Symbian OS C++程序员编码诀窍

版本 1.0，2003 年 10 月 24 日

NOKIA

声明

本文档中的信息基于其现有状况，不存在任何保证，包括销售保证、适用某一特殊用途的保证，或从任何建议、规范或范例中衍生出来的保证。此外，本文档中提供的并非最终信息，在其最终发布前会做较大改动。本文档仅用作信息通报。

诺基亚公司不承担所有因实施本文档中所表述的信息而产生的相关责任，包括侵犯任何知识产权的责任。诺基亚公司并不保证或认为使用这些信息不会构成对相应知识产权的侵犯。

诺基亚公司保留不预先通知而随时修改或撤销本文档的权力。

授权许可

本授权仅限于因个人应用而下载和打印本说明，除此之外，不存在对其它任何知识产权的授权许可。

目录

修订纪录

2003 年 10 月 24 日	版本 1.0	替换文档《针对 *Symbian OS* 的编码诀窍》

1 简介

1.1 读者对象和范围

本文的读者对象是：所有使用 C++语言为 Symbian OS 6.x/7.0s 开发应用的开发伙伴们。

有一个不成文的 80/20 法则，说的是：需要用 80%的时间去纠正开发中产生的 20%的问题。本文的目的就是要解决这 20%的问题。

2 内存

本节所述内容包括：对 Symbian OS 所提供的预防内存泄漏问题的一些技术作了回顾。所有开发者应该对此都有深刻理解：这是 Symbian OS 在编程方面的精髓！

2.1 有关清除堆栈（**CleanupStack**）

2.1.1 所有程序都应检查“资源用尽”出错

任何应用都可能在运行中发生因资源缺乏而导致的出错，例如，机器用尽了内存，或某个通讯端口不可用。这种类型的出错被称为一个异常。

必需区分异常与编程错误：编程错误用修改程序来解决，但一个程序是不可能完全消除出现异常的可能性。

因此，发生异常时，程序本身应该有能力从各种异常中恢复。在 Symbian OS 中，这一点特别重要，这是基于下列理由：

- 各种 Symbian OS 应用都被设计成能长时间运行（几个月，甚至几年）而不发生中断或系统重启。
- 各种 Symbian OS 应用都被设计成能在仅具备有限资源，特别是内存有限的设备上运行。因而，比起台式机上的应用，在有限资源设备上更容易发生“资源用尽”出错。

并非所有的 Symbian OS 设备都具有相同的资源，即，为某类 Symbian OS 设备设计并通过验证的应用可能在其他制造商的 Symbian OS 设备上发生资源性异常。

2.1.2 传统的侦错方法

在传统的 C 或 C++程序中，往往用一个 `if` 语句来检查是否发生了资源用尽出错。如：

```
if ((myObject = new CSomeObject()) == NULL)
PerformSomeErrorCode();
```

2.1.3 使用传统方法的问题

使用这种传统解决方法会产生两方面的问题：

它需要在每个可能导致资源用尽错误的独立函数周围放置许多额外的代码行。这样就会增加代码量，并降低可读性。

如果某个构造函数无法分配资源，就无法返回一个出错代码，因为构造函数没有返回值。结果就可能是一个不完整的被分配对象，这可导致程序崩溃。

- C++异常处理（try，catch 及 throw）机制为这些问题提供了一些解决方案，但并没有在 Symbian OS 中使用，这是因为其代码开销比较大。相反，Symbian OS 提供其本身的异常处理系统。

2.1.4 Symbian OS 中的解决方案

各种 Symbian OS 应用能使用下列规则获得有效的异常处理：

- 规则 1：所有可以异常退出的函数其名字都以字母‘L’结尾。各种异常都顺着调用栈通过一些“异常函数”向后传递，直到被一个“trap harness（捕获模块）”捕获为止。通常在针对各种控制台应用的 E32Main()主函数中实现这一功能，或作为图形用户界面程序的应用框架的一部分提供。
- 规则 2：当在堆中分配内存时，如果指向该内存的指针是一个自动变量（即，不是成员变量），必须将其推入清除堆栈中，以便当发生异常退出时能被释放掉。所有被推入该清除堆栈的对象都必须在销毁前弹出。
- 规则 3：C++构造函数或解构函数是不允许异常退出或失败的。因而，如果某个对象的构造函数出现资源不足错误而失败，所有可能导致失败的指令都必须移出该 C++构造函数，并将它们放入到 ConstructL()函数中，在 C++构造函数完成之后才调用该函数。这一过程被称为两阶段构造。

2.2 规则 1：异常退出函数和捕获模块

2.2.1 异常退出函数

Symbian OS 中的函数并不返回出错代码，而是一出现资源不足错误时就异常退出。一个异常退出就是对 User::Leave()的调用，它导致程序的执行被立即返回到捕获模块中，该函数就在其中执行。

所有可以异常退出的函数都以字母‘L’结尾。这使得程序员们明了：该函数是可以异常退出的。例如：

```
void MyFunctionL()
{
iMember = new (ELeave) CMember;
iValue = AnotherFunctionL();
User::LeaveIfError(iSession.Connect());
}
```

