**地图编码研究记录**

HDTT

目录

[一、前言 1](#_Toc9238663)

[1.1 License 1](#_Toc9238664)

[1.2 不懂代码也能读程序 2](#_Toc9238665)

[1.2.1 Class 2](#_Toc9238666)

[1.2.1.1 定义 2](#_Toc9238667)

[1.2.1.2 构造 3](#_Toc9238668)

[1.2.1.3 C＃构造函数 3](#_Toc9238669)

[二、重识地图 3](#_Toc9238670)

[1.3 关于本文的声明 4](#_Toc9238671)

[三、程序 4](#_Toc9238672)

[3.1 背景介绍 4](#_Toc9238673)

[3.2 文件的定义 5](#_Toc9238674)

[3.2.1 虚拟文件系统 5](#_Toc9238675)

[3.2.2 VirtualTextFile.cs 7](#_Toc9238676)

[3.2.2 MemoryFile 7](#_Toc9238677)

[3.3 数据格式 8](#_Toc9238678)

[3.3.1 MapObject.cs 8](#_Toc9238679)

[3.3.1.1 综述 8](#_Toc9238680)

[3.3.1.2 IsoTile 9](#_Toc9238681)

[3.3.1.3 Overlay 10](#_Toc9238682)

[3.3.2 TileLayer.cs 10](#_Toc9238683)

[3.3.3 MapFile.cs 12](#_Toc9238684)

[3.3.3.1 定义 12](#_Toc9238685)

[3.3.3.2 初始化地图的函数 13](#_Toc9238686)

[3.3.3.2 ReadTile 15](#_Toc9238687)

[四、编码方式 17](#_Toc9238688)

# 一、前言

## 1.1 License

本文内容源于CNC\_Maps\_Renderer开源项目的部分代码。

<https://github.com/zzattack/ccmaps-net>

首先按照开源协议，应附程序License。  
  
"  
以下许可证仅适用于程序中不包含位于源文件顶部的冲突许可证的那些部分。这些包括OpenRA和XCC项目的部分，这些部分是根据GPL v3许可的。  
  
（麻省理工学院许可证）  
  
版权所有（c）2007-2013 Frank Razenberg  
  
特此授予任何获得本软件和相关文档文件（“软件”）副本的人免费许可，无限制地交易本软件，包括但不限于使用，复制，修改，合并的权利根据以下条件，出版，分发，再许可和/或出售本软件的副本，并允许向其提供本软件的人员这样做：  
  
上述版权声明和本许可声明应包含在本软件的所有副本或实质部分中。  
  
本软件按“原样”提供，不提供任何明示或暗示的担保，包括但不限于适销性，适用于特定用途和不侵权的担保。在任何情况下，作者或版权所有者均不对任何索赔，损害或其他责任承担任何责任，无论是在合同，侵权行为还是其他方面的行为，由本软件引起或与之相关，或与本软件的使用或其他交易有关。软件。  
"

## 1.2 不懂代码也能读程序

文件格式部分基本全是定义的类，不会代码不要紧，能看懂英文，当成自然语言来读就好了。

以下来源于Microsoft关于Csharp的教程。

### 1.2.1 Class

#### 1.2.1.1 定义

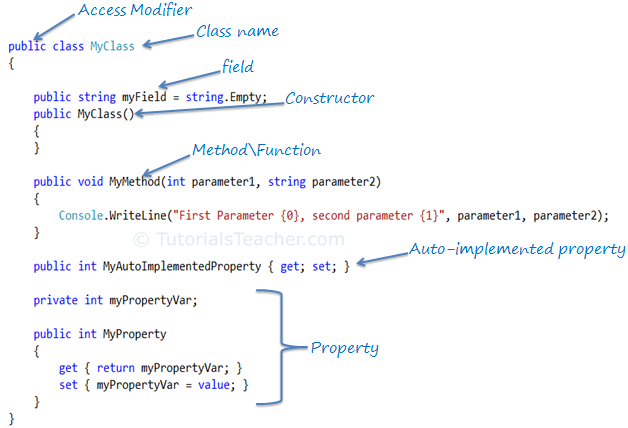
类就像特定对象的蓝图。在现实世界中，每个物体都有一些颜色，形状和功能。例如，豪华车法拉利。法拉利是豪华车型的对象。豪华车是指定速度，颜色，形状，内饰等特定特征的类别。因此，任何制造满足这些要求的汽车的公司都是豪华车型的对象。例如，宝马，兰博基尼，凯迪拉克的每一辆车都是名为“豪华轿车”的车型。在这里，“豪华车”是一个级别，每一辆实体车都是豪华车类的对象。

