Einsteig in Thorium

Was ist Thorium?

Thorium ist, neben dem chemischen Element #90, eine Scriptsprache, die in FreePascal[1] entwickelt wird. Ich sage deswegen wird, weil Thorium das Ende seiner Entwicklung noch (lange) nicht erreicht hat und einige Dinge noch ausstehen, während viele schon so funktionieren, wie sie sollen und wieder andere noch nicht in allen Situationen getestet wurden und daher Fehler aufweisen könnten.

Warum sollte Ich Thorium benutzen wollen?

Um diese Frage zu beantworten, gehe ich kurz darauf ein, warum ich Thorium überhaupt angefangen habe. Es gibt schließlich eine Menge Scriptsprachen auf dem Markt, noch dazu viele, die sehr ausgereift und featurereich und vor allem schnell sind.

Bevor ich Thorium angefangen habe, habe ich Lua[2] genutzt. Lua ist eine sehr schöne und sehr schnelle Scriptsprache, die als DLL / Shared Object verfügbar ist und für die ich eigens eine Pascal-Unit geschrieben habe, um sie zu verwenden. Ich bin ein Freund der objektorientierten Programmierung, was bei Lua eventuell zum Nachteil geraten kann. Ich hatte mich entschlossen, dem Scriptschreiber möglichst einfachen Zugriff auf meine Objekte und deren Eigenschaften zu bieten, und dazu die Felder und Objekte auf luaspezifische Weise an Lua exportiert. Das hat die Performance aber auch durch die Garbage Collection ziehmlich nach unten gezogen. Weiterhin war das Interface nicht besonders ausgereift. Ich wollte mich nach einer Alternative umschauen.

Ich hatte schon viel von Python gehört, aber ich konnte keine verwendbare Anbindung an FreePascal finden, daher fiel das auch raus. Ansonsten waren mir keine guten, zumindest hostseitig objektorientierten Scriptsprachen bekannt, so dass ich mich entschloss, eine eigene Scriptsprache zu entwickeln: Thorium.

Thorium besitzt in der aktuellen Version folgende Möglichkeiten:

- Aufteilung des Codes in Module.
- Public (für Hostanwendung und andere Module sichtbar) und private (nur für das aktuelle Modul sichtbar) Variablen und Funktionen.
- Eingebautes Stringhandling auf Basis der FreePascal-Strings.
- Zwischenkompilieren in Bytecode, um die Ausführung zu beschleunigen.
- Compile-time Auswertung von konstanten Ausdrücken.
- Optimierung des Bytecodes nach dem Kompilieren.
- Speichern und Laden der Module als Binärdateien.
- Export von Klassen aus der Hostumgebung unter Verwendung von RTTI-Informationen.
- Export von Methoden und Funktionen aus der Hostumgebung ohne die Notwendigkeit von Wrapper-Funktionen.
- Aufteilung der Host-Schnittstelle in einzelne Bibliotheken, die den unterschiedlichen Thorium-Instanzen einzeln zugänglich gemacht werden können, um die Funktionalität genau auf die Skripte zuzuschneiden.
- Und vielleicht noch die ein oder andere Kleinigkeit, die ich hier vergessen habe.

Die Syntax von Thorium ist C-basiert, mit einigen Einflüssen aus Pascal. Eine Revision der Syntax

ist zwar geplant, aber für etwas später. Da einige Teile wirklich über die Zeit hinweg entstanden sind, ist sie nicht perfekt. Aber durchaus benutzbar.

Vorbereitungen

Für Lazarus

Für Lazarus sind bei Thorium zwei Pakete mitgeliefert, die die Verwendung vereinfachen. Diese finden sich im packages-Unterverzeichnis des Downloadpakets. Die beiden .lpk-Dateien werden am besten sofort kompiliert, Installation ist nicht notwenig, da es keine Laufzeitpakete sind.

Damit ist die Installation für Lazarus auch schon abgeschlossen. Wenn Sie nun ein Projekt erstellen, welches Thorium verwenden soll, binden Sie einfach die Pakete als Abhängigkeiten ein. Sie finden den Menüpunkt dafür unter Projekt -> Projektinspektor, wo dann mit einem Klick auf das Plus eine neue Abhängigkeit für thoriumcorepkg hinzugefügt werden muss. Wenn Sie planen, auch die mitgelieferten Hostbibliotheken zu nutzen, fügen Sie auch thoriumlibpkg hinzu.