MyFunctionL 中的每一行都可能导致异常退出。其中的任何一行都使 MyFunctionL 成为一个异常退出函数。

然而需要注意的是：应用程序代码中很少有必要使用 TRAP，因为应用框架已经在适当的地方提供了这些捕捉错误的代码(TRAP)，也提供了相应的处理代码。在正常编码过程中并不需要使用错误捕捉代码。一般说来，处理各种异常退出的方法很简单，就是在函数名字后面加上一个字母‘L’，从而让其能顺着函数传递。

2.2.2 new (ELeave)运算符

在 Symbian OS 中， New 运算符失败的可能性很高，以至该运算符已经被重置而带上了一个参数，即 Eleave。当用这个参数调用 New 时，如果没能分配到所需的内存空间，被重置的 new 运算符就会异常退出。这一功能已经得到了全局性实现，所以，任何类都可以使用该运算符的 new (ELeave)版本，如：

```
CSomeObject* myObject = new CSomeObject;
if (!myObject) User::Leave(KErrNoMemory);
Can be replaced in by:
```

```
CSomeObject* myObject = new (ELeave) CSomeObject;
```

2.2.3 NewL()和NewLC()惯例

习惯上，Symbian OS 的一些类经常实现 NewL()和 NewLC()方法。这两个方法在类定义中被声明为 static 方法，这就使得它们可以在该类的一个实例存在之前就被调用。可以使用类范围来调用它们。如：

```
CSomeObject* myObject = CSomeObject::NewL();
```

NewL()在堆上创建了该类的一个新实例，当出现内存不足错误时，它就会异常退出。对简单对象来说，这仅仅涉及到对 new (ELeave)的调用。然而，对复合对象来说，它要用到两阶段构造（请见下面对“规则 3”的讲述）。

NewLC()在堆上创建了该类的一个新实例，并将其推入到清除堆栈（见下面对“规则 2”的讲述），如果出现了内存不足错误，就发生异常退出。（总的说来，某一个方法尾部的‘C’后缀是指：它在返回前将一个已创建的对象推入到堆中。）

当创建 C-类（C-class）对象时，如果某个成员函数会指向该对象，就应该在程序中使用 NewL()；而如果某个自动变量会指向该对象，就应该使用 NewLC()。但是，并不建议对每个类都实现 NewL()和 NewLC()。实际上，如果仅仅从应用中的一个地方调用 NewL()和 NewLC()，实现它们的代码行比起所保存的要多许多。较好的做法是：对每个单一类都作一下评估，看看其是否需要用到 NewL()和 NewLC()。

2.2.4 TRAP and TRAPD 使用捕获模块：TRAP 和 TRAPD

在出现异常的情形中，开发者可以用一个捕获模块来处理一个异常。然而，TRAP 和 TRAPD 的使用仅限于特殊情况，而对所有的一般性编码来说，则应避免使用。通常，最佳反应过程是：允许该异常退出传递回 Active Scheduler（活动调度器），以便进行默认处理。如果不能确认是否真正需要一个捕获模块，应该存在一个经济的或明晰的方法，以实现相同的功能。

Symbian OS 提供了两种非常相似的捕获模块宏，即 TRAP 和 TRAPD。当捕获模块中的代码执行发生异常退出时，程序控制立即返回给这个陷阱宏。然后该宏返回一个可以由调用函数使用的出错代码。

要在某个捕获模块中执行一个函数，可以使用 TRAPD，如下所示：

```
TRAPD(error, doExampleL());
if (error != KErrNone)
{
// Do some error code
}
```

TRAP 与 TRAPD 的不同之处仅仅在于：前者的程序代码必须声明异常代码变量。TRAPD 用起来更方便，因为在宏的内部声明了 error。如果用 TRAP，上述代码就变成：

```
TInt error;
TRAP(error, doExampleL());
if (error != KErrNone)
{
// Do some error code
}
```

所有被 doExampleL()调用的函数也在捕获模块内部执行，就像所有被其调用的函数一样。在 doExampleL()内部嵌套的任何函数如果发生了异常退出，也将返回到这个捕获模块中。其他的 TRAP 模块也可以被嵌套（nested）在第一个内部，这样就可以在该应用内部的不同级别上对所有的出错进行检查。

2.3 规则 2：使用清除堆栈

2.3.1 为何需要清除堆栈（Cleanup Stack）

如果某个函数出现了异常，就立即将控制返回给在其中调用它的 TRAP 模块。一般说来，默认的 TRAP 模块处于该线程的活动调度器内。这意味着：TRAP 模块中这些被调用函数内部的任何自动变量都被销毁了。然而，如果这些自动变量中的任何一个是指向堆中已分配对象的指针，就会产生问题。当发生异常退出并销毁了这个指针时，被指向对象就悬空了，从而产生内存泄漏。

例如：

```
void doExampleL()
{
CSomeObject* myObject1 = new (ELeave) CSomeObject;
CSomeObject* myObject2 = new (ELeave) CSomeObject;// WRONG
}
```