同样，在面向对象的编程中，类定义了某些属性，字段，事件，方法等。类定义了数据的种类和它们的对象将具有的功能。

通过类，您可以通过将其他类型，方法和事件的变量组合在一起来创建自己的自定义类型。

在C＃中，可以使用class关键字定义类。

1.2.1.2 构造



#### 1.2.1.3 C＃构造函数

类可以具有参数化或参数较少的构造函数。创建类的实例时将调用构造函数。可以使用访问修饰符和类名来定义构造函数：<access modifiers> <class name>(){ }

这个说的是如何快速定义一个东西为一个特定的类。

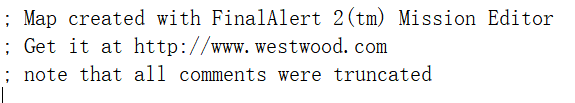
# 二、重识地图

要编程，就要从数据结构的角度分析地图。

首先是[标题]这种，它在程序里相当于字典的索引，在程序中查找[标题]，来读取内容。

然后是地图的注释，在yrm里，“;”是注释符号，

这一点从地图的开头就能看出:



注释是什么意思呢，就是告诉程序不要读这行。

然后是很关键的，[IsoMapPack5]和[OverLayPack]

可以看到它们的格式是……一堆看起来是乱码的东西。但很显然，这些“乱码“只包含0~9、a~z，A~Z和+号和/号。

它们其实是64位编码！

附一下最初找到的资料，给了我很大帮助：

<https://www.modenc.renegadeprojects.com/IsoMapPack5>

<http://www.shikadi.net/moddingwiki/Command_%26_Conquer_Tileset_Format>

具体而言，是把地图的每一块Tile的参数合成一个bytes格式的串，然后再把所有Tile的串合成一个总串，最后再把这个串压缩成64位编码。

这里我必须说一下，IsoMapPack5使用的是miniLZO解压缩方法，而OverLayPack使用的是LCW算法，也就是所谓的format80算法。这是很重要的信息。

## 1.3 关于本文的声明

本文内容源于CNC\_Maps\_Renderer开源项目的部分代码。

<https://github.com/zzattack/ccmaps-net>

本文的目的是总结国内外资料，便于有意向研究地图编码及其他问题的研究者迅速掌握原理和架构。在我看来，有开源程序和已知的地图的解读方式已经是个奇迹。

本文的编写者是HDTT，遵循开源程序CNC\_Maps\_Renderer的License。

**因此，本文是可以由他人自由编辑、复制、发表的。但是，如果需要长篇复制，请保留本文1.3 关于本文的声明。**

# 三、程序

## 3.1 背景介绍

lz发现zzattack写的地图截图器CNC\_Maps\_Renderer是开源的，于是开始从逆向研究变成读代码。  
为什么要读这个软件呢？因为它能读取yrm文件并渲染出游戏中的地图样式，说明它是具有解码地图、处理地图文件、调用游戏内部文件、渲染地图显示的全套功能的，理解了它就理解了地编乃至游戏对地图的读取、编辑、保存、显示。  
  
再加上相比Xcc Mixer的c++项目，它是用Csharp写的。csharp的特点是函数的定义和声明都要写在一起，相比C++只给你一个空荡荡的头文件好读得多。  
首先介绍一下这个开源项目的直观架构。  
项目链接为：<https://github.com/zzattack/ccmaps-net>  
主文件包括  
· CNCMaps.Engine(游戏引擎，来自openRA项目)  
· CNCMaps.FileFormats（地图文件格式相关）  
下面三个我没看↓  
· CNCMaps.Objects.Westwood  
· CNCMaps.Renderer.GUI  
· CNCMaps.Renderer  
  
其中，FileFormat文件夹里的代码是对我最重要的，因为其详细记录了地图的储存方式和相关函数。  
在该目录下包括三个文件夹:  
Encodings（与地图有关的一系列编码函数）  
Map（储存了地图文件格式）  
VirtualFileSystem（虚拟文件格式，与地图有关的底层文件格式）  
  
