

# zDefine 过程定义详解

zDefine.inc 是全局的，几乎在以上所有开源项目中都会引用它，zDefine 主要对编译过程做预处理使用。

文档版本:2.0

最后更新:2024-3

更新日志

大规模更新 Z.Define.inc 内容

## 目录

在 FPC 编译器中,全程序一律使用 FPC_DELPHI_MODE 定义 .....	3
FPC 独占参数: {\$MODESWITCH AdvancedRecords} .....	3
FPC 独占参数: {\$NOTES OFF} .....	3
FPC 独占参数: {\$STACKFRAMES ON}.....	3
FPC 独占参数: LITTLE_ENDIAN, BIG_ENDIAN .....	3
构建定义: FirstCharInZero.....	4
构建定义: OVERLOAD_NATIVEINT .....	4
构建定义: FastMD5 .....	4
构建定义: OptimizationMemoryStreamMD5 .....	4
构建定义: parallel.....	5
构建定义: FoldParallel.....	5
构建定义: SystemParallel .....	5
构建定义: InstallMT19937CoreToDelphi.....	6
构建定义: MT19937SeedOnTComputeThreadIs0 .....	6
构建定义: SSE_Optimize_Disntance_Compute.....	6
构建定义: Intermediate_Instance_Tool .....	7
构建定义: Core_Thread_Soft_Synchronize.....	7
构建定义: Tracking_Dealy_Free_Object .....	7
构建定义: DoubleIOFileSystem .....	7
构建定义: LimitMaxComputeThread .....	7
构建定义: LimitMaxParallelThread .....	7
构建定义: ZNet_C4_Auto_Repair_First_BuildDependNetwork_Fault .....	7
构建定义: ZDB2_Thread_Engine_Safe_Flush.....	8
构建定义: C4_Safe_Flush .....	8
构建定义: Z_AI_Dataset_Build_In.....	8
构建定义: ZDB_BACKUP .....	9
构建定义: ZDB_PHYSICAL_FLUSH .....	9
构建定义: SMALL_RASTER_FONT_Build_In, LARGE_RASTER_FONT_Build_In .....	10
构建定义: CriticalSimulateAtomic .....	11
构建定义: SoftCritical .....	11
构建定义: ANTI_DEAD_ATOMOMIC_LOCK .....	11
构建定义: ZNet 的默认物理 IO .....	11
构建定义: FillPtr_Used_FillChar .....	12
构建定义: CopyPtr_Used_Move .....	12
构建定义: UsedSequencePacket .....	13
构建定义: UsedSequencePacketOnP2PVM.....	13
构建定义: initializationStatus .....	13
Delphi 中的 Debug 与 Release .....	14

## 在 FPC 编译器中,全程序一律使用 FPC\_DELPHI\_MODE 定义

在 20 行左右可以看到对 FPC 编译器的条件判断，如果是 FPC 编译器，也会走 Delphi 语法

```
{IFDEF FPC_DELPHI_MODE}
{$MODE delphi}
{$ELSE FPC_DELPHI_MODE}
{$MODE objfpc}
{$ENDIF FPC_DELPHI_MODE}
```

## FPC 独占参数：{\$MODESWITCH AdvancedRecords}

在结构体中可以包含方法的开关，使用 FPC 构建代码时，必须打开

## FPC 独占参数：{\$NOTES OFF}

因为 fpc 构成代码会产生大量参考信息，比 delphi 还多，这是关闭不必要的提示信息

## FPC 独占参数：{\$STACKFRAMES ON}

堆栈框架，如果我们需要在 lazarus 环境对代码 debug 一下，这是需要打开的。

如果我们使用 fpc 做 release 版本的构建，可以关闭这个参数，同时配合 -o3 开关，对全程序进行优化构建，可以小幅提速

## FPC 独占参数：LITTLE\_ENDIAN, BIG\_ENDIAN

只有 fpc 的编译器，才能构建早期小型机的大端字节序数据结构，现在几乎已经统一的使用了小端字节序

它主要影响的库是 CoreCipher.pas，这是一套内核加解密库，在默认情况下，它只能工作在小端字节序的 cpu 上。该参数是自动化生成的，我们不需要去刻意更改它。

