# 游戏引擎架构

## 第3章 游戏软件工程基础

本章简要介绍了面向对象编程的基础概念。

### 3.1 重温C++及最佳实践

继承：避免菱形继承是许多高手的习惯。

多态：多态是一种语言特征，容许采用单一共同接口操作一组不同类型的对象。共同接口能使异质对象集合从使用接口的代码来看显得是同质的。

代码示例如下。

```
struct Shape
{
  virtual void Draw() = 0;
};

struct Circle : public Shape
{
  virtual void Draw()
  {
    // do someting
  }
};

struct Rectangle : public Shape
{
  virtual void Draw()
  {
    // do someting
  }
};

void drawShapes(std::list<Shape*> shapes)
{
  std::list<Shape*>::iterator shapeItr = shapes.begin();
```

```
    std::list<Shape*>::iterator shapeEnd = shapes.end();

    for(; shapeItr != shapeEnd; ++shapeItr)
    {
      (*shapeItr)->Draw();
    }
  }
```

合成：合成是指用一组互动的对象去完成高阶任务。理解上类似于组件模式。

#### 3.1.1.6 设计模式

当同一类型的问题反复出现，而不同程序员们却采用类似的方案去解决这些问题，就可以说，该问题引发了一个**设计模式（Design Pattern）**。常用有如下几个。

- 单例：确保某个类只有一个实例，并提供这个单例的全局存取方法。
- 迭代器：迭代器提供高效存取一个集合的方法，同时不需要暴露该集合之下的实现。
- 抽象工厂：抽象工厂提供一个接口，创造一组相关或互相依赖的类，而不需要指明那些类的具体类。

### 3.1.2 编码标准：为什么及需要多少

笔者认为，编码约定中最需要达到的事情如下。

- **接口为王**：保持接口(.h)整洁、简单、易于理解，并且有良好注释。
- **好名字促进理解及避免混淆**：花时间找适合的名字。
- **不要给命名空间添乱**：使用C++命名空间或统一的名字前缀，以确保自己的符号不会与其他库的符号冲突。小心文本替换的宏，它会跨越全部C/C++作用域及命名空间范围。
- **遵从最好的C++实践**：《Effective C++》、《Effective STL》等。
- **始终如一**
- **显露错误**：http://www.joelonsoftware.com/articles/Wrong.html

## 3.2 C/C++的数据、代码及内存

#### 3.2.1.1 数值底数

人类是用底数10的方式对待数字，而计算机使用二进制存储。十六进制是计算机领域另一种常见的记法，因为计算机分组存储数据，每8位一个字节，一个十六进制正好4位，所以两个十六进制数字恰好能代表一个字节。

#### 3.2.1.4 浮点数

最流行的标准是IEEE-754，最高位是符号位，紧随的8位是指数和23位尾数。

#### 3.2.1.5 基本数据类型

编译器不同，数据类型的位数不同，原则是尽可能使目标硬件达到最大效能。

- char - 通常是8位，足矣存储ASCII和UTF-8字符，分有无符号。
- int、short、long - int一般为计算机最高效的运算单位(例如你的windows是64位，则是64位)，short比int低，一般为16位。long大于int，一般为64位。
- float - 大部分编译器是32位。
- double - 64位。
- bool - 不会定义为1位，一般是8位或者32位。

SIMD是从硬件角度用于加速计算，名为single instruction，multiple data。游戏方面常用把4个32位浮点数存进128位SIMD寄存器，如使矢量点积和矩阵乘数更高效。

不同的处理器SIMD的叫法不同。

为了移植性，多数游戏引擎会定义自定的数据类型。

#### 3.2.1.6 多字节值和字节序

~~没看懂，后期补~~

### 3.2.2 声明、定义及链接规范

#### 3.2.2.1 在谈翻译单元

编译器操作的最小翻译单位是cpp文件，生成个别的对象文件(.o或.obj)，内包含编译后的机器码和全局变量与静态变量。其中有一个特殊的是包含未解决引用，这是其它cpp文件定义的函数和全局变量。

由于编译器每次操作只针对一个单元，遇到为解决引用的外部变量和函数，只能相信它们存在。链接器的主要作用是解决外部引用，因此也只能报告以下两种错误。

1. 找不到extern引用的目标，报"无法解决的外部符号(unresolved external symbol)"。
2. 若找到的大于两个，报"符号被多重定义(multifly defined symbol)"。

#### 3.2.2.2 声明和定义

变量声明和定义之后才可以使用。

- 声明 - 是数据对象或函数的描述。其实就是告诉编译器方法或变量的名字和数据类型。
- 定义 - 是程序中个别内存区域的描述。可以用来放变量、struct、class的实例，以及函数的机器码。

