the first time in
// for this thread.
if (tss_data == 0)
{
        // Allocate memory off the heap and store
        // it in a pointer in thread-specific
        // storage (on the stack...).
        tss_data = new TYPE;

        // Store the dynamically allocated pointer in
        // thread-specific storage.
        pthread_setspecific (this->key_, (void *) tss_data);
}

return tss_data;
}
```

TSS模板是一个代理，它透明地将普通C++类转换为类型安全、线程专有的类。它结合了操作符->和其他一些C++特性，像模板、内联和重载。它还利用了像双重检查锁定优化[5]和代理[6, 7]这样的模式。

在代码中，双重检查锁定优化模式用于在操作符->中两次测试once_标志。尽管多个线程可以同时访问TSS的同一实例，仅有一个线程可以合法地创建一个专有钥（也就是，通过pthread_key_create）。随后所有线程将使用这一专有钥来访问参数化类TYPE的线程专有对象。因此，操作符->使用Thread_Mutex keylock_来确保仅有一个线程执行pthread_key_create。

第一个获取keylock_的线程设置once_为1，所有调用操作符->的后续线程将发现once != 0，于是跳过初始化步骤。对once_的第二次测试处理这样的情况：在第一个线程设置once_为1之前，多个并行执行的线程在keylock_处排队等候。在这种情况下，当其他排队等待的线程最终获得互斥体keylock_，它们会发现once_等于1，就不会执行pthread_key_create。

一旦key_被创建，不再需要有进一步的锁定来访问线程专有数据。这是因为pthread_{getspecific, setspecific}函数从调用线程的状态处获取类TYPE的TS Object，而此线程状态是独立于其他线程的。

除了减少锁定开销，上面所示的类TSS的实现将应用代码与对象是专有于调用线程的这一事实屏蔽开来。为达成这一点，该实现使用了像模板、操作符重载和委托操作符（也就是，操作符->）这样的C++特性。

5.9.2.2 构造器和析构器

TSS类的构造器是很小的，它只是简单地初始化局部实例变量：

```
template <class TYPE>

TSS<TYPE>::TSS (void): once_ (0), key_ (0) {}
```

注意我们没有在构造器中分配TSS专有钥或是一个新的TYPE实例。这样设计有若干原因：

- **线程专有存储语义**：最初创建TSS对象的线程（例如，主线程）常常不是使用该对象的线程（例如，工作者线程）。因此，在构造器中预先初始化一个新的TYPE并没有好处，因为此实例只能被主线程访问。
- **延期的初始化：**在某些OS平台上，TSS专有钥是有限的资源。例如，Windows NT仅允许每个进程总共有64个TSS专有钥。因此，不到绝对需要的时候，不应分配专有钥。相反，初始化被延期到操作符->第一次被调用时。

  TSS析构器给我们带来若干棘手的设计问题。显而易见的解决方案是在操作符->中释放所分配的TSS专有钥。但是，这一方法有若干问题：

  - **特性缺乏：**Win32和POSIX pthreads定义了函数来释放TSS专有钥。但是，Solaris没有。因此，很难编写一个可移植的包装。
  - **竞争状态：**Solaris线程不提供函数来释放TSS专有钥的主要原因是实现起来很昂贵。问题在于每个线程都分别维护通过同一个专有钥引用的对象。只有在所有这些线程退出、且内存被回收后才能安全地释放该专有钥。

作为上面所提到的问题的结果，我们的析构器是一个空操作。

```
template <class TYPE>

TSS<TYPE>::?TSS (void)

{

}
```

cleanup_hook是一个静态方法，它在删除其ptr参数之前，将其强制转换到适当的TYPE *。

```
template <class TYPE> void

TSS<TYPE>::cleanup_hook (void *ptr)

{

// This cast is necessary to invoke

// the destructor (if it exists).

delete (TYPE *) ptr;

}
```

这确保了在线程退出时，每个线程专有对象的析构器都会被调用。

5.9.2.3 用例

下面的方案基于C++模板包装来解决我们一直在讨论的例子：被多个工作者线程访问的线程专有存储Logger。

```
// This is the "logically" global, but

// "physically" thread-specific logger object,

// using the TSS template wrapper.

static TSS<Logger> logger;


// A typical worker function.

static void *worker (void *arg)

{

// Network connection stream.