在这个范例中，如果成功创建了 `myObject1`，但却没有足够的内存空间可分配给 `myObject2`，`myObject1` 就会在堆中悬空。

这样，我们就需要某些机制来保留这类指针，以便让其所指向的内存在异常退出后得到释放。Symbian OS 在清除堆栈中为此目的提供了一种机制。

2.3.2 使用清除堆栈

清除堆栈中含有一些指针，它们指向所有当发生异常退出时需要释放的对象。这意味着：所有 C-类（C-class）对象都由自由变量而不是实例数据所指向。

当发生异常退出时，会弹出 TRAP 或 TRAPD 宏，并销毁从 TRAP 起始时推入到该清除堆栈中的一切东西。

所有的应用程序都有自己创建的清除堆栈。（应用程序框架在图形用户界面应用中自动创建了一个。）典型的情况是：所有的应用程序将至少有一个对象被推入到清除堆栈中。

我们用 `CleanupStack::PushL()`将对象推入到清除堆栈中，而用 `CleanupStack::Pop()`将其弹出。如果位于清除堆栈中的那些对象不再有机会因异常退出而悬空，就必须将这些对象弹出。通常在释放该对象之前会发生异常退出。我们一般使用 `PopAndDestroy()`，而不是 `Pop()`，因为前者将确保该对象在弹出的同时被释放掉，从而避免释放前发生异常退出及内存泄漏。

拥有指向其他 C-类（C-class）对象指针的复合对象必须在其解构器中被释放掉。因此，并不需要将任何由另一个对象的成员数据（而不是一个自动变量）所指向的对象推入到清除堆栈中。事实上，一定不需要将其推入到清除堆栈中，否则当发生异常退出时它就会被销毁两次：一次由解构器，另一次由这个 TRAP 宏。

2.4 规则 3：两阶段构造

有时候，某个构造函数需要分配资源，如内存。最普遍的情况就是某个复合 C-类（C-class）：如果某个复合类含有一个指向另一个 C-类（C-class）的指针，它就需要在自己的构造过程中为那个类分配内存。

（注意：Symbian OS 中的 C-类（C-class）总是被分配在堆中，而且总是将 `Cbase` 作为其最根本的基类。）

在下列范例程序中，CmyCompoundClass 具有一个数据成员，这是一个指向 CmySimpleClass 的指针。

这里是 CmySimpleClass 的定义：

```
class CMySimpleClass : public CBase
{
public:
CMySimpleClass();
~CMySimpleClass();
…
private:
TInt  iSomeData;
};
```

这里是 CmyCompoundClass 的定义：

```
class CMyCompoundClass : public CBase
{
public:
CMyCompoundClass();
~CMyCompoundClass();
…
private:
CMySimpleClass* iSimpleClass; // owns another C-class
};
```

开发者可能会为 CmyCompoundClass 撰写构造函数：

```
CMyCompoundClass::CMyCompoundClass()
{
iSimpleClass = new CMySimpleClass;  // WRONG
}
```

现在来考虑当创建了一个新的 CmyCompoundClass 时发生了什么：

```
CMyCompoundClass* myCompoundClass = new (ELeave) CMyCompoundClass;
```

用上面这个构造函数将产生下列依次发生的事件：

为 CmyCompoundClass 的实例分配了内存。

调用了 CmyCompoundClass 的构造函数。

该构造函数创建了 CmySimpleClass 的一个新实例，并将一个指向它的指针存储到 iSimpleClass 中。

构造函数完成工作。

但是，如果由于内存不足而导致第三步失败，将发生什么？不可能从构造函数返回一个出错代码以指出该构造过程并没有完成。New 运算符将返回一个指向分配给 CmyCompoundClass 的内存的指针，但它指向的是一个部分构造的对象。

如果我们让该构造函数异常退出，那么当该对象没有完全构造时就能被探测到，如下所示：

```
CMyCompoundClass::CMyCompoundClass()   // WRONG
{
iSimpleClass = new (ELeave) CMySimpleClass;
```

```
}
```

然而，这并不是发现出错的可行方法，因为我们已经为 CmyCompoundClass 的实例分配了内存。某次异常退出将销毁指向所分配内存的指针（this），而且无法释放它，从而导致内存泄漏。

解决方案是：在 C++构造函数对该复合函 数进行初始化之后，为该对象的组件分配所有的内存。按惯例，在 Symbian OS 中这是在 ConstructL()中实现的，如：

```
void CMyCompoundClass::ConstructL() // RIGHT
{
iSimpleClass = new (ELeave) CMySimpleClass;
}
```

The C++ constructor should contain only initialization code that cannot leave (if any):

该 C++构造函数应该仅含有不可能异常退出（如果有的话）的初始化代码：

```
CMyCompoundClass::CMyCompoundClass()   // RIGHT
{
// Initialization that cannot leave.
}
```

现在，构造对象如下：

```
CMyCompoundClass* myCompoundClass = new (ELeave) CMyCompoundClass;
CleanupStack::PushL(myCompoundClass);
myCompoundClass->ConstructL();   // RIGHT
```