在某翻译单元中可使用extern声明，以供其他翻译单元使用，如下。

```
extern float score;
```

内联函数是调用该函数时复制机器码到调用处，需注意内联函数的定义需在头文件中，并且由编译器做

是否内联的决定。某些编译器提供了如__forceinline，绕过编译器成本效益分析，直接内联。

#### 3.2.2.3 链接规范

- 外部链接 - 可被定义处意外的翻译单元看见并引用。
- 内部链接 - 只能被该定义所处的翻译单元看见并引用。

有几点需要注意：所有定义预设均为外部链接，通过static关键字可以改为内部链接。

### 3.2.3 内存布局

当生成c/c++程序时，链接器创建可执行文件(例如windows下的exe文件)。可执行文件总是包含程序的部分映像，程序执行时此部分映像会在内存中。成为"部分"映像是因为除了把可执行映像至于置于内存中，一般也会分配额外内存。

映像文件一般最少由以下4部分组成。

- **代码段** - 此段包含程序中定义的全部函数的可执行机器码。
- **数据段** - **获初始化**的全局和静态变量。
- **BSS段** - 包含**未初始化**的全局变量和静态变量。
- **只读数据段** - 包含程序中定义的只读全局变量。

#### 3.2.3.2 程序堆栈

当可执行程序被载入内存时，操作系统会保留一块称为程序堆栈的内存。此内存块被称为堆栈帧，其一般包括以下几个部分。

- 调用函数的返回地址。
- 相关CPU寄存器的内容。
- 函数内的局部变量。(该地址在函数执行完之后就可能被别人的调用函数所覆盖)

#### 3.2.3.3 动态分配的堆

全局变量和静态变量处于可执行镜像中，局部变量处于堆栈中，动态分配的内存处于堆中，此内存块称为**堆内存**或**自由存储**。

### 3.2.4 成员变量

c struct和c++ class都可用来把变量组成逻辑单元。class或struct的声明不占用内存。

### 3.2.5 对象的内存布局

#### 3.2.5.1 内存对齐

```
struct InefficientPacking
{
    U32   mU1; // 32位
    F32   mF2; // 32位
    U8    mB3; // 8位
    I32   mI4; // 32位
    bool  mB5; // 8位
    char* mP6; // 32位
};
```

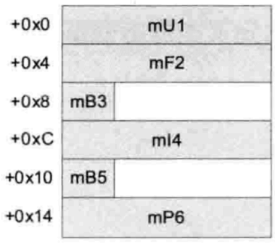


图 3.13: 混合数据成员大小导致低效的struct包裹。

如上图所示，为了供CPU高效的从内存读/写，在存储时编译器会产生间隔的情况。数据对象的对齐是指，其内存地址是否为对齐字节大小的倍数。

- 1字节对齐的对象，可置于任何地址。
- 2字节对齐的对象，只可置于偶数地址（地址有效半字节为0x2，0x4，0x6，0x8，0xA，0xC，0xE）。
- 4字节对齐的对象，只可置于为4倍数的地址(地址有效半字节为0x0，0x4等)。
- 16字节对齐的对象，只可置于为16倍数的地址（地址有效半字节为0x0）。

```
struct MoreEfficientPacking
{
    U32   mU1; // 32位(4字节对齐)
    F32   mF2; // 32位(4字节对齐)
    I32   mI4; // 32位(4字节对齐)
    char* mP6; // 32位(4字节对齐)
    U8    mB3; //  8位(1字节对齐)
    bool  mB5; //  8位(1字节对齐)
};
```

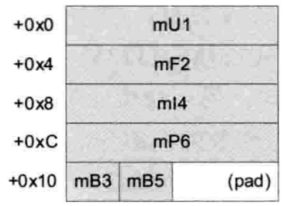


图 3.15: 小成员组合在一起，包裹更高效。

上图就是内存对齐的正确姿势。

#### 3.2.5.2 C++中类的布局

在内存布局中，C++的类有别于C的结构之处有二——继承和虚函数。

当B类继承于A类，内存中B的数据成员会紧跟A类数据成员之后。当类含有或继承了一个或多个虚函数，那么会在类的布局中4字节(视目标平台指针所占字节数目)，通常在类布局的最前端，名为虚表指针，指向虚函数表。

### 3.3 捕捉及处理错误

总结就是**让错误变得明显**，并且尽量不要阻断别的成员的工作。使用断言。

## 第5章 游戏支持系统

每个引擎都会包括一些底层支持系统。例如启动和终止引擎、文件管理系统、渲染系统等。

由于一些系统包含依赖关系，所以需要谨慎处理管理类的创建和销毁，以下c++式构建对象不能保证构建和析构顺序，所以pass。

```
class RenderManager
{
  public RenderManager()
  {
   //启动管理器
  }
  ~RenderManager()
  {
   //终止管理器
  }
};
static RenderManager gRenderManager;
```

#### 5.1.1.1 按虚建构

C++有一条规定：函数内声明的静态变量并不会于main()之前构建，而是在第一次调用时构建。

```
class RenderManager
{
  public:
    static RenderManager& get()
    {
      static RenderManager sSingleton;
      return sSingleton;
    }
  RenderManager()
  {
    // 需依赖的话，可以先构建
    VideoManeger::get();
    TextureManager::get();
  }
};
```

