# 游戏编程设计

## 设计模式篇

### 命令模式

概念：将一个请求封装成对象，从而使你可用不同的请求对客户进行参数化；对请求队列可用进行记录或回滚操作

代码：

class Command{public:

virtual ~Command() {}

virtual void execute() = 0;};

通过发送Command 执行函数

通过中间商记录Command队列，以及Command 实现Undo方法 为命令提供记录以及撤销功能

QFrame框架：使用Command用于Controller与Model之间通信。

优点：解耦,Controller和Model之间不直接通信。Command可以复用。增加和删除命令比较方便。容易设计一个组合命令。记录Command可以进行回滚操作。

缺点：每个操作都会产生Command类，增加了系统开发的复杂性。

### 享元模式

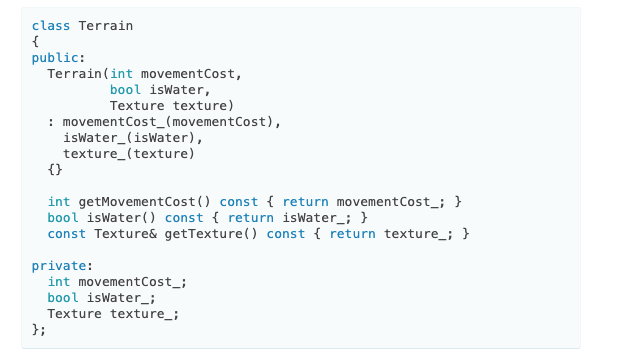
概念：主要用于减少创建对象的数量，以减少内存占用和提高性能。

用例：一片地图上有草，泥土，丘陵，湖泊，河流，以及其它任何你可以想到的地形。 我们基于区块建立地表：世界的表面被划分为由微小区块组成的巨大网格。 每个区块都由一种地形覆盖。不同的地形都有以下特效

1. 玩家移动速度
2. 判断是否可以用船通过，
3. 渲染的材质

这些特性只与该地形相关，与其上下文无关（即在世界的位置） 世界管理可以为每个地形实例化一个对象。地图都使用已实例化的对象而不是新建一个对象。

代码：





项目中：暂时没遇见

### 观察者模式

当一个对象被修改时，会自动通知依赖它的对象。

典型的用例：事件系统

优化点：可以添加对象池保存调用对象

项目中：事件系统,应该事件对应多个监听，使用者监听事件并注册方法回调。

优点：解耦

缺点：当物体被销毁没有移除监听将会造成卡死。

### 原型设计模式

用于创建重复的对象，同时又能保证性能。

典型的用例：Clone

好处：可以复制当前对象的状态

缺点：实现需要考虑深拷贝和浅拷贝

项目中：经常用于克隆数据，防止数据在操作时影响到原始数据

优点：安全

缺点：性能消耗较大

### 单例模式

保证一个类只有一个实例，并且提供了访问该实例的全局变量

典型用例：instance

特性：

1. 延迟实例化，在使用时才会实例。 好处：节省资源，坏处：不能控制在何时实例化。
2. 单例可继承。好处：可通过子类实现不同平台使用不同的实现。

FileSystem& FileSystem::instance(){

#if PLATFORM == PLAYSTATION3

static FileSystem \*instance = new PS3FileSystem();