为方便起见，可以将其封装在一个 NewL()或 NewLC()方法中。

2.4.1 用 NewL()和 NewLC()实现两阶段构建

如果某个复合对象有一个 NewL()方法（或 NewLC()方法），那么就应该同时包含构造过程的两个阶段。分配阶段之后，如果 ConstructL()发生了异常，应该在调用 ConstructL()之前将该对象推入到清除堆栈中。例如：

```
CMyCompoundClass* CMyCompoundClass::NewLC()
{
CMyCompoundClass* self = new (ELeave) CMyCompoundClass;
CleanupStack::PushL(self);
self->ConstructL();
return self;
}
CMyCompoundClass* CMyCompoundClass::NewL()
{
CMyCompoundClass* self = new (ELeave) CMyCompoundClass;
CleanupStack::PushL(self);
self->ConstructL();
CleanupStack::Pop(); // self
return self;
}
```

2.5 公共错误

2.5.1 误用 TRAP 和 TRAPD

一些类会重复使用下列形式的代码：

```
void NonLeavingFunction()
{
TRAPD(error, LeavingFunctionL());
}
```

这是一段合法的代码，但却不应该广泛使用。考虑到可执行二进制代码的大小和执行速度，错误捕捉模块的代价高昂，除非很小心使用，否则将导致代码丢失错误。经常情形是：在该方法名的尾部加上字母‘L’，使异常退出能够向上传递。然而需要注意的是：为维持库兼容性，有时候这成为不可能。库设计应该充分考虑到未来异常退出的需要。

下列代码非常不好，因为整个 TRAP 都是无意义的！

```
void NonLeavingFunction()
{
TRAPD(error, LeavingFunctionL());
if (error != KErrNone)
User::Leave(error);
}
```

2.5.2 错误使用了 new 运算符

下面的代码是非法的，也是危险的:

```
void NonLeavingFunction()
{
bar* foo = NULL;
TRAPD(error, foo = new bar());
foo->DoSomething();
}
```

在这种情形中，基本地，我们应该使用 new 运算符（本身不会退出）的 new (ELeave)版本，否则就会导致内存泄漏，也会导致对某个未初始化指针的使用。

2.5.3 错误使用了后缀‘L’

```
void NonLeavingFunction()
{
LeavingFunctionL();
bar* foo = new (ELeave) bar();
bar* foo1 = bar::NewL();
}
```

该函数的所有三行代码都违反了后缀‘L’的使用规则。这里有两种选择：

1. 退出行必须在一个错误捕捉代码（TRAP）中被捕获（也许不是最佳方案）。
2. 函数 NonLeavingFunction 必须变成一个‘L’函数（也许较佳）。

请注意：这段代码还违反了规则 2（使用清除堆栈，如上所述），因为当 NewL 退出时，foo 在堆中就被悬空了。

2.6 内存泄漏

在 Symbian OS 代码的开发过程中经常进行内存测试非常重要。如果发现了一个内存泄漏，那么就容易在当前的工作环境内部解决这一问题，而不需要去搜寻整个应用程序。

Symbian OS 提供了可用于辅助 Symbian OS 代码内存压力测试的、针对编译连接的各种堆内存失败的调试工具。用这些工具我们将看到应用程序在两方面的表现：

1. 内存用完时应用程序的表现。
2. 应用程序关闭时所报告的内存泄漏。

目标是：至少能“向用户传达完整的数据信息”。特别重要的是：在内存测试时使用‘**Back**（返回）’功能键。直接使用右上部的关闭按钮来关闭模拟器将使得内存检查代码无法运行。

2.6.1 使用 WINS 模拟器中的工具

WINS 模拟器提供了一个能检查内存性能的工具，只要按 **CTRL-SHIFT-ALT-P** 键就可执行这种检查。在 SDK 文档及《专业 Symbian 编程》(Professional Symbian Programming)一书的第 158 页中都有详细介绍。该书所讲述的实用程序可用于大部分基于 Symbian OS 的 SDK，如 Series 60 SDK。各个 SDK 的测试实例其屏幕外观各不相同。

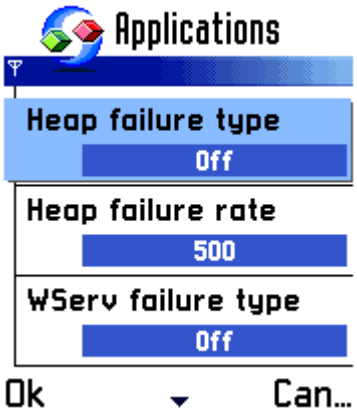

图 1. Series 60 终端模拟器内存泄漏压力测试实用程序

图 2. 诺基亚 9210 通信器模拟器内存压力测试实用程序

在 Symbian OS 中调试内存泄漏是一件令人生畏的事情，但有些技术可以使这一过程变得不那么痛苦。然而，寻找内存泄漏从来不是一件小事，预防其发生才是最好的对付办法！下列窍门可以在一开始就防止出现内存泄漏，以免日后搜寻之苦。