SOCK_Stream *stream = static_cast <SOCK_Stream *> arg;


// Read from the network connection

// and process the data until the connection

// is closed.

for (;;)

{

       char buffer[BUFSIZ];

       int result = stream->recv (buffer, BUFSIZ);
```

```
        // Check to see if the recv() call failed.

        if (result == -1)

        {

                if (logger->errno () != EWOULDBLOCK)

                        // Record error result.

                        logger->log ("recv failed, errno = %d",
                        logger->errno ());

        }

        else

                // Perform the work on success.

                process_buffer (buffer);

}

}
```

考虑上面对logger->errno的调用。C++编译器用两个方法调用来替换这一调用。第一个是对TSS::操作符->的调用，它返回一个驻留在线程专有存储中的Logger实例。随后编译器生成第二个方法，调用前面的调用返回的Logger对象的errno方法。在这种情况下，TSS作为一个代理，允许应用访问和操作线程专有的错误值，就如同它们是平常的C++对象一样。

上面的Logger例子是一个好范例，在其中使用逻辑上的全局访问点是有益的。因为worker函数是全局的，线程要同时管理Logger对象的物理*和*逻辑的分离并不那么简明。相反，线程专有的Logger允许多个线程使用单个逻辑访问点来操作物理上分离的TSS对象。

5.9.2.4 评估

基于C++操作符->的TSS代理设计有以下好处：

- **最大化代码复用：**通过使线程专有方法与特定的应用类（也就是，形式参数类TYPE）去耦合，不再需要重写微妙的线程专有钥的创建和分配逻辑。
- **提高可移植性：**将应用移植到其他线程库（比如Win32中的TLS接口）只需要改变TSS类，而不是所有使用此类的应用。
- **更大的灵活性和透明性：**将一个类变为线程专有类（或相反），仅需要改变此类对象的定义方式。这可以在编译时被决定，如下所示：

```
#if defined (_REENTRANT)

static TSS<Logger> logger;

#else

// Non-MT behavior is unchanged.