接下来将依次从底层格式、与地图有关的数据结构、与地图有关的操作函数来叙述。

## 3.2 文件的定义

### 3.2.1 虚拟文件系统

首先是介绍虚拟文件系统（VirtualFileSystem）。该文件夹中，和地图有关的文件是VirtualFile.cs、[VirtualTextFile.cs](https://github.com/zzattack/ccmaps-net/blob/master/CNCMaps.FileFormats/VirtualFileSystem/VirtualTextFile.cs" \o "VirtualTextFile.cs)、MemoryFile.cs。

所谓VirtualFile，也就是虚拟文件，意思就是给一个类(Class)定义写入、读取、查找三大基本功能，这个类不就相当于一个文件了吗。之所以要重新重新定义，是因为地图有其特有的文件结构，按着结构定义一个虚拟文件格式，会大大简化主函数的写法。

首先介绍最底层的VirtualFile.cs文件。它的部分定义如下：

（不想看代码可以直接跳过代码部分）

public class VirtualFile : Stream {//Stream是一个抽象类。有写入、读取、查找三个功能

        public Stream BaseStream { get; internal protected set; }//TODO Returns the underlying stream.

        protected int BaseOffset;

        protected long Size;

        protected long Pos;

        virtual public string FileName { get; set; }

        //In C#, for overriding the base class method in a derived class, you have to declare a base class method as virtual and derived class method asoverride:

        byte[] \_buff;

        readonly bool \_isBuffered;

        bool \_isBufferInitialized;

        public VirtualFile(Stream baseStream, string filename, int baseOffset, long fileSize, bool isBuffered = false) {

            Size = fileSize;

            BaseOffset = baseOffset;

            BaseStream = baseStream;

            \_isBuffered = isBuffered;

            FileName = filename;

        }

        public VirtualFile(Stream baseStream, string filename = "", bool isBuffered = false) {//重载

            BaseStream = baseStream;

            BaseOffset = 0;;

            Size = baseStream.Length;

            \_isBuffered = isBuffered;

            FileName = filename;

        }

这个定义的类内含参数：offset，文件大小，指针位置，文件名称，是否缓存，是否初始化。

这个类的函数针对地图文件的特点，对读取、写入、查找操作进行了改写。

之后地图文件的创建都是基于这个虚拟文件类。

可以把它想象成顺丰收快递的小哥，一个快递进来以后，快递尺寸、快递大小、快递物品类型、是否经过安检、是否需要包装等信息都被录入系统，便于分类和调用。

### 3.2.2 VirtualTextFile.cs

然后是VirtualTextFile.cs

它不是很重要，只在读取文件的时候用了一下。它的主要作用是定义了如何读取地图文件。

        public virtual string ReadLine() {

            // works for ascii only!

            var builder = new StringBuilder(80);

            while (CanRead) {

                char c = (char)ReadByte();

                if (c == '\n')

                    break;

                else if (c != '\r')

                    builder.Append(c);

            }

            return builder.ToString();

        }

### 3.2.2 MemoryFile

然后是基于VirtualFile类的MemoryFile，姑且称它为稍微上层一点的类。

public class MemoryFile : VirtualFile {//定义见VirtualFile.cs,它的基类是Stream

        public MemoryFile(byte[] buffer, bool isBuffered = true) :

            base(new MemoryStream(buffer), "MemoryFile", 0, buffer.Length, isBuffered) { }// //TODO Calling the base class method?

            //MemoryStream:Creates a stream whose backing store is memory.

            //base是什么函数

由MemoryFile的定义可知，在读取过程中是将地图的地形信息读取到了内存(或者说是缓存)中。它是调用了VirtualFile的第二种定义方法，也就是只输入缓存区数据和是否缓存（默认Ture）。这个类的作用是使得在读取地图时可以只输入至少一个参数——也就是地图文件的Stream——就能将地图读入内存（Meomory）。

举例而言，在MapFile.cs文件中，地形数据是这样读入的。

var mf = new MemoryFile(isoMapPack);//TODO mf是mapfile的意思吗?