1. 理解清除堆栈和 `Leave/TRAP` 的范例。
2. 经常生成并运行代码 – 如果发生了泄漏，这样就更容易了解其出处。
3. 使用 Symbian OS 6.x/7.0s 的堆检测宏。
4. 测试时，请退出该应用。不要只是杀掉模拟器。
5. 代码检查非常有用。

有两种类型的内存泄漏。“静态”泄漏是一种可重复泄漏，总是发生在应用运行时，它由 `new` 和 `delete` 运算符的相互不匹配引起。这些泄漏相对比较容易找到，因为它们总发生在相同的地方，所以是可调试的。“动态”泄漏不太会重复。举例来说，由出错状态，或争抢状态所导致的泄漏就是如此。

2.6.1.1 泄漏了什么？

当关闭某个应用时，如果内存泄漏了，模拟器会出现严重提示（panic，实际上这是运行了一个 `_UHEAP_MARKEND` 宏）。应用程序需要干净地退出，即使在开发进行过程中也应该如此。当开发过程中出现‘程序关闭严重提示’这一情况时，可以非常直接对其进行处理。如果拖而不决，以后处理的难度将十倍于此。

在微软的 Visual C++调试程序中，严重提示（panic）以“由位于 0xxxxxx 的代码调用的用户断点”对话框形式出现。栈跟踪（用“*View>Debug Windows>Call Stack*”）显示其位于 `CcoeEnv` 解构函数中。

接下去，请按‘OK’和‘F5’。这时会遇到另一个用户断点，这一次位于 `DebugThreadPanic`。这时输出窗口显示 `Panic ALLOC` 及一个地址。选择这个地址，将其复制到剪贴板上（“*Edit>Copy*”）。

这里是尚未释放内存单元的 16 进制地址。试着将这个地址投射到一些可能的类型，就有可能从这个地址找出泄漏类的类型。使用 Visual Studio 中的“Quick watch”窗口，并努力将 `badCell` 指针投射到下列类型上：

`CBase*` (in case it is a `CBase`-derived object).

`TDesC16*` (in case it is a string).

这些投射无法给出任何有用信息，虽然当没有关闭某个客户端时服务器一般应出现严重提示(panic)，但也有可能这是一个 R-类（R-Class，资源处理）。另外，它也可以是一个被错误地置于堆内存中的 T 类（T-Class）。请注意：当某个大型的复合 C-类（C-Class）发生了泄漏，这种技术可能会给出稍稍偏离的信息，因为很有可能会报告该大型类的一个成员函数，而不是父函数本身。

2.6.1.2 它被分配到了何处？

一旦知道了已泄漏内存的地址，可以在堆内存的分配器函数中设定一个条件断点以确定其所分配的点。

所有的堆内存分配都通过函数 RHeap::Alloc(int)进行。所以，先在那里放一个断点。Symbian 目前并不开放这一函数的源代码，但却可以用微软的 Visual C++ 中的“*Edit>Breakpoints>Break At*”功能明确无误地设置一个断点。

用“*Debug>Go*”（**‘F5’**）继续操作，直到系统进行首次分配。源代码是不可见的，但却可以看到反汇编的代码。顺着反汇编代码往下看，经过 retryAllocation，直到下一个函数 roundToPageSize 开始前的一行。在 RET 行放置一个断点，在这个点上，注册器 EAX 将包含来自 RHeap::Alloc 函数的返回值。当其值等于出现问题的内存单元时，用“*Edit>Breakpoints*”来设置一个断点。请先去除 RHeap::Alloc 处的断点，选择新的断点并使用‘条件’来设置这样的条件：返回值为被跟踪单元。

在两个对话框中都点击‘OK’，然后用“*Debug>Go*”继续。与之前一样运行该应用程序。当程序在断点处停止执行时，请检查堆栈，看看问题单元被分配到了何处。

有时可能分配了相同的单元，又多次释放了这个单元。这种情况下，我们只对最后一次分配感兴趣。

如果并没有分配单元，这也许是因为，这次运行与第一次不太一样，而泄漏单元则位于不同的地方。继续工作，直到应用程序退出，并找到新问题单元的地址。然后在同一位置将其设置为另一个断点，但加上一个条件，即捕获新的问题单元。这时用“*Debug>Restar*”来重新启动。可能会出现“不能恢复所有断点”这样的出错信息。这是因为：当可执行模拟器（EPOC.EXE）第一次启动时没有加载 EUSER DLL。解决办法是：重新激活 RHeap::Alloc(int)断点，运行程序直到该断点，然后恢复其它的条件和断点。

请注意：这段代码由于使用了断点，执行速度会大大降低，所以请在最后一刻才激活断点！同时，同样的地址可以被分配许多次，哪一个才与泄漏有关呢？这就需要当每次遇到断点时都对调用栈进行调查，以发现当时的场景！

2.7 检查和严重提示(Asserts and Panics)

使用__ASSERT_DEBUG 测试宏可以避免许多问题。应该不受限制地使用这些宏，检查是否有比较愚蠢的参数进入到了这些函数中，是否有空指针，以及其他的出错条件等。许多出错条件并不直接导致应用的失败，但却会在以后导致一些副作用。如果能于错误出现之时就捕获它，以后的调试就变得非常容易。例如：

```
CMyClass::Function(CThing* aThing)
{
__ASSERT_DEBUG(aThing, Panic(EMyAppNullPointerInFunction));
}
```

