深入理解函数内静态局部变量初始化

函数内部的静态局部变量的初始化是在函数第一次调用时执行; 在之后的调用中不会对其初始化。 在多线程环境下，仍能够保证静态局部变量被安全地初始化，并只初始化一次。下面通过代码来分析一些具体的细节：

```
void foo() {
    static Bar bar;
    // ...
}
```

通过观察 gcc 4.8.3 为上述代码生成的汇编代码， 我们可以看到编译器生成了具有如下语义的代码：

```
void foo() {
    if ((guard_for_bar & 0xff) == 0) {
        if (__cxa_guard_acquire(&guard_for_bar)) {
            try {
                Bar::Bar(&bar);
            } catch (...) {
                __cxa_guard_abort(&guard_for_bar);
                throw;
            }
            __cxa_guard_release(&guard_for_bar);
            __cxa_atexit(Bar::~Bar, &bar, &__dso_handle);
        }
    }
    // ...
}
```

虽然 bar 是 foo 的局部变量， 但是编译器在处理上与全局静态变量类似， 均存储在 bss 段 (section)，只是 bar 在汇编语言层面上的符号名称是对 foo()::bar 的编码 (mangling)，具体细节这里不做过多讨论。 guard_for_bar 是一个用来保证线程安全和一次性初始化的整型变量，是编译器生成的，存储在 bss 段。它的最低的一个字节被用作相应静态变量是否已被初始化的标志， 若为 0 表示还未被初始化，否则表示已被初始化。__cxa_guard_acquire 实际上是一个加锁的过程， 相应的 __cxa_guard_abort 和 __cxa_guard_release 释放锁。__cxa_atexit 注册在调用 exit 时或动态链接库(或共享库) 被卸载时执行的函数， 这里注册的是Bar的析构函数。值得一提的是__cxa_atexit可被用来实现atexit， atexit(func) 等价于 __cxa_atexit(func, NULL, NULL) (__cxa_atexit 函数原型： int __cxa_atexit(void (*func)(void *), void * arg, void * dso_handle))。

下面列出 __cxa_guard_acquire、 __cxa_guard_abort 和 __cxa_guard_release 这三个二进制标准接口(Itanium C++ ABI)的一种具体实现的源代码:

```
// From : http://www.opensource.apple.com/source/libcppabi/libcppabi-14/src/cxa_guard.cxx
// Headers (omitted)

// Note don't use function local statics to avoid use of cxa functions...
static pthread_mutex_t __guard_mutex;
static pthread_once_t __once_control = PTHREAD_ONCE_INIT;

static void makeRecusiveMutex() // 将 __guard_mutex 初始化为递归锁
{
    pthread_mutexattr_t recursiveMutexAttr;
    pthread_mutexattr_init(&recursiveMutexAttr);
    pthread_mutexattr_settype(&recursiveMutexAttr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&__guard_mutex, &recursiveMutexAttr);
}

__attribute__((noinline))
static pthread_mutex_t* guard_mutex()
{
    pthread_once(&__once_control, &makeRecusiveMutex); // 一次性初始化 __guard_mutex
    return &__guard_mutex;
}

// helper functions for getting/setting flags in guard_object
static bool initializerHasRun(uint64_t* guard_object)
{
    // 取最低字节作为是否已初始化的标志
    return ( *((uint8_t*)guard_object) != 0 );
}

static void setInitializerHasRun(uint64_t* guard_object)
{
    *((uint8_t*)guard_object)  = 1;
}

static bool inUse(uint64_t* guard_object)
{
    // 取次低字节作为 guard_object 是否正在被某个线程使用的标志
```

```
    return ( ((uint8_t*)guard_object)[1] != 0 );
}

static void setInUse(uint64_t* guard_object)
{
    ((uint8_t*)guard_object)[1] = 1;
}

static void setNotInUse(uint64_t* guard_object)
{
    ((uint8_t*)guard_object)[1] = 0;
}



//
// Returns 1 if the caller needs to run the initializer and then either
// call __cxa_guard_release() or __cxa_guard_abort().  If zero is returned,
// then the initializer has already been run.  This function blocks
// if another thread is currently running the initializer.  This function
// aborts if called again on the same guard object without an intervening
// call to __cxa_guard_release() or __cxa_guard_abort().
//
int __cxxabiv1::__cxa_guard_acquire(uint64_t* guard_object)
{
    // Double check that the initializer has not already been run
    if ( initializerHasRun(guard_object) ) // 如果对象已被初始化
        return 0;

    // We now need to acquire a lock that allows only one thread
    // to run the initializer.  If a different thread calls
    // __cxa_guard_acquire() with the same guard object, we want
    // that thread to block until this thread is done running the
    // initializer and calls __cxa_guard_release().  But if the same
    // thread calls __cxa_guard_acquire() with the same guard object,
    // we want to abort.
    // To implement this we have one global pthread recursive mutex
    // shared by all guard objects, but only one at a time.

    int result = ::pthread_mutex_lock(guard_mutex());
    if ( result != 0 ) {
```

```
        abort_message("__cxa_guard_acquire(): pthread_mutex_lock "
                      "failed with %d\n", result);
    }
    // At this point all other threads will block in __cxa_guard_acquire()

    // Check if another thread has completed initializer run
    if ( initializerHasRun(guard_object) ) { // 再次判断， 对象是否已被其他线程初始化
        int result = ::pthread_mutex_unlock(guard_mutex());
        if ( result != 0 ) {
            abort_message("__cxa_guard_acquire(): pthread_mutex_unlock "
                          "failed with %d\n", result);
        }
        return 0;
    }

    // The pthread mutex is recursive to allow other lazy initialized
    // function locals to be evaluated during evaluation of this one.
    // But if the same thread can call __cxa_guard_acquire() on the
    // *same* guard object again, we call abort();
    if ( inUse(guard_object) ) {
        abort_message("__cxa_guard_acquire(): initializer for function "
                      "local static variable called enclosing function\n");
    }

    // mark this guard object as being in use
    setInUse(guard_object);

    // return non-zero to tell caller to run initializer
    return 1;
}



//
// Sets the first byte of the guard_object to a non-zero value.
// Releases any locks acquired by __cxa_guard_acquire().
//
void __cxxabiv1::__cxa_guard_release(uint64_t* guard_object)
{
    // first mark initalizer as having been run, so
```

```
    // other threads won't try to re-run it.
    setInitializerHasRun(guard_object);

    // release global mutex
    int result = ::pthread_mutex_unlock(guard_mutex());
    if ( result != 0 ) {
        abort_message("__cxa_guard_acquire(): pthread_mutex_unlock "
                      "failed with %d\n", result);
    }
}



//
// Releases any locks acquired by __cxa_guard_acquire().
//
void __cxxabiv1::__cxa_guard_abort(uint64_t* guard_object) // 初始化异常时被调用
{
    int result = ::pthread_mutex_unlock(guard_mutex());
    if ( result != 0 ) {
        abort_message("__cxa_guard_abort(): pthread_mutex_unlock "
                      "failed with %d\n", result);
    }
    // now reset state, so possible to try to initialize again
    setNotInUse(guard_object);
}
```

最后提供一个很有价值的参考： http://wiki.osdev.org/C%2B%2B