## 3.3 数据格式

### 3.3.1 MapObject.cs

#### 3.3.1.1 综述

MapObject声明了地图的所有物体信息。

public class MapObject {

        public IsoTile Tile;

    }

    public class NamedMapObject : MapObject {

        public string Name { get; set; }

    }

    public class NumberedMapObject : MapObject {

        public virtual int Number { get; set; }

    }

MapObject将物体分成了两类。

一类是NamedMapObject，下级分类有Aircraft、Infantry、Smudge、Structure、Unit；这类的属性比较简单，默认只有名字一个属性。

一类是NumberedMapObject，下级分类有IsoTile、Overlay、Waypoint。这个类的属性默认有名字、字符串两个属性。

我们注意到NamedMapObject的自身属性是不涉及坐标的，

例如：

    public class Structure : NamedMapObject {

        public Structure(string owner, string name, short health, short direction) {

            Owner = owner;//所属方，请注意这里要输入的是string而不是int，因此可能不是对应所属方的硬编码

            Name = name;//名称

            Health = health;//血量

            Direction = direction;//建筑方向？

        }

因此推测读取地图信息时是不会直接把坐标存到这类物体内部的，那是怎么存呢？推测是在坐标上声明存在这样一个物体。

而NumberedMapObject都需要坐标，或者所谓键值，例如

public class Overlay : NumberedMapObject {

        public byte OverlayID { get; set; }

        public byte OverlayValue { get; set; }

        public Overlay(byte overlayID, byte overlayValue) {

            OverlayID = overlayID;

            OverlayValue = overlayValue;

        }

        public override int Number {

            get { return OverlayID; }

            set { OverlayID = (byte)value; }

        }

    }

就定义了ID和键值。

#### 3.3.1.2 IsoTile

IsoTile有七个参数，其中ushort是两个字节、byte是一个字节。

public class IsoTile : NumberedMapObject {

//TODO 前四个参数意义我不是很清楚

        public ushort Dx;//ushort Unsigned 16-bit integer 2-bytes

public ushort Dy;

        public ushort Rx;

        public ushort Ry;

        public byte Z;//1 bytes 高度？

        public short TileNum;//16-bit TileNum，也就是声明了是哪个块

        public byte SubTile;//1 bytes 子块

//这里声明了定义方法

        public IsoTile(ushort p1, ushort p2, ushort rx, ushort ry, byte z, short tilenum, byte subtile) {

            Dx = p1;

            Dy = p2;

            Rx = rx;

            Ry = ry;

            Z = z;

            TileNum = tilenum;

            SubTile = subtile;

        }

这里定义了一个函数。

        public List<byte> ToMapPack5Entry() {

            var ret = new List<byte>();

            ret.AddRange(BitConverter.GetBytes(Rx));//2 bytes

            ret.AddRange(BitConverter.GetBytes(Ry));//2 bytes

            ret.AddRange(BitConverter.GetBytes(TileNum));//2 bytes

            ret.Add(0); ret.Add(0);//1+1 bytes

            ret.Add(SubTile);//1 bytes

            ret.Add(Z);//1 bytes

            ret.Add(0);//1 bytes

            return ret;

        }

    }

注意，ToMapPack5Entry是编码时一个重要函数。可以看到，它将11个字节的信息按顺序组成了一个字符串，分别是：Rx+Ry+TileNum+0+0+SubTile+Z+0，并存储在为元素为byte类的list：ret中。

请注意这里加了三个零，意义是什么呢？分隔符？

并且一个值得注意的事情是，dx和dy是没有被存进去的。

#### 3.3.1.3 Overlay

定义了覆盖物的ID和键值。我猜是对应了是什么，和 是哪个？//TODO

public class Overlay : NumberedMapObject {

        public byte OverlayID { get; set; }

        public byte OverlayValue { get; set; }

        public Overlay(byte overlayID, byte overlayValue) {

            OverlayID = overlayID;

            OverlayValue = overlayValue;

        }

        public override int Number {

            get { return OverlayID; }

            set { OverlayID = (byte)value; }

        }

    }

### 3.3.2 TileLayer.cs

定义了一个能容纳全图Tile的类。就像一个坐标纸。

public class TileLayer : IEnumerable<IsoTile> {

//IEnumerable<T>是一个保证给定类是可迭代的接口.表示实现的类IEnumerable<T>可以被认为并用作一系列元素。

//T表示序列中元素的数据类型

IsoTile[,] isoTiles;//TODO 这是什么

        private Size fullSize;//Size是指定一个矩阵 Size(Int32, Int32)指定一个m\*n的矩阵

        public TileLayer(int w, int h)