3 系统资源的使用（ROM 和 RAM）

3.1 重要性

移动电话是一种资源有限设备。然而，它却存在大量的可用功能，这对现有的系统资源提出了很高的要求。开发者需要注意这些制约，尽可能地少用这些有限的资源。

3.2 减少代码量

最终编译后的代码必须尽可能得小，以便为设备留出尽可能多的可用空间，这一点非常重要。以下诀窍就如何保证不浪费存储空间提供了一些指导性意见。为解决这一问题，你需要花一点时间去检查代码，同时还要考虑一些其他的方法，使得编译后的代码量变小。

3.2.1 不必要的导出函数

当使用 IMPORT_C 和 EXPORT_C 从某个 DLL 中导出一些函数时，它们会因为导出表而耗尽空间。只需要导出那些必需在该 DLL 外使用的函数。

3.2.2 复制和粘贴

复制和粘贴经常导致代码臃肿。当需要重用其他模块中的代码时，请向自己提问下列问题：

1. 这段代码是否实际需要？
2. 为此任务是否复制了过多的代码？
3. 如果将该函数提取到某个基类，或到一个帮助模块，使一个以上的地方都能使用它，这样是不是更好？
4. 针对所需任务，该代码是否能重写，使其更为有效，而不是去复制那些接近需求的东西？

3.2.3 明显不可分解的函数

在许多地方，一些函数出现在同一个类中，这些函数实现非常类似的任务。经常的情况是：可以把这些公共代码提取到一个单一的函数中去，对该函数实施参数化，以便完成所需的不同任务。

3.2.4 过分的 TRAP 模块

当编译错误捕捉代码模块时，它们会消耗内存空间。含有许多 TRAP 宏的代码（如，在一个类中含有五个以上的 TRAP 宏）将消耗太多的空间。另一种可能的情形是：设计不正确，使得 TRAP 模块不是广泛地用于正常代码。在这里，我们允许高级开发工作中有特别的出错处理和恢复程序。

3.2.5 调试发行代码

如果有任何用于登录、调试，或测试的代码，必须将它们从发布版中剔除。可以为此目的使用编译指示#ifdef _DEBUG。

3.2.6 不必要的虚函数

不必要的虚函数是有害的，原因类似于函数导出，它们会创建额外的 vtable（虚表）函数。

3.2.7 使用公共控件

如果可能，请使用系统（或其它共享 DLL）提供的框架控件， 而不是去开发新的控件。

3.2.8 _L 宏的误用

现在已经不建议使用带字母_L 的宏了，取而代之的是效率更高的_LIT 宏。

3.3 减少使用 RAM

有许多方法可以减少 RAM 的使用。其中的一些方法（如 bitfields）可能使代码可读性变差，所以经常要在减少 RAM 使用和增加代码复杂性这两者之间作折衷。

3.3.1 使用 bitfields（位元组合）， 而不使用太多的 Tbools

考虑用 bitfields（位元组合）来存储类中大量的布尔数据。每一个 Tbool 需要 32 位的 RAM，而这 32 位可以用位元组合的形式保存 32 个布尔值。如上所述，我们可以比较一下：提高代码复杂性和使用 bitfields 各自的潜在利益。

3.3.2 阵列粒度的使用警示

可以为所有继承自 CArray 的类规定粒度。其目的是：只以一定大小的块为阵列分配空间，从而使代码更为高效。这种方法很有效，但需要考虑粒度的选取问题。如果需要为 5 至 8 个对象准备一个阵列，那么粒度定为 4 到 5 就是明智的。如果一个阵列总是含有 15 个对象，那么粒度就应该定为 15。然而，如果对象的数目是 2 个到 3 个，那么粒度定为 100 就很愚蠢了。类似的，如果有 101 到 105 个对象，那么粒度为 100 也是愚蠢的，因为每次都需要分配 200 个空间。当然，粒度为 1 也属不智，因为这将需要太多次的重新分配。最终选择取决于使用方式。

3.3.3 避免全局数据

不要使用全局数据。对于只用于一个函数内的变量，请用局部变量，而不是成员变量。

3.3.4 小心基类的成员数据

如果要写一个用途广泛的基类，请小心成员数据。不要将只用于某些继承类的成员数据添加其中，因为每个继承类除了拥有它，别无其他选择。注意只将真正普通的成员函数包括其中。

3.3.5 正确使用清除堆栈

如果正确地使用了清除堆栈，代码中就不应该再有内存泄漏，这样就能保证该应用没有使用超出其需要的 RAM。

3.3.6 尽早删除

如果在堆中分配了临时对象，当不再需要它们时请将其立即删除。如果这些临时对象的生命长于其需要的时间，那么该应用的 RAM 开销往往要高于其实际所需要的。请记住，如果删除了某个临时对象，而指向这个对象的指针还在，那么就需要将该指针设为 NULL，以防止非法访问或两次删除。