此方法不能按需析构，所以pass。

### 5.1.2 行之有效的简单方法

让构建和析构不做任何事情，而是提供启动和终止函数，用于在main()中明确次序的调用。

## 5.2 内存管理

- **动态内存分配** - malloc()或者c++的new操作是非常慢的操作。要提升效能，就要尽量避免动态分配。

- **内存访问模式** - 大粒度的数据不连续，也会拖垮运行速度，即要缓存友好。

### 5.2.1 优化动态内存分配

堆分配通常是很快的，原因如下。

- **堆分配器(heap allocator)**是通用的设施，它必须处理任何大小的需求，从1字节到1000兆字节，这需要大量的管理开销，导致malloc()和free()变得缓慢。
- 需要从用户模式切换到内核模式处理请求，然后再切换到原来的程序，这种上下文切换需要耗费非常多的时间。

原则是：**维持最低限度的堆分配，并且永不在紧凑循环中使用对分配**。

所以游戏引擎一般都会实现一个或多个**定制分配器（custom allocator）**。定制器比操作系统分配器更优的原因如下。

- 从已分配的内存中(来自于new和malloc)完成分配请求，这样就避免了上下文切换。
- 通过对定制分配器的使用模式做出多个假设。

#### 5.2.1.1 基于堆栈的分配器

堆栈分配器最适合的场景是游戏中加载场景，切换场景时释放该场景。堆栈分配器不能以任意顺序释放内存，必须以分配时相反的顺序释放。伪代码如下。

```
class StackAllocator
{
  public:
    // 表示堆栈的当前顶端
    typedef U32 Marker;
    // 给定总大小
    explicit StackAllocator(U32 stackSize_bytes);
    //给定内存块大小
    void* alloc(U32 size_bytes);

    Marker getMarker();

    void freeToMarker(Marker marker);

    // 清空堆栈
    void clear();
};
```

#### 5.2.1.1 双端堆栈分配器

从两个方向同时开始分配，只要保证所用之和不会大于分配即可。例如可以一个方向用于关卡管理，另一个用于细碎的临时内存块，这种方案保证了不会产生**内存碎片**问题。

#### 5.2.1.2 池分配器

如果需要分配大量同等尺寸的内存块，那就是池分配器了。

#### 5.2.1.3 含对齐功能的分配器

每个变量和数据对象都有对齐要求，8位整数可对齐至任何地址，但32位整数或浮点变量则4字节对齐。所有内存分配器都必须能传回对齐的内存块。实现的方式是比请求多分配一点内存，在向上调整地址至适当的对齐，最后传回调整后的地址。

额外分配的字节等于对齐字节，例如对齐字节是16字节，则多分配16字节。一个计算方式如下。

```
//alignment应为2的幂(一般是4或16)
void* allocateAligned(U32 size_bytes, U32 alignment)
{
  //应该分配的总量
  U32 expandedSize = size_bytes + alignment;
  //分配内存
  U32 rawAddress = (U32)allocateUnsigned(expandedSize);
  //使用掩码去除地址低位部分，计算"错位"量，从而计算调整量
  U32 mask = alignment - 1;
  U32 misalignment = (rawAddress & mask);
  U32 ajustment = alignment - misalignment;

  U32 alignedAddress = rawAddress + adjustment;
  return (void*)alignedAddress;
}
```

当需要释放时，需要找到未对齐之前的地址，要完成这个，可以储存一些元信息至额外分配的内存，这些内存原本只是用于内存对齐。有偏移量即可知道以前的地址，即知道ajustment即可。代码如下。

```
{
  //上一代码块放这里

  U32* pAdjustment = (U32*)(alignedAddress - 4);
  *pjustment = adjustment;
}
```

而对应的freeAligned()函数可实现如下。

```
void freeAligned(void* p)
{
  U32 alignedAddress = (U32)p;
  U8* pAdjustment = (U8*)(alighedAddress - 4);
  U32 adjustment = (U32)*pAdjustment;
  U32 rawAddress = alighedAddress - adjustment;
  freeUnaligned((void*)rawAddress);
}
```

#### 5.2.1.4 单帧和双缓冲内存分配器

单帧分配器需要先预留一块内存，然后使用上面介绍的堆栈分配器进行分配，需要注意的是数据只在当前帧有效，下一帧就会被自动清除。效率及其高效，但是需要程序员去保证没有滥用，代码如下。

```
StackAllocator g_singleFrameAllocator;
while(true)
{
  g_singleFrameAllocator.clear();
  void* p = g_singleFrameAllocator.alloc(nBytes);
}
```

双缓冲分配器是指在第n帧分配的数据可以第n+1帧使用，实现思路是建立两个相同尺寸的单帧堆栈分配器，并在每帧交替使用。代码如下。

```
class DoubleBufferedAllocator
{
  U32 m_curStack;
  StackAllocator m_stack[2];

 public:
  void SwapBuffers()
  {
    m_curStack = (U32)!m_curStack;
  }

  void clearCurrentBuffer()
  {
    m_stack[m_curStack].clear();
  }

  void* alloc(U32 bytes)
  {
```

```
        return m_stack[m_curStack].alloc(bytes);
      }
    };

    DoubleBuffedAllocator g_doubleBufAllocator;

    while(true)
    {
      g_doubleBufAllocator.swapBuffers();
      g_doubleBufAllocator.clearCurrentBuffer();
      void* p = g_doubleBufAllocator.alloc(nBytes);
    }
```

