5. 释放类

5.1. 记住释放类实例

类实例不得不手动释放,否则你将得到内存泄露.我建议使用 FPC -gl -gh 选项来侦查内存泄露(查看 https://castle-engineio/manualoptimization.php#section_memory).
注意，这不涉及提引（raised）异常。尽管当提引（raising）一个异常时，你确实创建一个类（并且它是一个完美的正常的类，并且你也可以创建你拥有的为这个目的类）。但是这个类实例是自动释放的。

5.2. 如何释放

为释放类实例，最好在你的类实例上调用 FreeAndNil(A)。它检查是否 A 是 nil ，如果不是—调用它的析构函数（destructor），并且设置 A 到 nil 。所以在一个行中调用它很多次不是一个错误。
它是或多或少的一个快捷方式（shortcut），对于

```
if A <> nil then
begin
  A.Destroy;
  A := nil;
end;
```

事实上，这是一个过度简化的，在一个适合的引用（reference）上，因为 FreeAndNil 做了一个有用的骗局(trick)，并在调用析构函数（destructor）前设置变量 A 为 nil。这帮助阻止某一个类的错误（bugs）—这个主意是，"外部的" 代码应该永远不能访问类的一个半破坏的（half-destructed）实例。
你将经常看到人们使用 A.Free 方法。这是像做

```
if A <> nil then
  A.Destroy;
```

这释放 A ，除非它是 nil 。
注意，在正常情况下，在一个可能是 nil 的实例上，你应该永远不调用一个方法。所以，如果 A 可能是 nil，乍看之下，调用 A.Free 可能看起来令人怀疑。然而， Free 方法是这个规则的一个例外。在实施（implementation）中它做的有些卑鄙（dirty）— 也就是，检查是否 Self <> nil。这个卑鄙的骗局（dirty trick）仅工作在非虚拟（non-virtual）方法中(这不调用任何的虚拟方法和不访问任何的字段（fields））。
我建议总是使用 FreeAndNil(A)，没有例外的情况，并永远不直接地调用 Free 方法或 Destroy 析构函数（destructor）。*Castle Game Engine* 像这样做的。它帮助保持一个精确的断言（assertion），*所有的引用（references）不是 nil，就是指向有效的实例。*.

5.3. 手动和自动释放

在很多情况下，需要释放实例不是多少问题。你只是（just）写一个析构函数（destructor），这匹配一个构造函数（constructor），并解除重新分配（deallocates）被分配在构造函数（constructor）中的一切（或，更彻底，在类的整个使用期限）。 注意，

仅释放每个东西**一次**. 通常，设置释放的引用（reference）为 nil 是一个好主意，通常，通过调用 FreeAndNil(A) 来做它是最安逸的.
所以,像这个：

```
uses SysUtils;

type
  TGun = class
  end;

  TPlayer = class
    Gun1, Gun2: TGun;
    constructor Create;
    destructor Destroy; override;
  end;

constructor TPlayer.Create;
begin
  inherited;
  Gun1 := TGun.Create;
  Gun2 := TGun.Create;
end;

destructor TPlayer.Destroy;
begin
  FreeAndNil(Gun1);
  FreeAndNil(Gun2);
  inherited;
end;
```

为避免明确地(explicitly)释放实例,也可以使用*"ownership"*的 TComponent 特征.一个*被拥有*的对象将自动地通过*拥有者*释放.该机制是智能的，并且它将永远不会释放一个应该释放的实例(因此，如果你早期手动释放拥有的对象，事情也将正确地工作).我们可以更改前一个示例到这个：

```
uses SysUtils, Classes;

type
  TGun = class(TComponent)
  end;

  TPlayer = class(TComponent)
    Gun1, Gun2: TGun;
    constructor Create(AOwner: TComponent); override;
  end;

constructor TPlayer.Create(AOwner: TComponent);
begin
```

```
  inherited;
  Gun1 := TGun.Create(Self);
  Gun2 := TGun.Create(Self);
end;
```