            : this(new Size(w, h)) {//创建一个新的空白TileLay方法

        }

        public TileLayer(Size fullSize) {

            this.fullSize = fullSize;

            isoTiles = new IsoTile[fullSize.Width \* 2 - 1, fullSize.Height];//TODO

        }

        public int Width {

            get { return fullSize.Width; }

        }

        public int Height {

            get { return fullSize.Height; }

        }

        public virtual IsoTile this[int x, int y] {

            get {

                if (0 <= x && x < isoTiles.GetLength(0) && 0 <= y && y < isoTiles.GetLength(1))

                    return isoTiles[x, y];

                else

                    return null;

            }

            set {

                isoTiles[x, y] = value;

            }

        }

在TileLayer里，写明了如何把数据压缩并存储。

public void SerializeIsoMapPack5(IniFile.IniSection isoMapPack5) {

            int cells = (Width \* 2 - 1) \* Height;

            int lzoPackSize = cells \* 11 + 4; // last 4 bytes contains a lzo pack header saying no more data is left

            var isoMapPack = new byte[lzoPackSize];

            var isoMapPack2 = new byte[lzoPackSize];//TODO 这是什么

            long di = 0;

            foreach (var tile in this.isoTiles) {

                var bs = tile.ToMapPack5Entry().ToArray();//ToMapPack5Entry的定义在MapObjects.cs

//ToArray将ArrayList转换为Array：

Array.Copy(bs, 0, isoMapPack, di, 11);//把bs复制给isoMapPack,从di索引开始复制11个字节

                di += 11;//一次循环复制11个字节

            }

            var compressed = Format5.Encode(isoMapPack, 5);

            string compressed64 = Convert.ToBase64String(compressed);

            int i = 1;

            int idx = 0;

            isoMapPack5.Clear();

            while (idx < compressed64.Length) {

                int adv = Math.Min(74, compressed64.Length - idx);//74是什么

                isoMapPack5.SetValue(i++.ToString(), compressed64.Substring(idx, adv));//start length

                idx += adv;//idx=adv+1

            }

        }

### 3.3.3 MapFile.cs

#### 3.3.3.1 定义

读取文件使用了MapFile.cs，它定义了地图文件类，记录了地图尺寸、瓷砖信息、覆盖物、脏污、Terrain、建筑、兵种、单位、飞行单位、路径点、内置ini这些地图的属性。

可以看到MapFile类是基于IniFile类定义的。

public class MapFile : IniFile {

        private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

        public Rectangle FullSize { get; private set; }//TODO Rectangle(Point, Size) 是一个矩阵/Rectangle(Int32, Int32, Int32, Int32)

public Rectangle LocalSize { get; private set; }

        public TileLayer Tiles;//TileLayer的类定义在TileLayer.cs里

        public readonly List<Overlay> Overlays = new List<Overlay>();//list<>表示可通过索引访问

        public readonly List<Smudge> Smudges = new List<Smudge>();

        public readonly List<Terrain> Terrains = new List<Terrain>();

        public readonly List<Structure> Structures = new List<Structure>();

        public readonly List<Infantry> Infantries = new List<Infantry>();

        public readonly List<Unit> Units = new List<Unit>();

        public readonly List<Aircraft> Aircrafts = new List<Aircraft>();

        public readonly List<Waypoint> Waypoints = new List<Waypoint>();

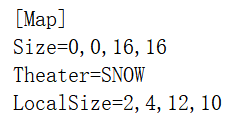
        public readonly List<IniSection> MiscSections = new List<IniSection>();

        public Lighting Lighting;

#### 3.3.3.2 初始化地图的函数

MapFile.cs声明了如何初始化一张地图。可以看出读到了是把地图读到了信息流中，再存到了缓存区，并用读取到的文件包含的地图信息给地图类的各个属性赋值。

说明一下：GetSection在别的文件中有定义，总之它能返回输入的标签所对应的地图内容。

解释一下流程：  
1.在读取的地图文件中获得标签为"Map"的内容，并赋值给MapFile类的属性。可以用写字板打开地图文件，可以查找到这个。  
  
  
它声明了地图的尺寸、类型、能显示的尺寸。  
2.根据地图尺寸生成一个全新的TileLayer阵列，命名为Tiles。（从感性的角度看，这就像是制造了一个专门的容器，为之后读取地图的地形并储存做准备。）  
3.调用读取其信息的函数，并写入log日志中。  
4.地图灯光。  
  
  
至此地图读取就结束了。

public void Initialize() {//读入地图文件中的[Map]

            var map = GetSection("Map");//这里的"Map"指的是地图文件里的了

            string[] size = map.ReadString("Size").Split(',');//size是地图文件里的那行，有四个参数

            FullSize = new Rectangle(int.Parse(size[0]), int.Parse(size[1]), int.Parse(size[2]), int.Parse(size[3]));