3.3.7 用最大数据集进行硬件测试

如果某个数据集有上限，那么就用最大数据集来进行硬件测试。如果从来没有对硬件进行过极限测试，很有可能会忽略某些非常慢的运算，或导致问题等。

3.3.8 分解复杂的长运算

在屏幕上显示冗长列表会对 RAM 使用形成压力。而且，当初始化各列表控件（如：设备上所有联系人的列表，或便条列表）时，其表现极差。可以编写特别的控件来避免这种情况，这些控件只组装屏幕上可见的栏目。当滚动时，释放那些离开了屏幕区的栏目，同时添加新出现的栏目。

3.4 减少堆栈的使用

目标硬件上某个应用可用的堆栈比起 Windows NT 环境中模拟器可用的巨量堆栈来要小得多。结果是：在 `WINS` 模拟器中能良好运行的代码在硬件中却不能运行，而且出现很明显的随机性严重提示 (panic)。减少堆栈使用并不容易，但还是需要引起密切的关注。

3.4.1 正确使用描述符

有两种类型的描述符，即堆描述符（`HBufC`）和栈描述符（`HBufC`）。所有的描述符都使用其中之一来储存。当栈溢出时，90%时间是由栈中大型的描述符引起的。小心对待那些将导致隐含复制描述符的那些操作，并尽可能避免这种情况的出现。在某些情况下，最好分配 `HbufCs`，而不是 `Tbufs`。

某些相关的 Symbian OS 类，如 `Tparse`，也能开销掉许多栈空间。可以考虑使用那些耗费栈空间较少的版本（如：`TParseBase`）。

向描述符传参数比传值更好。

3.4.2 小心使用递归，在限度内生成

如果需要递归程序，请注意栈需求。应该努力降低向下传递的参数的大小，并力图将本地自动变量移出该函数的递归部分。尽可能地在递归限制深度内生成（build）代码，以免栈溢出。

3.4.3 注意登录代码

登录代码往往涉及到对超长描述符及将其写入到文件中的格式化工作。由于这一理由，它们往往成为栈溢出的原因。

3.5 盘容量降低的处理

对闪存文件系统（Flash File System，FFS，其别名是 C:驱动器）上可用自由空间的监测系统，我们定义了两个级别：警示级（Warning Level，WL）和临界级（Critical Level，CL）。当自由磁盘空间遇到这些级别中的一个时，系统（`EikSrvUI`）将显示一个全局提示，向用户发出有关当前情势的警示。此后各种应用程序和服务器就忽略掉警示级而专注于临界级。

所有对磁盘文件以已知的文件尺寸进行创建或写入操作都必须首先以那个尺寸作为方法 `FFSSpaceBelowCriticalLevelL` 的参数来检查临界级。如果磁盘空间已经很低，或者说已经低于临界级，这个方法就会返回 `Etrue`。应用程序就不能再进行写入操作，同时通知用户，磁盘已满。（用 `KerrDiskFull` 出错代码作异常退出可以达到这个目的。）

所有对磁盘文件以一个不知的文件尺寸进行创建或写入操作都必须先检查临界级，向方法 FFSSpaceBelowCriticalLevelL 传递一个合适的预估尺寸或‘0’（默认）作为参数。这里的‘0’可用于检查是否已经低于临界级了。

比较麻烦的情况是在几个数据库（如联系人）中创建单一项目。在这些情况中，可以使用一个预估值，用作因添加该项目而需要的数据库尺寸增量。

SysUtil.h/SysUtil.dll 中有临界级检查方法。其 API 看上去如下所示:

```
/**
 * Checks if the free FFS (internal Flash File System) storage
 * space is or will fall below Critical Level (CL).
 * The CL and FFS drive letter is defined by this module.
 * @param aFs File server session.
 *  Must be given if available in the caller,
 *  e.g. from EIKON environment.
 *  If NULL this method will create a temporary session for
 *  a check, but then the check is more expensive.
 * @param aBytesToWrite number of bytes the caller is about to add
 *  FFS, if known by the caller beforehand.
 *  The default value 0 checks if the current
 *  space is already below the CL.
 * @return ETrue if storage space would go below CL after adding
 *  aBytes more data, EFalse otherwise.
 *  Leaves on error.
 */
IMPORT_C static TBool FFSSpaceBelowCriticalLevelL(
RFs* aFs, TInt aBytesToWrite = 0);
```

4 生成（Build）ARM 目的文件

4.1 概述

针对 ARMI 的生成（Build）工作总体上比针对 WINS 的要困难得多，因此，从一开始就寻找由 gcc 报告的额外编译错误和报警信息就是再正常不过了。首先这是由于：在很多情况下 gcc 比微软的编译器要严格得多，而且具有一些微妙的差异，它们在第一次的 ARMI 生成（Build）过程中就会表现出来。下面几节涉及一些最通用的问题。