Logger logger;
```

```
#endif /* REENTRANT */
```

注意不管使用的是线程专有还是非线程专有形式的Logger，Logger的使用方式都保持不变。

5.10 已知应用

下面是线程专有存储模式的已知应用：

- 在支持POSIX和Solaris线程API的OS平台上实现的errno机制是被广泛使用的线程专有存储模式的例子[1]。此外，与Win32一起提供的C运行时库支持线程专有的errno。Win32 GetLastError/SetLastError也实现了线程专有存储模式。
- 在Win32操作系统中，窗口属于线程[8]。每个拥有窗口的线程都有一个私有的消息队列，OS会在其中放入用户接口事件。获取等待处理的下一消息的API调用使下一个消息从调用线程的消息队列中出队，而该消息队列就驻留在线程专有存储中。
- OpenGL[9]是一个用于渲染三维图形的C API。程序根据多边形来渲染图形；多边形通过反复调用glVertex函数、以传递多边形的每一顶点给库来描述。在顶点被传递给库之前设置的状态变量精确地决定OpenGL在接收到顶点时如何进行绘制。该状态在OpenGL库中、或是在图形卡上作为封装的全局变量存储。在Win32平台上，OpenGL库为每个使用该库的线程在线程专有存储中维护一组唯一的状态变量。
- 线程专有存储被用于在ACE网络编程工具包[10]中实现它的错误处理方案，该方案与5.9.2.3描述的Logger方法相类似。此外，ACE还实现了5.9.2描述的线程安全的线程专有存储模板包装。

5.11 相关模式

用线程专有存储实现的对象常常被用作“per-thread”的单体（Singleton）[7]，例如，errno就是一个“per-thread”单体。但是，并非所有线程专有存储的使用都是单体，因为线程可以拥有从线程专有存储中分配的一种类型的多个实例。例如，在ACE[10]中实现的每一个Task对象都在线程专有存储中存储一个清理挂钩。

5.8所示的TSS模板类被用作代理，将库、构架和应用与OS线程库提供的线程专有存储的实现屏蔽开来。

双重检查锁定优化模式[5]通常被用于这样的应用：它们利用线程专有存储模式来避免约束线程专有存储钥的初始化顺序。

5.12 结束语

由于防止竞争状态和死锁所需的额外的并发控制协议，使现有应用多线程化常常会显著地增加软件的复杂性[11]。通过允许多线程使用一个逻辑上的全局访问点来获取线程专有数据，而又不给每次访问

带来锁定代价，线程专有存储模式减轻了一些同步开销和编程复杂性。

应用线程使用TS Object Proxy来访问TS Object。代理委托TS Object Collection来获取相应于每个应用线程的对象。这确保了不同的应用线程不会共享同一个TS Object。

5.9.2显示怎样实现线程专有存储模式的TS Object Proxy，以确保线程通过强类型的C++类接口来访问只属于它们自己的数据。与其他模式（比如代理、单体和双重检查锁定）和C++语言特性（比如模板和操作符重载）相结合，可实现TS Proxy，使得使用线程专有存储模式的对象可像传统对象一样被处理。

感谢

感谢Peter Sommerlad和Hans Rohnert对本论文早期版本的富有洞察力的意见。

参考文献

[1] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[2] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[3] F. Mueller, "A Library Implementation of POSIX Threads Under UNIX," in *Proceedings of the Winter USENIX Conference*, (San Diego, CA), pp. 29－42, Jan. 1993.

[4] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.

[5] D. C. Schmidt and T. Harrison, "Double-Checked Locking - An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[8] C. Petzold, *Programming Windows 95*. Microsoft Press, 1995.

[9] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release* 1. Addison-Wesley, Reading, MA, 1993.

[10] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[11] J. Ousterhout, "Why Threads Are A Bad Idea (for most purposes)," in *USENIX Winter Technical Conference*,(San Diego, CA), USENIX, Jan. 1996.

[12] H. Mueller, "Patterns for Handling Exception Handling Successfully," *C++ Report*, vol. 8, Jan. 1996.

[13] N. Pryce, "Type-Safe Session: An Object-Structural Pattern," in *Submitted to the 2nd European Pattern Languages of Programming Conference*, July 1997.

[14] J. Gosling and F. Yellin, *The Java Application Programming Interface Volume 2: Window Toolkit and Applets*. Addison-Wesley, Reading, MA, 1996.

附录A 可选方案

在实践中，线程专有存储通常用于为面向对象软件解决下面两种使用情况：

1. 在模块间隐式地传递信息（例如，错误信息）。
2. 将遗留的以过程风格编写的单线程软件改编到现代的多线程操作系统和编程语言。

但是，对于#1使用情况，避免使用线程专有存储常常是一个好主意，因为它可能会增强模块间的耦合并降低可复用性。例如，对于错误处理的情况，常常可以使用A.1描述的异常来避开线程专有存储。

除非重新设计，对于#2使用情况，不能不使用线程专有存储。但是在设计新软件时，常常可以使用如下所描述的异常处理、显式的组件间通信上下文或对象化线程来避开线程专有存储。

A.1 异常处理

在模块间报告错误的一种优雅方法是使用异常处理。许多现代语言，比如C++和Java，使用异常处理来作为错误报告的机制。它还被用于一些操作系统中，比如Win32。例如，下面的代码演示一种假想的OS，其系统调用会扔出异常。

```
void *worker (SOCKET socket)
{
// Read from the network connection
// and process the data until the connection
// is closed.
for (;;)
{
       char buffer[BUFSIZ];

       try
       {
              // Assume that recv() throws exceptions.
              recv (socket, buffer, BUFSIZ, 0);