在我们使用 delphi 构建程序时，包括 ios+android+x86 这类体系，默认都使用的是小端字节序。

## 构建定义：FirstCharInZero

当我们做字符串遍历，Copy，计算这类处理时，标准库的首个字符的逻辑位置是否从 0 开始  
标准库的 pascal 字符串的首个位置都是从 1 开始，后来，由 emb 提出了新的数据结构，让  
字符串从 0 开始处理，并且应用到了 IOS,Android 这类平台中

它主要影响的库：PascalStrings.pas, UpascalStrings.pas

当我们在 fpc 构建程序时：{\$UNDEF FirstCharInZero}

当我们在 delphi 构建 ios 和 android 程序时：{\$DEFINE FirstCharInZero}

当我们在 delphi 构建 windows 和 linux 程序时：{\$UNDEF FirstCharInZero}

## 构建定义：OVERLOAD\_NATIVEINT

是否将 NativeInt,NativeUInt 视为一个独立的原子变量对待，在 FPC 编译器中，NativeInt 被指定成 Integer or Int64，当我们重载 NativeInt 时会与 Integer 发生冲突，Delphi 则是将 NativeInt 视为一个独立原子变量对待

## 构建定义：FastMD5

MD5 在数据验证中的使用非常频繁，这个定义代表是否使用宏汇编构建的加速 MD5。它只能工作于 windows 平台下，并且只能在 delphi 中打开它。我们的程序如果使用 fpc 构建不可以打开它。它主要影响的库是：FastMD5.pas

*MD5 有一个分组的变换步骤，既按 512 比特位输入长度进行 ABCD 一共 16 轮的非线性变换操作，这一步操作如果使用 pure pascal 实现，会非常耗时，后来 maximmasiutin 提出了 x64 版本的宏汇编 16 轮非线性的实现，同时他也给出了早期 1994 年的 x86 参照版本，后来被我引用于 FastMD5*

<https://github.com/PassByYou888/FastMD5>

[https://github.com/maximmasiutin/MD5\\_Transform-x64](https://github.com/maximmasiutin/MD5_Transform-x64)

---

## 构建定义：OptimizationMemoryStreamMD5

在基础库中，UnicodeMixedLib.pas, FastMD5.pas，都有面向 Stream 计算 MD5 的支持，这个定义是在 TStream 是 TmemoryStream 或则 TmemoryStream64 时，它会自动绕过 stream 的流式数据 copy，直接基于指针来高速计算 MD5。

在多数情况下，我们尽可以打开它，它可以支持 fpc 和 delphi，它能正常工作于任何操作系统。

## 构建定义：parallel

这是并行的构建定义，打开它以后，多核 `cpu` 可以被充分利用起来。如果是目标程序跑在手机这类平台上，这个定义是可以关闭的，手机上的并行程序并不会得到太明显的加速。并行模式的程序相比于普通程序，提速大概在 2-8 倍间，具体提速取决于北桥芯片组和 `cpu` 核心的工作频率

关闭并行程序，多数情况下，是在手机，IOT 这类平台中，或则是我们需要调试程序时。`Parallel` 影响到的库很多，这里无法一一列举。

在多数情况下，我们尽可以打开它，它只可以 `delphi`，它能正常工作于任何操作系统。

因为 `FPC` 的核心底层并没有很好解决原子锁问题，并行程序会出现不稳定的情况。在 `FPC` 中，`Parallel` 机制无效，它永远都是单线程在工作。

2019 年 11 月版本更新：在 `FPC+Lazarus` 中弃用了 `mtprocs.pas` 并行支持单元，并行支持由函数 `FPCParallelFor` 来支持，它基于 `TComputeThread` 工作，可靠性优于 `mtprocs`

## 构建定义：FoldParallel

并行程序采用折叠方式选址，如果关闭会采用分块方式选址，折叠选址对于多核 `CPU` 的性能挖掘会更加深入，大规模计算性能优于分块选址。假如并行计算量很小，分块选址则更快。