Für FreePascal ohne Lazarus

Damit Thorium von FreePascal auch gefunden wird, müssen die Paketverzeichnisse im Suchpfad von FreePascal liegen. Lazarus erledigt das automatisch, wenn die Pakete zu den Abhängigkeiten hinzugefügt werden. So müssen die Suchpfade entweder beim Kompilieren manuell über die -Fu Kommandozeilenoption hinzugefügt oder direkt in der fpc.cfg eingetragen werden.

Thorium benutzen

Einen Kontext erzeugen

Um Thorium zu benutzen muss zunächst ein Thorium-Kontext erzeugt werden. Ein Thorium-Kontext ist eine Instanz der Klasse TThorium, welche sich in der thorium.pas findet, die zum Kernpaket von Thorium gehört. Dementsprechend sollten Sie diese Unit erst einmal in die uses-Klausel einfügen. Dann muss noch eine Variable vom Typ TThorium angelegt und initialisiert werden:

```
1.    var
2.      Engine: TThorium;
3.    begin
4.      Engine := TThorium.Create;
5.      try
6.
7.      finally
8.        Engine.Free;
9.      end;
10.   end;
```

Der Try-Finally-Block dient zum Freigeben des Kontextes wenn eine unbehandelte Exception auftritt, was aber eigentlich eher selten der Fall sein dürfte.

Ein Modul laden

In dieser Form ist der Kontext reichlich nutzlos. Zwar können wir ihn erzeugen und auch wieder Freigeben und das sogar ohne Fehler, aber es bringt nicht viel. Hier ein Beispielskript, welches wir gleich mal ausführen werden:

```
1.    loadlibrary "core.std.io"
2.
```

```
3.    public void main()
4.    {
5.      printf("Hello World!");
6.    }
```
Diese Script sollten Sie im Anwendungsverzeichnis als hello_world.tss ablegen.

Ein erfahrener Programmierer wird nun schon erraten können, was dieses Skript tun soll: Es wird Hello World zur Konsole ausgeben (Anmerkung: FreePascal erzeugt normalerweise eine Konsolenanwendung. Falls Sie Lazarus verwenden, stellen Sie sicher, dass die -WG Option nicht übergeben wird („Win32 GUI application“, zu finden in den Compileroptionen in der Kategorie Linker)).

In der ersten Zeile wird die Hostbibliothek core.std.io geladen, welche printf und noch einiges mehr enthält.

Jetzt müssen wir Thorium noch erklären, dass es das Modul laden und ausführen soll. Außerdem müssen wir dem Kontext noch die Bibliothek core.std.io übergeben, damit diese auch geladen werden kann, wenn das Modul sie anfragt.

Dazu muss zunächst die Unit thoriumlibstdio.pas aus dem thoriumlibpkg eingebunden werden. Dann kann mithilfe der Methode LoadLibrary von TThorium die Bibliothek folgendermaßen geladen werden:

```
1.    Engine.LoadLibrary(TThoriumLibStdIO);
```
Ab sofort ist diese dann dem Kontext bekannt und kann von Modulen genutzt werden. Danach laden wir das Modul mit der LoadModuleFromFile-Methode:

```
2.    Engine.LoadModuleFromFile('hello_world');
```
Diese Methode wird versuchen, die Datei hello_world.tss im aktuellen Arbeitsverzeichnis zu öffnen, es sei denn, Sie weisen dem Event OnOpenModule einen Handler zu, der sich mit dem öffnen der Datei beschäftigt. Wenn er keine hello_world.tss finden kann, wird er es mit hello_world.tsb versuchen, was die Standarddateierweiterung für Thorium Binärdateien ist.

Damit wir mit dem geladenen Modul etwas anfangen können, müssen wir entweder den Rückgabewert von LoadModuleFromFile speichern oder aber später auf das Modul über einen Index zugreifen. Die sicherste Methode ist das vorherige Speichern und genau diesen Weg werden wir nun auch gehen. Und zwar brauchen wir dafür eine Variable vom Typ TThoriumModule, die den Rückgabewert aufnehmen kann, z.B.:

```
1.    var
2.      [...]
3.      HelloWorld: TThoriumModule;
4.    begin
5.      [...]
6.        HelloWorld := Engine.LoadModuleFromFile('hello_world');
7.      [...]
8.    end;
```
Sollte beim Laden irgendwas schief gehen, wirft LoadModuleFromFile eine Exception mit einer Fehlerbeschreibung.