              // Perform the work on success.
              process_buffer (buffer);
```

```
        }
        catch (EWOULDBLOCK)
        {
                continue;
        }
        catch (OS_Exception error)
        {
                // Record error result in thread-specific data.
                printf ("recv failed, error = %s", error.reason);
        }
}
}
```

使用异常处理有若干好处：

- **它是可扩展的：**现代的OO语言通过一些尽量不侵犯现有接口和使用的特性（比如使用继承来定义异常类层次）来便利异常处理策略和机制的扩展。
- **它干净地使错误处理和正常处理得以去耦合：**例如，错误处理信息不是被显式地传递给操作的。而且，应用不会由于没有完成对函数返回值的检查而偶然地“忽略”了一个异常。
- **它可以是类型安全的：**在强类型的语言中，比如C++和Java，异常以一种强类型的方式被扔出和捕捉，以增强错误处理代码的组织和正确性。与显式地检查线程专有错误值相反，编译器会确保对于每种类型的异常，执行正确的处理方法。

但是，使用异常处理也有若干缺点：

- **它并非普遍可用的：**并非所有语言都提供异常处理，而许多C++编译器也并没有实现异常。同样地，如果一种OS提供异常处理服务，它们必须被语言扩展支持，从而也就降低了代码的可移植性。
- **它使多种语言的使用复杂化：**因为语言以不同的方式实现异常，或者根本不实现异常，在以不同语言编写的组件扔出异常时，可能会很难将它们集成在一起。使用整数值或结构来报告错误信息提供了一种普遍可用的解决方案。
- **它使资源管理复杂化：**例如，由于增加了C++代码块中退出路径的数目[12]，而导致这样的问题。如果语言或编程环境不支持垃圾回收，就必须付出努力来确保动态分配的对象在异常扔出时被删除。
- **它有时间和/或空间效率低下的可能性：**异常处理的糟糕实现会带来时间和空间的过度开销，即使没有异常被扔出也是如此[12]。对于必须保持小而高效的嵌入系统来说，此开销可能会特别地成问题。

对于必须在多种平台上可移植地运行的系统级构架（比如内核级设备驱动器，或低级通信子系统）来说，异常处理的缺点会特别地成问题。对于这些类型的系统，一种更为可移植、高效和线程安全的处

理错误的方法是定义一个错误处理器抽象，显式地维护关于操作的成功或失败的信息。

A.2 组件间通信的显式上下文

线程专有存储通常用于存储“per-thread”状态，以允许库和构架中的软件组件高效地进行通信。例如，errno被用于将错误值从被调用组件传递给调用者。同样地，对OpenGL API函数的调用传递信息给OpenGL库，并被存储在线程专有状态中。通过显式地将在组件间传递的信息表示为对象，可以避免使用线程专有存储。

如果预先已经知道组件必须为它的用户存储的信息的类型，调用线程可以创建对象，并将其作为操作的一个额外参数传递给组件。另外，组件必须创建一个对象来保持上下文信息，以响应来自调用线程的请求；并在线程对组件加以使用之前将一个标识符传给线程。这些类型的对象常被称为“*上下文对象*”（context object）；软件组件按需创建的上下文对象常被称为“*会话*”（session）。

下面的错误处理方案是一个简单的例子，演示调用线程怎样创建上下文对象；该方案传递一个显式的参数给所有的操作：

```
void *worker (SOCKET socket)
{
// Read from the network connection and
// process the data until the connection
// is closed.
for (;;)
{
       char buffer[BUFSIZ];
       int result;
       int errno;

       // Pass the errno context object explicitly.
       result = recv (socket, buffer, BUFSIZ, 0, &errno);

       // Check to see if the recv() call failed.
       if (result == -1)
       {
              if (errno != EWOULDBLOCK)
                     printf ("recv failed, errno = %d", errno);
       }
       else
```

```
            // Perform the work on success.

            process_buffer (buffer);