注意：我们需要在这里重写(override)一个虚拟 TComponent 构造函数（constructor）。所以我们不更改构造函数（constructor）参数。(事实上，你可以一用 reintroduce(再引入)声明一个新的构造函数（constructor）。但是注意，由于一些功能,例如.streaming(流),将仍然使用虚拟构造函数（constructor）,所以，在任何情况下确保它正确地工作.)
自动释放的另一种机制是 list-classes(列表-类)的 OwnsObjects 功能/(默认已经是 true!),像 TFPGObjectList 或 TObjectList.所以我们也可以写：

```
uses SysUtils, Classes, FGL;

type
  TGun = class
  end;

  TGunList = specialize TFPGObjectList<TGun>;

  TPlayer = class
    Guns: TGunList;
    Gun1, Gun2: TGun;
    constructor Create;
    destructor Destroy; override;
  end;

constructor TPlayer.Create;
begin
  inherited;
  // Actually, the parameter true (OwnsObjects) is already the default
  Guns := TGunList.Create(true);
  Gun1 := TGun.Create(Self);
  Guns.Add(Gun1);
  Gun2 := TGun.Create(Self);
  Guns.Add(Gun2);
end;

destructor TPlayer.Destroy;
begin
  { We have to take care to free the list.
    It will automatically free it's contents. }
  FreeAndNil(Guns);

  { No need to free the Gun1, Gun2 anymore. It's a nice habit to set to "nil"
    their references now, as we know they are freed. In this simple class,
    with so simple destructor, it's obvious that they cannot be accessed
```

```
    anymore — but doing this pays off in case of larger and more complicated
    destructors.

    Alternatively, we could avoid declaring Gun1 and Gun2,
    and instead use Guns[0] and Guns[1] in own code.
    Or create a method like Gun1 that returns Guns[0]. }
  Gun1 := nil;
  Gun2 := nil;
  inherited;
end;
```

注意，列表类(list classes) "ownership"机制是简单的,并且，如果你使用一些其他的方法释放实例，你将获得一个错误,尽管它也包含在一个列表中.使用 Extract 方法来从一个列表移除一些东西而不释放它,因此,你自己有责任来释放它.

在 Castle Game Engine 中: 当作为另一个 TX3DNode 的子类(children)插入时,TX3DNode 的派生类(descendants)有自动内存管理. 根 X3D 节点, TX3DRootNode, 通常是依次由 TCastleSceneCore 拥有.其它的一些事物也有一个简单的所有权机制 —查看称为 OwnsXxx 的参数和属性.

5.4. 虚拟析构函数(destructor)被称为 Destroy(销毁)

如你在上面示例所见，当类被销毁时，它的析构函数（destructor）被称为 Destroy(销毁)被调用.

理论上,你可以有多个析构函数(destructors)，但是,在实践中,它几乎从来不是一个好主意.仅有一个称为 Destroy 的析构函数(destructors)更容易, Destroy 依次通过 Free 方法被调用, Free 方法依次通过 FreeAndNil 过程被调用.

Destroy 析构函数(destructors)在 TObject 中被定义为一个*虚拟* 方法, 所以你应该总是在你的类(尽管所有的类衍生(descend)自 TObject)中用 override 关键字标记它. 这使 Free 方法正确地工作. 回想起,虚拟方法方法如何工作,从 虚拟(Virtual)方法, override(重写) 和 reintroduce(再引入).

注意	这个信息是关于*析构函数(destructors)*, 实际上,与*构造函数(constructors)*不符. 它的正常的,一个类有多个构造函数(constructors). 通常它们被称为 Create, 只是有不同的参数, 但是,它也是很好的来为构造函数(constructors)创造其它的名称. 而且, Create 构造函数(constructors)在 TObject 中*不是虚拟的(virtual)*, 所以,你不要在派生物中用 override 标记它. 当定义构造函数(constructors)时,这一切给你一点额外的灵活性.使它们虚拟常常不是不要的, 所以,默认情况下,你不要强制做它. 注意,无论如何,这些对 TComponent descendants 派生物更改. TComponent 定义一个虚拟构造函数(constructors) Create(AOwner: TComponent). 为了流(streaming)系统来工作,它需要一个虚拟构造函数(constructors). 当定义 TComponent 的派生物时, 你应该重写(override)这个构造函数(constructors) (并且用 override 关键字标记它), 并在它的内部执行所有你的初始化.它仍然是很好的来定义附加的(additional) 构造函数(constructors), 但是它们仅应该充当*"帮手(helpers)"*. 当使用 Create(AOwner: TComponent) 构造函数(constructors)创建时,这个实例也应该工作, 否则,当流(streaming)时,它将不能被正确的构造(constructed).*流(streaming)*被使用.例如,当在

	一个 Lazarus 窗体上保存和加载这个组件时.