//rectangle 生成一个矩阵 FullSize是一个矩阵

            Tiles = new TileLayer(FullSize.Width, FullSize.Height);//TODO fullsize的定义不知道在哪儿

            size = map.ReadString("LocalSize").Split(',');//指的是可显示的那个localsize

            LocalSize = new Rectangle(int.Parse(size[0]), int.Parse(size[1]), int.Parse(size[2]), int.Parse(size[3]));//parse 转换为数字

            Logger.Info("Reading map");//write log.Info Debug 是日志输出的不同等级

            Logger.Debug("Reading tiles");

            ReadTiles();//TODO 定义在下面

            Logger.Debug("Reading map overlay");

            ReadOverlay();//TODO 定义在下面

            Logger.Debug("Reading map terrain objects");

            ReadTerrain();//TODO 定义在下面

            Logger.Debug("Reading map smudge objects");

            ReadSmudges();//TODO 定义在下面

            Logger.Debug("Reading infantry on map");

            ReadInfantry();//TODO where is declaration?

            Logger.Debug("Reading vehicles on map");

            ReadUnits();//TODO where is declaration?

            Logger.Debug("Reading aircraft on map");

            ReadAircraft();//TODO where is declaration?

            Logger.Debug("Reading map structures");

            ReadStructures();//TODO where is declaration?

            Logger.Debug("Waypoints");

            ReadWaypoints();//TODO where is declaration?

            Lighting = new Lighting(GetOrCreateSection("Lighting"));//TODO这是地图光照吗？

        }

上面的流程实现后，一张地图就非常清晰地存在了电脑内存中，很方便地可以调用。  
虽然上面的流程是如此直观，但不弄清读取和存储的具体细节，就相当于没用的伪代码。我只关心地形Tiles，也就是对应ReadTiles();  
接下来我就详细解释一下它的实现。

#### 3.3.3.2 ReadTile

直接上代码流程：

1.读取地图文件中[IsoMapPack5]的内容，赋值给mapSection这个变量.  
p.s.GetSection函数也是定义的。之后会说怎么实现的，在本程序中只需要知道如何使用即可。  
2.将mapSection合并成一整个字符串、转码，并赋值给定义一个byte []类的lzoData，用于之后的解压缩。  
3.读取地图大小，乘起来得到一共有多少块Tile。  
4.lzoPackSize=11\*cells+4;前文说过，每个Tile是11个字节，至于+4，注释里说了是一个休止符号。  
lzoPackSize也是用于解压缩的参数之一。lzoPackSize就等于解压前的数据大小，因为解压前数据大小就应该等于  
11\*Tile数+4；  
5.重点来了，解压缩，使用Format5的压缩方法，其实也就是miniLZO压缩方法。  
请注意，这里大大简化了miniLZO的使用，因为miniLZO解压缩是需要四个参数的，而这里通过又写了一层外壳函数，将参数缩减到了两个，也就是：待解压数据lzoData和与解压前的大小相同的byte[]类isoMapPack。  
再解释一下，DecodeInto这个函数的第二个参数isoMapPack在运行结束后会被赋予解压后的数据。

private void ReadTiles() {//Tile是地形块的意思 “瓷砖”

            var mapSection = GetSection("IsoMapPack5");//指的是地图文件中的"[IsoMapPack5]"

            byte[] lzoData = Convert.FromBase64String(mapSection.ConcatenatedValues());

//concatenated的意思是连接起来，应该是把mapSection连接成一个长的整体再一起转码

//所以IzoData是一个64位连续的串

            int cells = (FullSize.Width \* 2 - 1) \* FullSize.Height;

            //每个cell占 11 bytes，最后四个btyes是终止符

            int lzoPackSize = cells \* 11 + 4; // last 4 bytes contains a lzo pack header saying no more data is left

            var isoMapPack = new byte[lzoPackSize];//lzoPackSize长度的byte，可通过index访问

            //var 格式受读入定义.isoMapPack是一串bytes

            uint totalDecompressSize = Format5.DecodeInto(lzoData, isoMapPack);//TODO 源，目标 输入应该是解码后长度，isoMapPack被赋值解码值了