折叠选址 1 到 8

- 线程一：1, 3, 5, 7
- 线程二：2, 4, 6, 8

分块选址 1 到 8

- 线程一：1, 2, 3, 4
- 线程二：5, 6, 7, 8

## 构建定义：SystemParallel

该定义只针对 `delphi`，是否让 `DelphiParallelFor` 使用 `Delphi` 内置的 `TParallel.For`，如果打开该定义，`DelphiParallelFor` 会完全基于 `TParallel.For` 来工作，关闭该定义后，`DelphiParallelFor` 会基于 `TComputeThread` 来工作

在小粒度并行中，`TParallel.For` 的性能会优于 `TComputeThread`

在 `TMemoryRaster` 形态学支持系统中，`TParallel.For` 的性能与 `TComputeThread` 无差异

线程池机制说明

内核线程池机制几经周折，现在形成了一种成熟机制，当我们不使用 `SystemParallel` 时，内核线程池即开始工作，线程启动不会是重新创建一个新实例，而是检查可以回收的线程，该机制在 `CoreComputeThread.inc` 库中实现，感兴趣可以自行研究。

## 构建定义：InstallMT19937CoreToDelphi

替代 Delphi 原有的古典随机数(random 函数)，使用梅森旋转算法的高质量随机数  
梅森旋转算法的周期为  $2^{19937}-1$  次，分布率优于古典随机数

InstallMT19937CoreToDelphi 的来历细节

在工程统计学中，随机数经常被用于算子，当我们在多线程和并行程序大量使用 Random 函数时会出现：Random()=Random() 的情况，因为古典随机数周期短和重复率高。

从各种开源项目寻找很久解决方案，在标准 STL 库，MT19937 是一种标准随机数，它工作于对象容器中，我们可以在不同的线程创建 MT19937 的实例类来使用，这不会出现上述 random()=random()的情况，而它的做法如下：

```
r:=TRandomEngine.create  
X:=r.random
```

如果我们去修改程序的随机数程序机制，显然这很麻烦，因为需要修改的程序代码太多了，zAnalysis 都是几十万行以上的代码量，并且改算法代码很容易踩地雷。

处于以上考虑，我们需要设计一种不需要改动程序，同时又能兼容并行计算的方法：这时候，我开始动手编写新的并行线程随机数，而这种工作又很多，这时候，我直接放弃使用 delphi 的古典随机数，选择了梅森素数+并行线程随机数。（以解决短周期并行线程随机数为主，MT19937 是辅助的计算方法，MT19937 并不能解并行程序中的短周期，它只能做到比古典随机数周期更大）

InstallMT19937CoreToDelphi 的实现细节位于 Core\_MT19937.inc 文件中，参考资料已备注

当我们打开 InstallMT19937CoreToDelphi 构建定义后，Delphi 的 Random 在并行线程的周期将会一致。大规模统计学计算不会再出现算子碰撞的问题，其它使用 random 的程序也不会出现短周期问题

## 构建定义：MT19937SeedOnTComputeThreadIs0

在 TComputeThread 启动我们的线程调用时，是否要对 MT19937 的随机数种子置 0。关闭该定义后在 TComputeThread 启动时，会使用 TimeTick 作为 MT19937 的随机数种子。

## 构建定义：SSE\_Optimize\_Disntance\_Compute

使用 SSE 加速 Distance 函数计算,主要用于 KDTree 加速搜索和排序.默认为自动判断平台和硬件环境,在 win 平台默认打开,其余平台一律关闭.

## 构建定义：Intermediate\_Instance\_Tool

Z 系内核库一律使用 TCore\_Object\_Intermediate 作为中间层,中间层在构建和破坏时会影响全局的实例计数器.关闭定义后,创建对象不会更新计数器,可以极小幅度提速.打开以后可以为中数据中心提供服务器运行状态的分析功能.默认为关闭.具体使用方法详见 ZNet 手册.

## 构建定义：Core\_Thread\_Soft\_Synchronize

Z 系内核在多线程中提供自定义 Soft Synchronize,打开以后 TCompute.Sync 方法将不再使用 RunTime 库的 Thread.Synchronize 方法.默认为打开. 具体使用方法详见 ZNet 手册.