### 5.2.2 内存碎片

动态堆分配不可避免的会产生内存碎片，内存碎片的危害是即使剩余内存中总量满足分配的需求，但是因为**内存块不连续**，所以还是会导致分配失败。

在支持**虚拟内存**的系统上，内存碎片不是大问题。多数嵌入式设备并不能负担的起虚拟内存的开销，所以多数游戏引擎并不会使用虚拟内存。

#### 5.2.2.1 以堆栈和池分配器避免内存碎片

- 堆栈分配器可以彻底避免内存碎片，不过不可以避免的问题是不能乱序释放，只能倒序。
- 池分配器也无内存碎片问题，因为是固定大小的内存块，所以释放了可以补充上后期的使用。

#### 5.2.2.2 内存整理及重定位

内存整理的一个思路是一个使用中的内存块前方如果有被释放的内存，则前移，思路如下图。

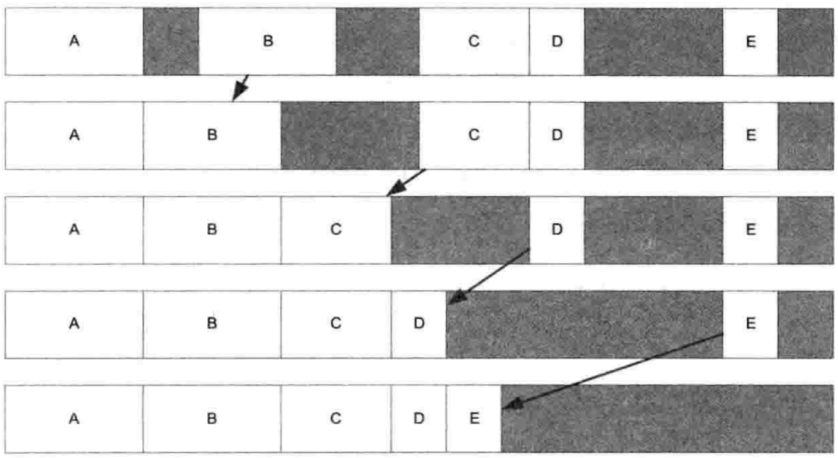


图 5.6: 通过搬移已分配的内存块来整理碎片。

如果有指针指向该内存块，需要重定向该指针到新的位置，但是当前c++的指针不能实现，取而代之的是使用智能指针或者句柄。

- 使用智能指针的话，可以将智能指针储存在一个全局链表里，当要移动某块内存，便可遍历该链表，指向新地址。
- 句柄通常实现为索引，索引指向句柄表中的元素，该元素是一个指针。当移动某块内存时，可以扫描句柄表。由于句柄索引不对，因此使用句柄的对象不受影响。

**某些内存块可能不能被重定位**。使用第三方库时，恰好该库不使用智能指针或句柄。两种思路，一种是置于可整理内存区外的内存中，或者接受。

碎片整理通常需要复制内存块，这通常很慢。所以通过分帧的操作降低影响。

### 5.2.3 缓存一致性

和CPU的速度相比，存取主系统内存是缓慢的操作，通常需要几千个处理器周期才能完成。然而CPU里的寄存器只需数十个甚至一个周期，所以现代处理器会采用高速的内存缓存。

当CPU首次读取某区域的主内存，会将内存小块载入高速缓存（大小为8至512字节，视平台架构而定，称之为缓存线）。后期要读取的数据在缓存中，就可以大大加块读取速度，如果不在，则被逼暂停，等待缓存线自主内存更新后继续运行，这称之为**缓存命中失败（cache miss）**。