4.2 函数导出

当定义导出函数时，gcc 的工具链比 WINS 工具链要严格得多。从某个 DLL 导出一个函数的正确方式如下所示：

在头文件中：

```
class CMyClass : public CBase
{
IMPORT_C void Function();
}
```

在 CPP 文件中：

```
EXPORT_C void CMyClass::Function()
{
}
```

WINS 工具链并不在意是否将 EXPORT_C 排除在 CPP 文件之外了，总之它会导出该函数。然而，gcc 工具链需要 IMPORT_C 和 EXPORT_C 之间能完美匹配。如果不能，就不能从 DLL 中导出该函数。最终，当试图连接这个 DLL 时将导致如“无法找到函数”之类的错误。

4.3 来自 PETRAN 的“MyDll.DLL has (un)initialized data”错误

Symbian OS 架构并不允许 DLLs 具有数据片（静态数据，已初始化的或未初始化的）。要确定这种数据片的意义是一个棘手问题：

- 该 DLL 的所有用户都能共享它吗？
- 是否需要针对每个附着于该 DLL 的处理都复制它？
- 在顶层存在着重要的运行时环境以解答任何可能的问题

然而，由于 WINS 模拟器使用了底层 Windows DLL 架构，它能用“copy-on-write”语法提供预处理 DLL 数据。这就是为什么在为某台实际 Symbian OS 设备生成（built）代码之前，总是查不出问题。

请看本节中的 C++代码，它被添加到了文件 QSORT.CPP 中。该文件是 ESTLIB.DLL 的一部分。

```
// variables
struct div_t      uninitialised1;   // in .DATA
static struct div_t  uninitialised2;// in .BSS
struct div_t      initialised1 = {1,1};   // in .DATA
static struct div_t  initialised2 = {2,2};// in .DATA

// constants
const struct div_t   const1 = {3,3};
const static struct div_t  const2 = {4,4};
const TPoint none(-1,-1);
```

```
static const TText* plpPduName[12] =
{
_S("Invalid"),
_S("DataFlowOff"),
_S("DataFlowOn"),
_S("ConnectRequest"),
_S("ConnectResponse"),
_S("ChannelClose"),
_S("Info"),
_S("ChannelDisconnect"),
_S("End"),
_S("Delta"),
_S("EndOfWrite"),
_S("PartialWrite")
};
```

当生成这段代码时，来自 PETRAN 的消息看上去如下所示：

```
PETRAN - PE file preprocessor V01.00 (Build 170)

WARNING: Dll 'ESTLIB[10003B0B].DLL' has initialised data.

WARNING: Dll 'ESTLIB[100002C3].DLL' has uninitialised data.
```

相关联的.map 文件含有能帮助向下追踪有关源文件的信息。

请到 Symbian OS\release\arm4\urel\dllname.map 查找。

搜寻“.data”或“.data”。

在这个范例中，我们发现：

```
.data    0x10017000 0x200
         0x10017000    __data_start__=.
 *(.data)
 .data   0x10017000 0x40 ..\..\Symbian
OS\BUILD\STDLIB\BMMP\ESTLIB\ARM4\UREL\ESTLIB.in(QSORT.o)
         0x10017000    initialised1
 *(.data2)
 *(SORT(.data$*))
         0x10017040    __data_end__=.
 *(.data_cygwin_nocopy)
.bss     0x10018000 0x18
         0x10018000    __bss_start__=.
 *(.bss)
 .bss    0x10018000 0x18 ..\..\Symbian
OS\BUILD\STDLIB\BMMP\ESTLIB\ARM4\UREL\ESTLIB.in(QSORT.o)
         0x10018008    uninitialised1
 *(COMMON)
         0x10018018    __bss_end__=.
```

所以，该 DLL 有 0x18 字节的未初始化数据（.bss）和 0x40 字节的已初始化数据（.data），所有这些都来自 qsort.o。

initialised1 和 uninitialised1 这两个变量都具有全局范围，所以.map 文件按文件名列出了它们（并将两者都放进了已初始化数据中）。

从上面的代码中移去最前面四行，只留下被声明为 const 的变量，但却只减少了.bss 的 0x08 字节，及.data 的 0x30 字节。这里还存在两个问题：

如果 C++对象有一个构造函数，那么将其定义为 const 也没什么用。虽然分配了未初始化数据的 8 个字节以保持 Tpoint 对象，但是在构造函数完成之前它并不会成为 const。

const TText*声明表示，可能无法改变 Ttext 的值，但它也不会使指针成为一个常量。已初始化数据的 48 个字节是 plpPduName 阵列中的 12 个指针。要使这些指针成为常量，并使其指向的值也成为常量，该声明中还需要在 TText*后加上额外的 const。

```
static const TText* const plpPduName[12] =
{
_S("Invalid"),
_S("DataFlowOff"),
_S("DataFlowOn"),
_S("ConnectRequest"),
_S("ConnectResponse"),
_S("ChannelClose"),
_S("Info"),
_S("ChannelDisconnect"),
_S("End"),
_S("Delta"),
_S("EndOfWrite"), _S("PartialWrite")
};
```

移去 Tpoint 全局变量并向 plpPduName 阵列添加额外的 const，最终将导致移去最后出问题的 .bss 和 .data。