}

}
```

组件创建的上下文对象可以使用类型安全的会话模式（Type-Safe Session Pattern）[13]来实现。在此模式中，上下文对象存储组件所需的状态，并提供一个可被多态地调用的抽象接口。组件返回一个指向该抽象接口的指针给调用线程，后者随后调用接口的操作来使用组件。

OpenGL和Java AWT库[14]（用于在诸如窗口、打印机或位图这样的设备上渲染图形）所提供接口之间的差异演示了怎样使用类型安全的会话。在AWT中，程序通过从设备请求一个GraphicsContext来在设备上进行绘制。GraphicsContext封装在设备上进行渲染所需的状态，并提供了一个接口，通过它程序可以设置状态变量，并调用绘制操作。可以动态地创建多个GraphicsContext对象，从而消除了任何保持线程专有状态的需要。

与线程局部存储和异常处理相比较，使用上下文对象有以下这些好处：

- **它更加可移植：**它不需要有可能不被普遍支持的语言特性；
- **它更为高效：**线程可以直接存储和访问上下文对象，而不必在线程专有存储表中进行查找。也不需要编译器构建额外的数据结构来处理异常；
- **它是线程安全的：**上下文对象或会话句柄可以存储在线程的栈中，这自然是线程安全的。

但是使用由调用线程创建的上下文对象也有若干缺点：

- **它是强制性的：**上下文对象必须传给所有操作，并且必须在每次操作后显式地检查。这搅乱了程序逻辑，并有可能需要改变已有的组件接口，以增加错误处理器参数。
- **增加了每次调用的开销：**每次调用都会有额外开销，因为必须给每个方法调用增加一个额外的参数，而不管是否需要该对象。尽管在某些情况下，这是可以接受的，但对于执行非常频繁的方法，开销可能会是相当显著的。相反，除非发生错误，基于线程专有存储的错误处理方案就不需要被使用。

与在调用线程中创建上下文对象相比较，使用由组件创建的会话有以下好处：

- **它较少强制性：**线程不必显式地将上下文对象作为操作的参数传给组件。编译器会安排将指向上下文对象的指针作为隐藏的this指针传给它的操作。
- **它使初始化和关闭自动化：**在线程从组件那里获取会话之前，它不能开始使用会话。于是如果多个操作在矛盾的状态中，组件就可以确保它们不会被调用。相反，如果组件使用隐藏的状态，调用者必须在调用操作之前显式地初始化库，并在组件结束后将它关闭。忘记这样做会导致含混的错误或资源浪费。
- **结构是显式的：**不同代码模块之间的关系被显式地表示为对象，这使得要理解系统的行为变得更容易。

与在调用者的栈上创建上下文对象相比较，在组件中创建它们有以下缺点：

- **分配开销：**组件必须在堆上、或是从某种封装的缓存那里分配会话对象。比起在栈上分配对象，这样做的效率常常要更为低下。

A.3 对象化线程

在面向对象语言中，应用可以显式地将线程表示为对象。线程类可以通过从一个抽象基类派生来定义，在此基类中封装了作为并发线程运行所需的状态；并调用一个实例方法来作为线程的入口。线程的入口方法可以在基类中被定义为纯虚函数，并在派生类中定义。任何所需的线程专有状态（比如会话上下文）可被定义为对象实例变量，并可为线程类的任何方法所用。对这些变量的同时访问可以通过使用语言级访问控制机制、而不是显式的同步对象来防止。

下面使用ACE Task（任务）[10]的一种变种来演示这一方法；任务可用于将线程控制和对象相关联：

```
class Task

{

public:

// Create a thread that calls the svc() hook.

int activate (void);


// The thread entry point.

virtual void svc (void) = 0;


private:

// ...

};


class Animation_Thread : public Task

{

public:
```

```
Animation_Thread (Graphics_Context *gc)

       : device_ (gc) {}


virtual void svc (void)

{

       device_->clear ();

       // ... perform animation loop...

}


private:

Graphics_Context *device_;

};
```

使用对象化线程有以下好处：

- **它更为高效**：线程无需在隐藏的数据结构中进行查找、以访问线程专有状态。
- **它不具强制性**：使用对象化线程时，指向当前对象的指针作为每个函数调用的额外参数被传递。不像显式的会话上下文，在源码中该参数是隐藏的，并由编译器自动管理，从而可使源码保持整洁。

使用对象化线程有以下缺点：

- **不容易访问线程专有存储：**只有类方法可以访问实例变量。这使得使用实例变量来在可复用库和线程间进行通信变得不直观。无论如何，以这种方式使用线程专有存储增强了组件间的耦合。一般而言，异常提供了在模块间报告错误的一种更为弱耦合的方法，尽管在像C++这样的语言里它们有着自身的陷阱和缺陷[12]。
- **额外开销：**传给每个操作的额外、隐藏的参数会带来一些开销。在执行非常频繁的函数中，这些开销可能会非常地显著。