#### 5.2.3.1 一级及二级缓存

主内存比L2缓存（通常在主板）慢，L2缓存比L1缓存（通常在CPU上）慢，因此L2缓存命中失败通常比L1缓存命中失败的成本高。

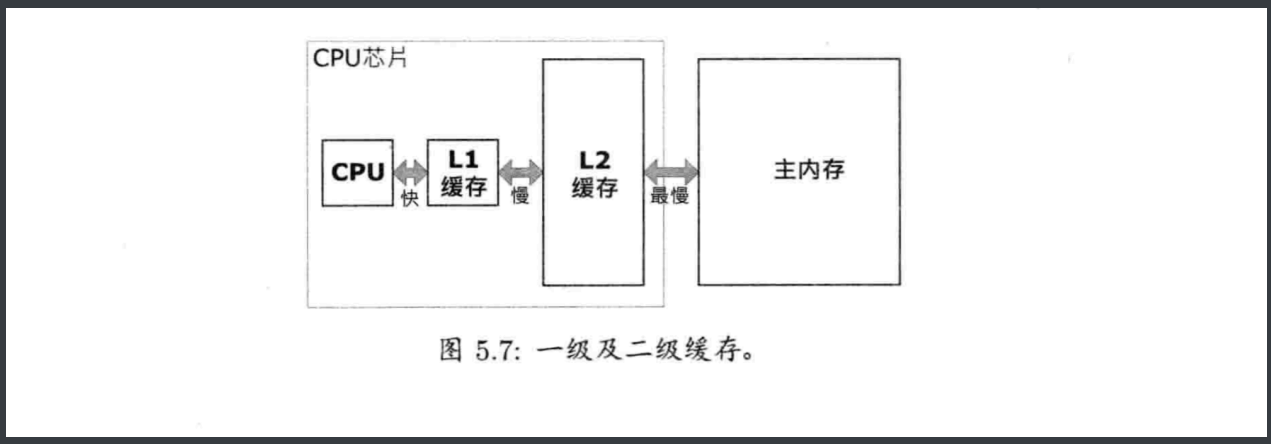

图 5.7: 一级及二级缓存。

#### 5.2.3.3 避免缓存命中失败

避免**数据缓存**命中失败通常是数据连续（并且越小越好），访问连续的时候，缓存命中最佳。

由于编译器和链接器决定了代码的内存布局，所以读者觉得无法控制**指令缓存**命中失败，但是知晓以下几个方面，便能控制代码的内存布局。

- 单个函数的机器码通常置于连续的内存。
- 编译器和连接器按照函数在翻译单元出现的次数排列内存布局。

因此，避免指令缓存命中失败有以下经验之谈。

- 高效能代码体积越小越好。
- 在性能关键的代码段落中，避免调用函数。
- 如要调用函数，被调用函数最好在该函数前面。

## 5.3 容器

常见的数据结构或者说容器包括但不限于以下几个。

- 数组(array) - 内存连续，长度通常是编译期静态可定。
- 动态数组(dynamic array) - 可在运行期动态改变长度的数组，如std::vector。
- 链表(linked list) - 有序集合但内存不连续，如std::list。
- 堆栈(stack) - 后进先出。
- 队列(queue) - 先进先出。
- 双端队列(double-ended queue, deque) - 可以在两端高效的插入及删除数据，如std::deque。
- 优先队列(priority queue) - 高效的弹出优先级最高的元素，通常使用二叉树实现，如std::priority_queue。
- 树(tree) - 以层阶结构组织元素，有0或者1个父节点，有0或者多个子节点。
- 二叉搜索树(binary search tree，BST) - 每个节点最多含两个子节点，有多种类型，例如红黑树(red-black tree)、伸展树(splay tree)、AVL树(AVL tree)。
- 二叉堆(binary heap) - 通常采用完全二叉树的数据结构，通常使用数组储存，根节点必定最大或者最小。
- 字典(dictionary) - 由键值对组成的表。字典又称之为映射(map)或者散列表(hash table)，技术上说，散列表只是字典的其中的一种实现方式。如std::map、std::hash_map。
- 集合(set) - 里边的数据不会重复。
- 图(graph) - 节点的集合，节点之间有单双向的可能。
- 有向非循环图(directed acyclic graph，DAG) - 图的特例，单向且无循环。

### 5.3.1 容器操作

- 插入
- 移除
- 顺序访问/迭代
- 随机访问
- 查找
- 排序

### 5.3.2 迭代器

迭代器像是数组索引或指针——每次他都会指向容器中某个元素，可以移至下一个元素，并用某种方式表示是否遍历结束。C++风格迭代器如下。

```
void processList(std::list<int>& container)
{
  std::list<int>::iterator pBegin = container.begin();
  std::list<int>::iterator pEnd = container.end();
  std::list<int>::iterator p;
  for(p = pBegin; p != pEnd; ++p)
  {
    int element = *p;
  }
}
```

迭代器的好处如下。

- 直接访问破坏容器类的封装。迭代器通常为容器类的友元。
- 迭代器简化了迭代过程。

### 5.3.4 建立自定义容器

原因如下。

- **完全掌控**。内存需求、算法、何时分配释放等。
- **优化的机会**。某些平台可能有某些硬件功能，可借这些功能优化数据结构和算法。
- **可定制性**。可自行添加想要的功能。
- **消除外部依赖**。如使用第三方库，若库有问题就gg了。

#### 5.3.4.1 建还是不要建

STL（standard template library）的优点如下。

- 提供了丰富的功能。
- 在许多平台都有不错的实现。
- 几乎所有C++编译器都带有STL。

也有许多缺点，如下。

- 陡峭的学习曲线，主要是头文件晦涩难懂。
- 相比为某问题打造的数据结构，STL会比较慢。
- 相比自行设计，占得内存较多。
- STL会进行许多动态内存分配。
- STL的实现和行为在各编译器上有差异，增加了多平台引擎应用STL的难度。

如果多平台的话，笔者建议使用 STLPort。

Boost的目标是扩展STL并与STL联合工作，以下是一些简要。

- 提供了许多STL没有的功能。
- 能有效处理一些负责问题，如智能指针。
- 文档较好，并会写开发该库的设计决定、约束和需求，因此阅读该文档也是学习软件设计原则的好办法。
- 有一些库生成颇大的lib库。
- 不保证向后兼容。