Nun müssen wir nur noch die Methode aufrufen. Dazu brauchen wir zunächst eine virtuelle Maschine, die sich mit dem Ausführen des Bytecodes befasst, den Thorium beim Laden und Kompilieren des Moduls erzeugt hat. Den Großteil beim Erzeugen nimmt Thorium uns ab, ein Aufruf der InitializeVirtualMachine Methode genügt. Warum wird das nicht automatisch erledigt? Ganz einfach. Wenn die virtuelle Maschine erstmal erzeugt wurde, darf sich am Thorium-Kontext so gut wie nichts mehr ändern, da es sonst zu Inkonsistenzen im Stack kommen kann, was zu fiesen Crashes führt. Daher darf InitializeVirtualMachine erst nach dem Laden aller Module aufgerufen werden.

Wenn die virtuelle Maschine erstmal initialisiert ist, kann man auch ganz einfach eine Funktion aus

einem Skript aufrufen. Dazu sucht man erstmal die Funktion aus dem passenden Modul raus, mithilfe der FindPublicFunction Methode, welche als einzigen Parameter den (großgeschriebenen) Namen der Funktion erwartet und dafür im Erfolgsfall die Funktion zurückliefert. Diese kann dann mit der Call oder der SafeCall-Methode aufgerufen werden. Die Call-Methode ist etwas schneller, dafür prüft die SafeCall Methode vorher, ob alle Parameter auch richtig übergeben wurden und wirft im Fehlerfall eine Exception. Falsche Parameter können, wenn sie unbeachtet durchrutschen für einige Furore sorgen und den Kontext mächtig aus dem Takt bringen.

```
1.    Engine.InitializeVirtualMachine;
2.    HelloWorld.FindPublicFunction('MAIN').SafeCall([]);
```

Da die virtuelle Maschine automatisch freigegeben wird, wenn der Kontext mittels Free gelöscht wird, brauchen wir auch keinen weiteren try-finally-Block.

Wenn dies nun ausgeführt wird, sollte auf der Konsole ein „Hello World!“ erscheinen.

Wenn Sie das Modul im kompilierten Zustand speichern wollen, müssen Sie einfach nur einen TFileStream mit Schreibrechten auf die gewünschte Zieldatei erzeugen und ihn an die SaveToStream-Methode des Moduls übergeben. Eine Binärdatei kann, wie oben schon angedeutet, ebenfalls durch die LoadModuleFromFile-Methode geöffnet werden, wenn sie die entsprechende Dateierweiterung hat.

Ein etwas komplexeres Skript

Wollen wir nun mal schauen, wie das mit Rückgabewerten und Parametern von Skriptfunktionen abläuft. Dazu nehmen wir mal dieses Beispielskript:

```
1.    public float add(float a, float b)
2.    {
3.      return a + b;
4.    }
5.
6.    public float sub(float a, float b)
7.    {
8.      return a – b;
9.    }
```

und speichern es als param.tss.

Sie laden das Skript wie oben und speichern es in einer Variable namens Param vom Typ TThoriumModule zwischen.

Dann legen Sie noch eine Variable Return vom Typ TThoriumValue an, in die später der Rückgabewert der Funktion gespeichert wird. Diese können Sie nun mit folgendem Methodenaufruf aufrufen:

```
1.    Return :=
Param.FindPublicFunction('ADD').SafeCall([ThoriumCreateFloatValue(1.0),
ThoriumCreateFloatValue(2.0)]);
```

In Return.BuiltIn.Float sollte nun das Ergebnis 3.0 zu finden sein. Sie könnten das zum Beispiel mit einer Konsolenausgabe überprüfen.

Ebenso ließe sich auch die sub-Methode aufrufen und das Ergebnis abfragen.

Abschluss

Dies war ein Kurzeinstieg in Thorium, um Sie mit den Grundfunktionen der Skriptsprache vertraut zu machen. Mit diesem Wissen, ein bisschen Neugierde und den Quelltexten für die Bibliotheken im thoriumlibpkg sollten Sie schon einiges anstellen können.

1 <http://www.freepascal.org>
2 <http://www.lua.org>