## 构建定义：Tracking\_Dealy\_Free\_Object

Z.Notify 为延迟事件支持库,当 Tracking\_Dealy\_Free\_Object 被打开,每次使用 Z.Noitfy 延迟释放实例时将会抛出一个消息.默认为关闭.

## 构建定义：DoubleIOFileSystem

当打开该定义,ZNet 的双通道会含有文件系统.默认为打开.

## 构建定义：LimitMaxComputeThread

打开该定义以后 TCompute 每次启动线程实例时将会被最大线程(CpuCount\*20)所限制.默认为关闭.

## 构建定义：LimitMaxParallelThread

限制最大并行计算线程,默认为打开,Z 系内核最大并行线程为 8 个.  
具体使用方法详见 ZNet 手册.

## 构建定义：ZNet\_C4\_Auto\_Repair\_First\_BuildDependNetwork\_Fault

打开该定义以后,当 C4 框架首次入网时出现物理完了过无法连接会重复连接.默认为打开.

## 构建定义：ZDB2\_Thread\_Engine\_Safe\_Flush

打开该定义后,ZDB2 每次往硬盘写入数据都不会直接访问 IO,而是写入到临时内存区,只有当 flush 被调用时才会写入 IO,当中途断电写入失败时 ZDB2 会自动修复.默认为打开.  
具体使用方法详见 ZNet 手册.

## 构建定义：C4\_Safe\_Flush

打开该定义后,在 C4 框架中所有使用 ZDB2 模块,都会启用防断电机制.默认为打开.

## 构建定义：Z\_AI\_Dataset\_Build\_In

```
// automated loading common AI data sets on boot-time  
// {$DEFINE Z_AI_Dataset_Build_In}
```

在 ZAI 项目中,有一些是已经训练好的应用模型,Z\_AI\_Dataset\_Build\_In 是我们编译的 EXE 是否会包含这类模型的数据。

当我们打开这个定义后,EXE 将会增加 80M 左右的体积。

如果没有打开 Z\_AI\_Dataset\_Build\_In,EXE 在启动时,会从 EXE 当前目录或则我的文档,去寻找一个名字叫 AI\_BuildIn.OXC 的文件,如果没有找到,会使用 DoStatus 提示

Z\_AI\_Dataset\_Build\_In 影响到的库是: zAI.pas

如果我们构建 ZAI 发行程序,那么可以打开这个构建开关

如果我们的程序没有引用 zAI.pas 可以无视 Z\_AI\_Dataset\_Build\_In 定义

*AI\_BuildIn.OXC 是使用 FilePackage 这类工具制作而出的文件包,类似 zip,rar*

---



## 构建定义：ZDB\_BACKUP

```
// ZDB_BACKUP is automatically made and replica caching is enabled.  
// usage ZDB_BACKUP so slows the open of large size ZDB file, after time, but does is high performance.  
// {$DEFINE ZDB_BACKUP}
```

该定义基于 ZDB 最底层的 `ObjectData.pas` 工作，如果以文件方式打开 ZDB 数据库，包括.OX 文件，会先备一份一次数据，当正常关闭 ZDB 数据库以后，备份才会被删除。

如果 ZDB 发生损坏，会自动化的从备份文件去恢复。

该定义对于备份文件尺寸小于 1GB 的 ZDB 数据库会比较安全，而文件尺寸太大时，备份就会非常耗时，这时候，我们需要关闭这个开关，只能使用回写式的 ZDB 擦写数据。

在多数情况下，这个开关不需要打开

如果我们的应用程序，使用了 ZDB，而 ZDB 文件又都不大时，这时候才可以打开该参数。

定义 ZDB\_BACKUP 后，打开 ZDB 将会变慢，而数据会更加安全

## 构建定义：ZDB\_PHYSICAL\_FLUSH

该定义基于 ZDB 最底层的 `TObjectDataManagerOfCache` 工作，在触发 Flush 写入硬盘数据库前，会先创建一个临时回写文件 `~flush`，当存储中断(断电，蓝屏，程序崩溃)，`~flush` 会在下一次打开 ZDB 数据时自动修复。