Loki是最知名且可能是最强大的C++ TMP（template metaprograming，模板元编程）库。它的核心是利用编译器做一些通常在运行期才做的事情。

#### 5.3.4.3 链表

如果想要高效插入及移除元素，则链表是不二之选。链表中的储存的元素称之为节点，节点数据结构如下。

```
template<typename ELEMENT>
struct Link
{
  Link<ELEMENT>* m_pPrev;
  Link<ELEMENT>* m_pNext;
  ELEMENT* m_pElem;
};
```

**外露式表(extrusive list)**是一种链表，其节点数据结构完全和元素的数据结构分离。每个节点含指针指向元素，如上述例子。优点是一个元素能同时置于多个链表，缺点是需要动态分配内存。

**侵入式表(intrustive list)**是另一种链表，其节点的数据结构被嵌入目标元素本身。此方法的最大好处是无需在动态分配节点。例如用以下方式。

```
class SomeElement
{
  Link<SomeElement> m_link;
}
```

## 5.4 字符串

### 5.4.1 字符串的使用问题

在C和C++中，字符串不是一个原子数据类型，而是实现为**字符数组**。面对可变长度的字符串，如果不强制长度，那么就得动态分配内存。

另一个问题是**本地化**。包括字体(font)、字符字形(character)、字符串的方向。例如希伯来文是由右向左的。游戏也需要优雅的处理译文比原文长或者短的情况。

### 5.4.2 字符串类

传递时要用引用传递或者地址传递。否则可能引起一次拷贝构造函数的调用。

### 5.4.3 唯一标识符

游戏中所有资源都需要**唯一标识符**。不过当需要比较标识符的速度时，strcmp()完全不能达到要求，需要方法既有字符串的表达能力和弹性，又有整数表达的速度。

#### 5.4.3.1 字符串散列标识符

把字符串**散列**是个好方案，字符串散列码能如整数般比较，并且凭散列码可以拿回字符串。虚幻中FName就是范例。

#### 5.4.3.2 一些关于实现的注意

从字符串产生字符串标识符的过程，有时候称为**字符串扣留**。该过程及其缓慢，因为需要进行散列，并分配内存加进全局字符表中。以下是internString()的实现方法之一。

```
typedef U32 StringId;

extern StringId internString(const char* str);
static HashTable<StringId, const char*> gStringIdTable;
StringId internString(const char* str)
{
  StringId sid = hashCrc32(str);
  HashTable<StringId, const char*>::iterator it =
gStringTable.find(sid);
  if(it == gStringIdTable.end())
  {
    // 这里复制字符串是以防原来的字符串是动态分配的并将被释放
    gStringIdTable[sid] = strdup(str);
  }
  return sid;
}
```

虚幻引擎采用的把字符串标识符和C风格字符串包装进一个细小的类，这个类是FName。

### 5.4.4 本地化

#### 5.4.4.1 Unicode

ASCII是只有128个字符，所以在多语言适配时就不够用了，这时Unicode就出现了。Unicode是一系列标准，其中有以下几个。

- **UTF-8** - 这个标准每个字符占1~3个字节，此称为多字节字符集(multibyte character set，MBCS)。

它的优点是向后兼容ASCII编码。
- **UTF-16** - 每个字符都确切地使用16位，它是宽字符集(wide character set，WCS)。

在windows中，提供了各种标准的额方法，如下表。

| ANSI | WCS | MBCS |
|---|---|---|
| strcmp() | wcscmp() | _mbscmp() |
| strcpy() | wcscpy() | _mbscpy() |
| strlen() | wcslen() | _mbslen() |

表 5.1: 常用C标准库字符串函数，其ANSI、宽字符集及多字节字符集的版本。

其他一些需要考虑的有音频、图集等，因此需要使用CSV将各个翻译按照key对应起来，使用特定版本的翻译。

### 5.5 引擎配置

引擎需提供渲染、音乐和音效的音量，控制等。有一些只为开发团队提供。

#### 5.5.1 读/写选项

可配置选项可简单实现为全局变量或单例中的成员变量，并且需要能储存到硬盘上。以下是一些简单读/写可配置选项的方法。

- **文本配置文件** - 经常使用的是windows的INI文件，虚幻和OGRE均使用此。此外xml也是一个选项。
- **Windows注册表** - 注册表以树形式储存，当中的内部节点称为**注册表项**，叶节点则以键值对储存个别选项。微软引进注册表的原因就是要取缔日益膨胀的INI文件。
- **命令行选项** - 可扫描命令行去取得选项设置。引擎需提供机制，使游戏中的部分或全部选项都经过命令行设置。
- **环境变量**
- **线上用户设定档**