            //uint  0 to 4,294,967,295  Unsigned 32-bit integer System.UInt32

就这样，解压结束后，我们得到了一个总的解压后大小：totalDecompressSize，  
和已经储存了所有Tile信息的byte[]类isoMapPack。  
但是我们如何从isoMapPack这个一整个数组中还原地图的每个Tile呢？

因为我们已经知道了压缩文件的结构，也就是每11个字节就是一个Tile，并且顺序分别为rx,ry,tilenom,zero1,subtile,z,zero2。所以我们只要11个11个读就可以。

读取以后，计算出dx和dy（这里我没弄懂rx ry dx dy都是干什么的）//TODO  
然后根据dx和dy作为行和列，赋值给之前生成的空白阵列Tiles。  
  
就这样，地形的信息就完全存在了电脑内存里。

var mf = new MemoryFile(isoMapPack);//TODO mf是mapfile的意思吗?

            //MemoryFile的定义见MemoryFile.cs

            int numtiles = 0;

            for (int i = 0; i < cells; i++) {

                //TODO 这些值是什么。

                ushort rx = mf.ReadUInt16();//ushort    0 to 65,535 Unsigned 16-bit integer System.UInt16

                ushort ry = mf.ReadUInt16();

                short tilenum = mf.ReadInt16();//short  -32,768 to 32,767   Signed 16-bit integer   System.Int16

                short zero1 = mf.ReadInt16();//Reads a 2-byte signed integer from the current stream and advances the current position of the stream by two bytes.

                byte subtile = mf.ReadByte();//Reads a byte from the stream and advances the position within the stream by one byte, or returns -1 if at the end of the stream.

                byte z = mf.ReadByte();

                byte zero2 = mf.ReadByte();

                //一次循环读11 bytes

                int dx = rx - ry + FullSize.Width - 1;

                int dy = rx + ry - FullSize.Width - 1;

                //上面是一个线性变换 旋转45度、拉长、平移

                numtiles++;//在最后日志用了一下

                if (dx >= 0 && dx < 2 \* Tiles.Width &&

                    dy >= 0 && dy < 2 \* Tiles.Height) {

                    var tile = new IsoTile((ushort)dx, (ushort)dy, rx, ry, z, tilenum, subtile);//IsoTile定义是NumberedMapObject

                    Tiles[(ushort)dx, (ushort)dy / 2] = tile;//给瓷砖赋值

                }

            }

            // fix missing tiles

            // import tiles

            for (ushort y = 0; y < FullSize.Height; y++) {

                for (ushort x = 0; x <= FullSize.Width \* 2 - 2; x++) {

                    var isoTile = Tiles[x, y];//从这儿来看，isoTile指的是一块瓷砖，Tile是一个二维数组，存着所有瓷砖

                    //isoTile的定义在TileLayer.cs里

                    if (isoTile == null) {

                        // fix null tiles to blank

                        ushort dx = (ushort)(x);

                        ushort dy = (ushort)(y \* 2 + x % 2);

                        ushort rx = (ushort)((dx + dy) / 2 + 1);

                        ushort ry = (ushort)(dy - rx + FullSize.Width + 1);

                        Tiles[x, y] = new IsoTile(dx, dy, rx, ry, 0, 0, 0);//TODO IsoTile有七个参数，定义在112行

                    }

                }

            }

# 四、编码方式

就简单说几句，miniLZO是用来解压缩IsoMapPack5的。Format80是用来解压缩OverLay的。

Format80实际上就是LCW（<http://www.shikadi.net/moddingwiki/Westwood_LCW>）

The LCW compression algorithm was Westwood Studios' response to Unisys' enforcement of their patent on the LZW algorithm. Before its real name was known to the fanbase, the compression format was generally referred to as "Format 80", because it is indicated with byte value 0x80 in the frame headers of [Command & Conquer 1 SHP files](http://www.shikadi.net/moddingwiki/Westwood_SHP_Format_(TD)), and because all compressed content ends on byte 0x80.

The internally used name, LCW, stands for Lempel–Castle–Welch, a variation on [Lempel–Ziv–Welch](http://www.shikadi.net/moddingwiki/LZW_Compression), with "Castle" referring to Louis Castle, who wrote the algorithm. It is unknown why this name was chosen, though, since neither the algorithm nor its output resemble LZW in any way. In fact, its final compressed data is much more similar to [Code-based RLE compression](http://www.shikadi.net/moddingwiki/RLE_Compression#Code), but with a much-extended set of commands. It is entirely possible the only reason for the name is that the algorithm was their replacement for LZW.