5.5. 释放 notification(通知)

如果你复制一个引用(reference)到实例, 这样,你有两个引用(reference)到相同的存储器,然后它们中的一个被释放—另一个变成一个*"悬挂指针"*. 它不能被访问,因为它指向到的一个不再被分配的存储器. 访问它可能导致在一个运行时中的错误,或者无用的数据被返回(returned) (因为存储器可以被你的程序中的其它原料(stuff)重新利用).
在这里使用 FreeAndNil 来释放实例没有帮助. FreeAndNil 设置为 nil 仅引用(reference)它获得—没有方法对它来设置所有其他的引用(reference).考虑这代码:

```
var
  O1, O2: TObject;
begin
  O1 := TObject.Create;
  O2 := O1;
  FreeAndNil(O1);

  // what happens if we access O1 or O2 here?
end;
```

1. 在这个语句块的结尾, O1 是 nil 。如果一些代码不得不访问它,它可以可靠地使用 if O1 <> nil ··· 来避免在一个释放的实例上调用方法,像 if O1 <> nil then WriteIn(O1.ClassName);

   尝试访问一个 nil 实例的一个字段导致一个可预见的在运行时(runtime)处的异常.因此,虽然一些代码将不核对 O1 <> nil, 并将盲目地访问 O1 字段, 你将在运行时(runtime)处获得一个明确的异常.
   相同的,去调用一个虚拟(virtual)方法,或调用一个非虚拟(non-virtual)方法来访问一个 nil 实例的字段 .
2. 对于 O2, 事情变的较少地可预见的.它不是 nil, 但是它是无效的.尝试来访问一个非无(non-nil)无效实例的一个字段导致一个不可预见的行为—可能一个访问违例异常,可能一个垃圾数据被返回(returned).

有属于它各种各样的的解决方案:

- 一个解决方案是来,完全地, 仔细的和阅读文档. 不要假设关于引用(reference)的生命时间(lifetime)的任何事,如果它通过其它代码被创建.如果一个类 TCar 有一个字段指向 TWheel 的一些实例,它是一个*惯例(convention)* ,引用(reference)到 *wheel* 是有效的,尽管引用(reference)到 *car* exists 继续存在,并且 *car* 将在它的析构函数(destructor)内部释放它的 *wheels*. 但是这只是一个惯例(convention),文档应该涉及,如果这里有更复杂的一些事进行.
- 在上面的示例中,在恰当地释放 O1 实例后, 你可以简单地明确地设置 O2 变量为 nil.在这个简单的情况下是不重要的.
- 最 future-proof 解决方案是来使用 TComponent 类"释放通知(notification)"机制.一个组件可以被通知(notified),当另外的组件被释放,并因此设置它的引用(reference)为 nil.
  于是你获得一些事,像一个*弱引用(weak reference)*. 它可以对付各种各样的用法事态,

例如你可以让来自类的外部的代码来设置你的引用(reference)，并且外部的代码也可以在任何时间释放实例.

这需要两类都派生自 TComponent. 一般而言使用它归结到调用 FreeNotification , RemoveFreeNotification, 并重写 Notification.

这里是一个完整的示例,显示如何使用这个机制,与构造函数(constructor) /析构函数和一个 setter 属性一起.有时它可以做的更简单,但是这是成熟的版本,换句话说,总是正确的:)

```
type
  TControl = class(TComponent)
  end;

  TContainer = class(TComponent)
  private
    FSomeSpecialControl: TControl;
    procedure SetSomeSpecialControl(const Value: TControl);
  protected
    procedure Notification(AComponent: TComponent; Operation: TOperation); override;
  public
    destructor Destroy; override;
    property SomeSpecialControl: TControl
      read FSomeSpecialControl write SetSomeSpecialControl;
  end;

implementation

procedure TContainer.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited;
  if (Operation = opRemove) and (AComponent = FSomeSpecialControl) then
    { set to nil by SetSomeSpecialControl to clean nicely }
    SomeSpecialControl := nil;
end;

procedure TContainer.SetSomeSpecialControl(const Value: TControl);
begin
  if FSomeSpecialControl <> Value then
  begin
    if FSomeSpecialControl <> nil then
      FSomeSpecialControl.RemoveFreeNotification(Self);
    FSomeSpecialControl := Value;
    if FSomeSpecialControl <> nil then
      FSomeSpecialControl.FreeNotification(Self);
  end;
end;
```

```
destructor TContainer.Destroy;
begin
  { set to nil by SetSomeSpecialControl, to detach free notification }
  SomeSpecialControl := nil;
  inherited;
end;
```

山东世联环保科技开发有限公司

CNWUT Shandong World United Environmental Protection Technology Development Co.,Ltd.

世联环保官方网站：http://www.cnwut.com

感谢 **山东世联环保科技开发有限公司** 资助 Lazarus 书籍的中文化翻译。

希望大家帮助与支持 **山东世联环保科技开发有限公司** 的发展，

为 **世联环保** 提供业务信息，进而支持 Lazarus 中文化发展！