该选项会让 ZDB 的物理写入性能降低 10%，而数据安全性会更高。假如我们在做大数据的导入导出，那么可以关闭它。假如我们在运行后台数据库，我们可以打开它。

## 构建定义: SMALL\_RASTER\_FONT\_Build\_In, LARGE\_RASTER\_FONT\_Build\_In

```
// With SMALL_RASTER_FONT_Build_In and LARGE_RASTER_FONT_Build_In, boot-time memory usage increase by 100M-200M and start-up time to be delay 100ms  
// {$DEFINE SMALL_RASTER_FONT_Build_In}  
// {$DEFINE LARGE_RASTER_FONT_Build_In}
```

在 TMemoryRaster 中, 绘制字体都使用内置的光栅, 与操作系统和平台无关。  
这些内置光栅都是独立的压缩文件

### SMALL\_RASTER\_FONT\_Build\_In

包含了中文简繁体体的内置光栅

光栅字符量:21033

光栅尺寸: 7208 x 9198

压缩后文件尺寸大约 10M

### LARGE\_RASTER\_FONT\_Build\_In

包含了中文简繁体与日韩字体的光栅

光栅字符量:48664

光栅尺寸: 10257 x 13923

压缩后文件尺寸大约 20M

SMALL\_RASTER\_FONT\_Build\_In 与 LARGE\_RASTER\_FONT\_Build\_In 我们只能指定其中一个, 如果两个都定义, 构建时会优先选择 LARGE\_RASTER\_FONT\_Build\_In。这也会导致 EXE 或则 APP 的体积增大 10-20M 左右

如果我们的发行程序引用了 memoryRaster.pas, drawEngine.pas,ZAI.pas 这类库, 那么我们即可指定使用哪种形式的内置字体光栅

如果 SMALL\_RASTER\_FONT\_Build\_In 与 LARGE\_RASTER\_FONT\_Build\_In 都没有定义, 那么 EXE 或则 App 在启动时会从当前 EXE 或则 APP 目录去寻找文件, 这些文件文件分别是, MemoryRasterLargeFont.zFont, MemoryRasterFont.zFont, 如果两个文件都存在, EXE 或则 APP 会优先使用 MemoryRasterLargeFont.zFont。如果这两个文件都不存在, 那么 EXE 或则 APP 会使用 MINI 光栅字体

*MINI 光栅字体不会包含中文简繁体体和日韩字体, 只会包含标准 ASCII 英文字体, MINI 光栅字体非常小, 以代码形式存储, 不使用文件存储, 它只会在下面两个定义*

**SMALL\_RASTER\_FONT\_Build\_In**

**LARGE\_RASTER\_FONT\_Build\_In**

*都没有定义时才存在*

---

## 构建定义: CriticalSimulateAtomic

使用互斥机制来模拟原子锁, 如果我们没有涉及很深入的并行程序, 多线程这类方式编程, 我可以不用管它。如果我们的 delphi 程序大量使用并行机制, 并且大量计算, 那么我们在移植到 FPC 时可以打开该定义。因为 FPC 的原子锁支持机制与 Delphi 不一样

## 构建定义: SoftCritical

放弃使用系统自带的互斥机制, 而使用软件形式的互斥机制。

系统自带的互斥机制, 在卡线程时, cpu 消耗为 0, 软件形式的互斥机制则会让工作核心满负荷, 软件机制通过不停检查状态机实现卡线程。

软件互斥机制只用于构建过程中的调试

## 构建定义: ANTI\_DEAD\_ATOMIC\_LOCK

该定义只在 SoftCritical 被定义以后才有效果, 它可以防止死锁, 多用于构建并行程序时寻找卡线程位置时使用。如果我们在构建 ZServer 的通讯接口, 比如我们要增加一个新的物理 IO, 则这个开关可以打开用于调试。

## 构建定义: ZNet 的默认物理 IO

因为使用 ZServer 需要指定使用一个物理 IO, 而这些物理 IO 可以任意, 处于简单和统一性, ZServer 内置了一个 PhysicsIO.pas 库, 一旦我们引用它, 我们即可使用在 zDefine.inc 定义的物理 IO, 这些定义分别如下:

- PhysicsIO\_On\_ICs9: PhysicsIO.pas 库默认使用 ICs9 作为物理 IO
- PhysicsIO\_On\_ICs: PhysicsIO.pas 库默认使用 ICs8 作为物理 IO
- PhysicsIO\_On\_CrossSocket: PhysicsIO.pas 库默认使用 CrossSocket 作为物理 IO
- PhysicsIO\_On\_DIOCP: PhysicsIO.pas 库默认使用 DIOCP 作为物理 IO
- PhysicsIO\_On\_Indy: PhysicsIO.pas 库默认使用 Indy 作为物理 IO, 在 Delphi 构建手机平台时, PhysicsIO.pas 库会默认使用 Indy 作为物理 IO
- PhysicsIO\_On\_Synapse: PhysicsIO.pas 库默认使用 Synapse 作为物理 IO, 在 fpc 中, 我们不需要在 zDefine.inc 再做定义, PhysicsIO.pas 默认使用 Synapse 作为物理 IO

## 构建定义: FillPtr\_Used\_FillChar

该定义是内核 FillPtr 函数直接使用 FillChar (fpc/delphi 内置填充) 而绕过内置实现。

说明:

内置实现的 FillPtr 使用纯粹的 pure pascal 填充, 实现代码位于 CoreClasses.pas 库。

FillChar 在 delphi/fpc 的基础库使用 x86/x64 对应的 IA32/64 汇编语言实现, 在物理设备中它的速度非常快, 大约是 pure pascal 的 1.8 倍。但是在 vmware/box 这类同构机中 pure pascal 实现的 FillPtr 则更快, 大约汇编 3 倍左右。

该定义只是对于原子 copy 操作的仔细研究分析。实际使用时, 是否打开该定义对于 app/service 没有影响。如果运行 HPC 程序, 瓶颈都在于并行程序工作方式, 可以忽略局部的小优化。

在默认情况下, 该定义是关闭的。

## 构建定义: CopyPtr\_Used\_Move

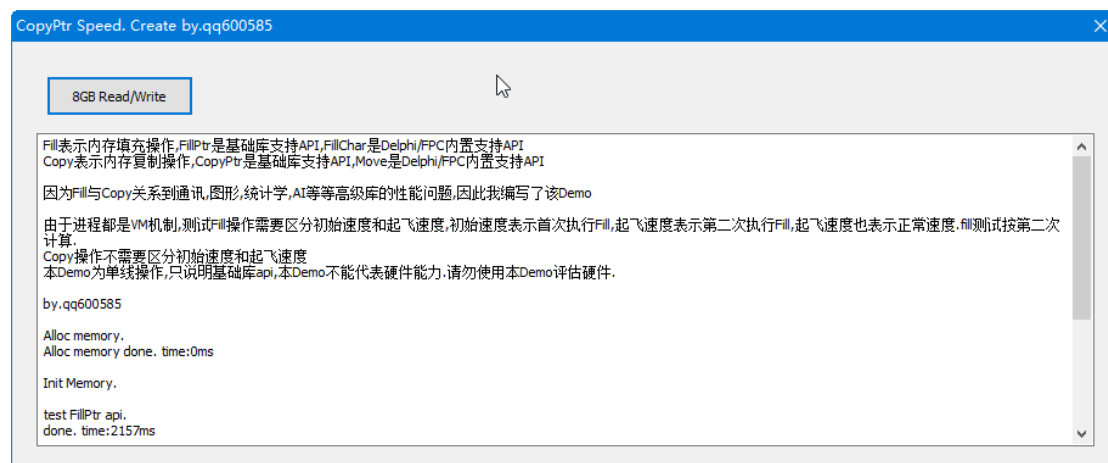
CopyPtr 是内置的原子内存块复制 api, 是实现代码位于 CoreClasses.pas 库。