## 第6章 资源及文件系统

### 6.1 文件系统

游戏引擎的文件系统通常提供以下几类功能。

- **操作文件名和路径**。
- **开、关、读、写个别文件**。
- **扫描目录下的内容**。
- **处理异步文件输入、输出请求(做串流之用)**。

#### 6.1.1 文件名和路径

路径是一种字符串，用来描述某文件的位置。通常包含一个可选的卷指示符紧接一串路径成分，它们之间以/或者\隔开。

不同操作系统的路径结构有不同变化，以下列出Windows、UNIX操作系统家族、苹果操作系统的一些区别。

- UNIX使用正斜线符(/)作为路径分隔符。当代的windows系统支持正反斜杠。
- Mac OS X基于UNIX，因此支持正斜杠。
- UNIX没有卷这个概念，均挂载与根目录。
- :c这种是本地定义卷，远端网络分享则\some_computer_name\some_share。这种是通用命名规则(universal naming convention，UNC)。
- 均有当前目录的概念。

如果路径相对于根目录，则是绝对路径。如果相对于文件系统层次架构中的其他路径，则是相对路径。

#### 6.1.1.3 搜寻路径

**搜寻路径**是含一串路径的字符串，各路径之间以特殊字符隔开，找文件时就会从这些路径进行搜寻。

## 6.2 资源管理器

资源管理由两部分组成。

- 一部分负责管理离线工具链，用来创建资产及把它们转换为引擎可用的格式。
- 一部分负责在资源需要时载入，不需要时卸载。

#### 6.2.2.1 运行时资源管理器的责任

- 确保任何时候，同一个资源在内存中只有一份**副本**。
- 管理每个资源的生命期，载入需要的资源，并在不需要时卸载。
- 处理复合资源的载入。例如三维模型有网格、材质、贴图、动画等。
- 维护**引用完整性**。这包括外部引用完整性（单个资源内的交叉引用）及外部引用完整性（资源间的交叉引用）。当载入某资源时，需确保所有子资源已载入，并正确的修补所有交叉引用。
- 管理资源载入后的**内存用量**，确保资源储存在内存中合适的地方。
- 容许按资源类型，载入资源后执行**自定义的处理**。
- 通常提供单一统一接口管理多种资源类型。理想的，还可提供接口用于扩展。
- 处理**串流**。

有些引擎会把多个资源包裹成单一文件，如ZIP存档或其他复合文件。这个手法的优点是减少载入时间。载入时间主要由三点决定：**寻道时间、开启文件时间、从文件读入数据至内存的时间**。ZIP格式有以下几个优点。

- ZIP是开放格式。zlib SDK（读）和zziplib SDK（写）是免费的。
- ZIP存档内的虚拟文件也有相对路径。
- ZIP存档可被压缩。更主要的是可减少载入时间。
- ZIP存档可用于模块。例如多语言适配时，不同语言的资源打成一个ZIP包。

虚幻采用类似这种方式。虚幻中所有资源需置于大型合成文件之中，这些文件称为包(package)。

#### 6.2.2.5 资源注册表

当某资源已经被加载进内存时，就存入全局字典中，键是GUID，值是资源对象。若是找不到该资源，就加载资源。不过这会有许多问题，例如卡顿。因此引擎可采取两种方法。

- 在游戏进行中，完全禁止加载资源。游戏关卡的所有资源在加载界面已全部加载完毕。
- 资源以异步的方式进行。

#### 6.2.2.6 资源生命期

资源管理器的职责之一就是管理资源生命期——可能是自动的，也可能是通过对游戏提供所需的API函数，供手动管理资源生命期。

每个资源对生命期各有不同需求。

- 有些资源必须在开始时载入，并驻留在内存直至游戏结束。例如，玩家角色的网格、材质、纹理、核心动画等。称之为LSR(load-and-resident)资源。
- 有些资源对应单个关卡，首次看到该关卡时，存到内存，下一关卡时卸载。
- 某些生命期短于所在关卡。例如过场动画中用的动画和音频等。
- 有些资源如背景音乐、环境音效等可以在使用时即使串流。

某资源在何时载入并不难回答，问题是何时卸载。因为有些资源会在多关卡共享，为了避免卸载又加载，可以使用引用计数方法。例如关卡1使用了资源A、B，关卡2使用了资源B、C、D、E，核心就是载入新关卡时遍历该关卡所需资源(未加载前)，引用计数加1。然后遍历要卸载的，减一。那么引用计数变为0的卸载，刚从0变为1的加载。该过程如图。

| 事件 | A | B | C | D | E |
|---|---|---|---|---|---|
| 起始状态 | 0 | 0 | 0 | 0 | 0 |
| 关卡1的引用计数加1 | 1 | 1 | 1 | 0 | 0 |
| 载入关卡1 | (1) | (1) | (1) | 0 | 0 |
| 玩关卡1 | 1 | 1 | 1 | 0 | 0 |
| 关卡2的引用计数加1 | 1 | 2 | 2 | 1 | 1 |
| 关卡1的引用计数减1 | 0 | 1 | 1 | 1 | 1 |
| 卸下关卡1，载入关卡2 | (0) | 1 | 1 | (1) | (1) |
| 玩关卡2 | 0 | 1 | 1 | 1 | 1 |