该定义用 fpc/delphi 内置的 move 替代 CopyPtr。

由于 cpu 的频率, 数据传输带宽, 高于高频率内存, 所以我并没有测试出这两者的差异。

该函数在开源项目 /zAI 等项目中, 有一个叫 CopyPtrSpeed 的测试 demo, 充分说明了该定义对于应用程序在物理硬件设备中的影响是微乎其微的。

在同构虚拟机中, pure pascal 版本的实现会优于汇编语言。



在默认情况下, 该定义是关闭的。

## 构建定义：UsedSequencePacket

ZServer 的物理模型默认情况下将会使用序列包机制，这样可以很好的解决 keep-alive 问题。如果关闭该定义，ZServer 的物理网络性能将会非常高，可以适应万兆以太网需求，而 keep-alive 系统将会完全依赖于系统 socket 支持。

在我们构建网络通讯系统时，UsedSequencePacket 的定义必须在通讯双方一致，否则将会发生无法建立连接问题。

## 构建定义：UsedSequencePacketOnP2PVM

该定义指 P2PVM 内置的 IO 是否使用序列包机制，默认构建是关闭的。因此 P2PVM 的工作效率非常高，有适应万兆以太网能力。

构建网络通讯系统时，服务器/客户端应该保证该定义一致，否则，不能正常通讯。

## 构建定义：initializationStatus

app 通过 IDE->Debug 启动后，该定义会提示初始化状态，只有提示作用。

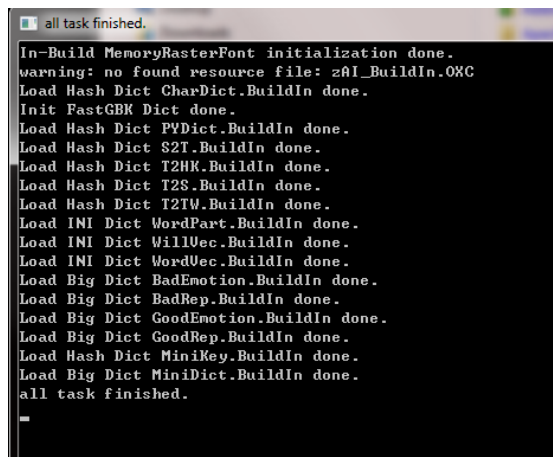
这些状态信息包括如下：

- ZAI 的 In-Build 数据库提示

- ZAI 成功激活状态提示

- MemoryRaster 的光栅字体初始化状态

- GBK 初始化的状态提示



```
all task finished.
In-Build MemoryRasterFont initialization done.
warning: no found resource file: zAI_BuildIn.OMC
Load Hash Dict CharDict.BuildIn done.
Init FastGBK Dict done.
Load Hash Dict PYDict.BuildIn done.
Load Hash Dict S2T.BuildIn done.
Load Hash Dict T2HK.BuildIn done.
Load Hash Dict T2S.BuildIn done.
Load Hash Dict T2TW.BuildIn done.
Load INI Dict WordPart.BuildIn done.
Load INI Dict WillVec.BuildIn done.
Load INI Dict WordVec.BuildIn done.
Load Big Dict BadEmotion.BuildIn done.
Load Big Dict BadRep.BuildIn done.
Load Big Dict GoodEmotion.BuildIn done.
Load Big Dict GoodRep.BuildIn done.
Load Hash Dict MiniKey.BuildIn done.
Load Big Dict MiniDict.BuildIn done.
all task finished.
-
```

# Delphi 中的 Debug 与 Release

Debug 模式会允许生成 Debug info，即编译器生成的代码信息，会让 EXE,DLL 更大，同时 Debug 没有+O 选项，诸如 Move,TmemoryRaster 的投影等操作，更加耗时。

Release 会默认打开+O 选项，构建的应用会快于 Debug，同时 Release 会打开自动化的 Inline 优化，影响平台：windows(32+64)，linux

Release 在 delphi 构建手机平台应用会关闭优化(-o)，我们尽可以使用 release 正常构建手机应用，这是无代码优化的。它会影响到各种安卓模拟器，安卓真机，苹果 IOS

在 FPC 中，所有的程序默认都会使用 release 方式构建

测试 Release 和 debug 性能差异，可以使用 CoreClasses.pas 中的 CopyPtr, FillPtrByte 函数，亦可以使用 TMemoryRaster 中的 Projection 方法，性能差异大概有 20%，视北桥芯片和 cpu 频率而定。

完

By.qq600585