表 6.2: 当载入/卸下两个关卡时，资源的引用变化。

()代码就要加载或卸载。

#### 6.2.2.9 处理资源间的交叉引用

- **使用全局统一标识符做交叉引用** - 需要维护一个全局资源查找表。每次载入资源对象至内存后，都要把其指针以GUID为键加入查找表中。当所有对象都加载进内存后，就可以扫描所有对象一次，对其交叉引用的资源对象GUID，通过全局查找表换成指针。
- **指针修正表** - 储存对象至二进制文件的另一常用做法是把**指针**转换为**文件偏移值**。

**— — — — — — — — — — — — 需要补充 — — — — — — — — — — —**

#### 6.2.2.10 载入后初始化

从序列化的文件加载至内存后，经常需要做一些处理才能使用，例如网格需要准备顶点缓冲、索引缓冲等。所以一般提供一个Init()和Destroy()方法做载入与卸载的处理。

# 第7章 游戏循环与实时模拟

游戏引擎需要很多种时间，包括实时、游戏时间、动画的本地时间线、某函数实际消耗的CPU周期等。

## 7.1 渲染循环

GUI例如Windows和Macintosh中的GUI会用一个矩阵失效的技术渲染，但是日常的3D游戏则使用以下架构。

```
while(!quit)
{
  updateCamera();
  updateSomeElements();
  renderScene();
  swapBuffers();
}
```

## 7.2 游戏循环

多数游戏引擎子系统都需要周期性的提供服务。例如动画经常是30帧和60帧，这是为了和渲染子系统同步。动力学则需要更高，如120帧。AI有时则是每秒一两次。

## 7.3 游戏循环的架构风格

### 7.3.1 视窗消息泵

在Windows中有一段代码称为**消息泵**。其基本原理是先处理来自Windows的消息，无消息时才执行引擎的任务。

```
while(true)
{
  MSG msg;
  while(PeekMessage(&msg, NULL, 0, 0) > 0)
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

  //无消息时执行游戏循环迭代一次
  RunOneIterationOfGameLoop();
}
```

当玩家在桌面上改变游戏的视窗大小或移动视窗时，游戏会愣住不动。

### 7.4 抽象时间线

- **真实时间** - 直接使用CPU的高分辨率计时寄存器来度量时间。此时间线的原点定义为计算机上次启动或重置之时。这种时间的度量单位是CPU周期(或倍数)，但其实只要简单乘以CPU的高分辨率计时器频率，此单位便可以转化为秒数。
- **游戏时间** - 可以定义**游戏时间线**，该时间独立于真实时间，可以通过减慢时间制作慢动作，或者停止更新时间暂停游戏等。暂停或者减慢游戏时间可以调试游戏，即冻结渲染，用调试摄像机去到场景中漫游查看渲染结果。
- **局部及全局时间线** - 每个动画和音频都含有一个**局部时间线**，加速、减慢甚至是反向播放动画，都可以视觉化为局部和全局时间线之间的**映射**。

### 7.5 测量及处理时间

### 7.6 多处理器的游戏循环

### 7.7 网络多人游戏循环

## 第8章 人体学接口设备(HID)

将输入送入游戏而设计的人体学接口设备种类繁多，例如摇杆、手柄、键盘、鼠标等。

### 8.2 人体学接口设备的接口技术

按设备的具体设计，游戏软件可用多种方式读取输入及写进输出。

#### 8.2.1 轮询

定期轮询(poll)硬件来读取输入意味着明确得查询设备的状态，有两种方法。

- **直接读取硬件寄存器**。
- **读取经内存映射的I/O端口**。

### 8.2.2 中断

有些HID只会在其状态改变时，才把状态传至游戏引擎，例如鼠标静止时不需要传。这类设备通常和主机以**硬件中断**的方式通信。所谓中断，是由硬件生成的信号，能让CPU暂停主程序，并执行一小段称为**中断服务程序**的代码。

## 8.5 游戏引擎的人体学接口设备系统

一般游戏引擎会在HID和使用之间添加一层，以实现一些功能。例如按键映射表。

### 8.5.1 典型需求

游戏引擎一般提供以下部分或全部功能。

- 死区(dead zone)。
- 模拟信号过滤(analog signal filtering)。
- 事件监测(event detection)(如按下和抬起按键)。
- 监测按钮的序列(sequence)以及多按钮的组合(又称为弦/chord)。
- 手势监测(gesture detection)。
- 为多位玩家管理多个HID。
- 多平台的HID支持。
- 控制器输入的重新映射。
- 上下文相关输入(context sensitive input)。
- 临时禁用某些输入。

### 8.5.2 死区

由于HID本质上是模拟式设备，其产生的电压含有噪声，以至实际上度量到的输入会轻微在$\mathrm{I}_0$附近浮动。此问题的常见解决方法是引入一个围绕$\mathrm{I}_0$的小死区。对于摇杆，死区可以定义为$[\mathrm{I}_0 - \delta, \mathrm{I}_0 + \delta]$，对于扳机，死区则定义为$[\mathrm{I}_0, \mathrm{I}_0 + \delta]$。

### 8.5.3 模拟信号过滤

为了解决不在死区范围的抖动，一般游戏引擎会过滤来自HID的原始信号。