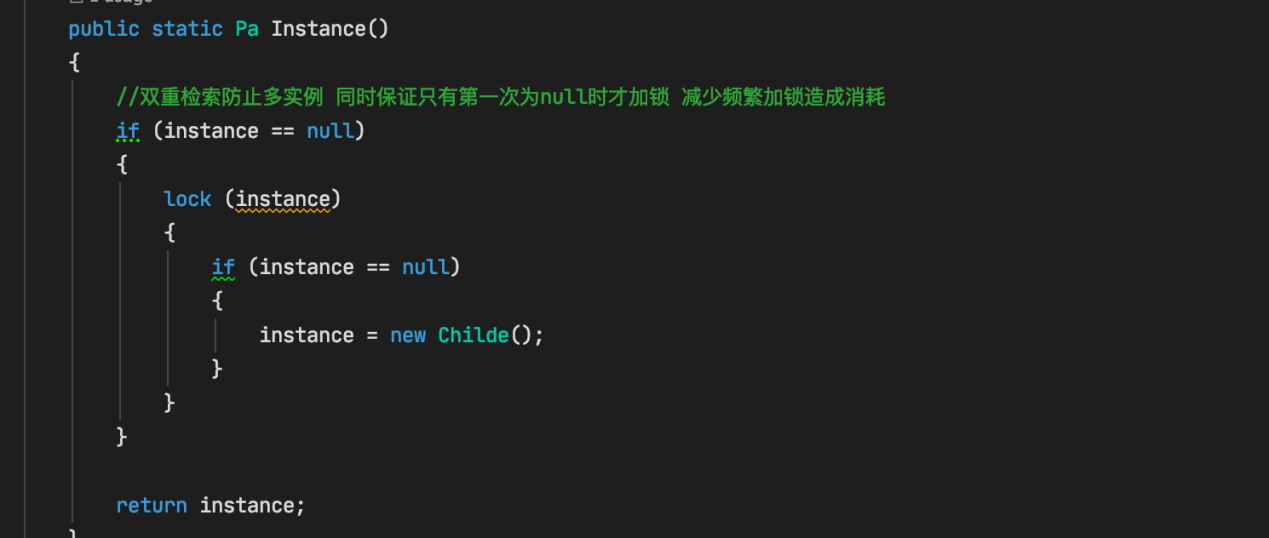
#elif PLATFORM == WII

static FileSystem \*instance = new WiiFileSystem();

#endif

return \*instance;}

缺点：使用懒汉实例化时可能导致多个实例生成。可以使用双重检索。



项目中：绝大多数全局变量都是使用懒汉+无锁单例实现。

好处：简单。

缺点：没有锁，可能造成多个实例。

**单例VS静态类：**

1. 单例类更像一个具有全局访问点的对象，静态类则像一个静态方法的集合
2. 静态类效率更高
3. 单例可以进行继承，静态类不行
4. 单例类可以延时初始化。

什么时候更应该使用静态类？

1. 当你类中没有任何成员变量，只有一些公共方法时，使用静态类会快很多。
2. 工具类

其他情况需要提供全局访问点时，使用单例会更优。

### 状态模式

概念：允许一个对象在其内部状态发生变化时改变自己的行为，该对象看起来就像修改了它的类型。

典型的例子：状态机

有限状态机: 有限个数状态以及在这些状态之间进行转移以及动作

优点：将各个状态隔离。防止代码混乱

缺点：用户表达有限，只能进行每个状态硬编码

并发状态机：多个状态机共同工作。比如玩家拿枪时可以进行之前的没枪时任何操作，如果按有限状态机 就需要将其分为拿枪状态和没拿枪状态(n\*m)。 如果使用两个状态机一个控制玩家一个控制枪 则只需要添加枪的新状态（ n+m）。

优点：可以合并多个状态机，当状态机之间互相独立时工作良好。

缺点：当状态机之间需要交互时比较棘手

分层状态机：有些操作是通用的，将其作为父状态，子状态继承该状态，运行时先检测子类操作，子类操作没反应时再检测父类操作。

优点：实现状态的重用

缺点：让子状态和父状态之间建立了比较强的耦合。

下推状态机：在状态机中加入栈用于存储状态的进出顺序。

优点：可以实现状态的回退。回到上一个状态。

缺点：实现比较复杂，状态迁移需要添加入栈和出栈操作，同时还需要考虑栈的情况进行迁移。

建议：目前游戏AI移向了行为树和规划系统。

项目中：切换波次使用简单的state标记当前状态，update里面根据state执行相应的方法并更新state

好处：简单易于实现。

缺点：各个状态之间的方法会有一定耦合。所有代码集中到一个类中，划分不清晰。

## 序列模式篇

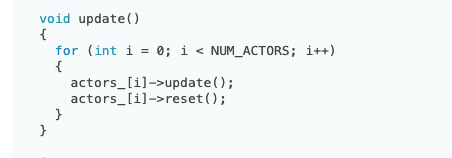
### 双缓冲模式

意图：用序列的操作模拟瞬间和同时发送的事情

典型用例：计算机图像的绘制。 如果在同一帧 程序在往帧缓冲中输入数据，同时屏幕在帧缓冲中读入数据。如果屏幕的读取速度超过了输入速度，将会导致屏幕绘制不完全。使用双缓冲，程序往当前帧缓冲中写入数据，屏幕从已经写好的上一帧中读取数据就不会导致屏幕绘制不完全。

示例代码：如果当前的人被打 他会打下一个人

原始情况： 0->1->2->0 按这个顺序 初始值为0被打



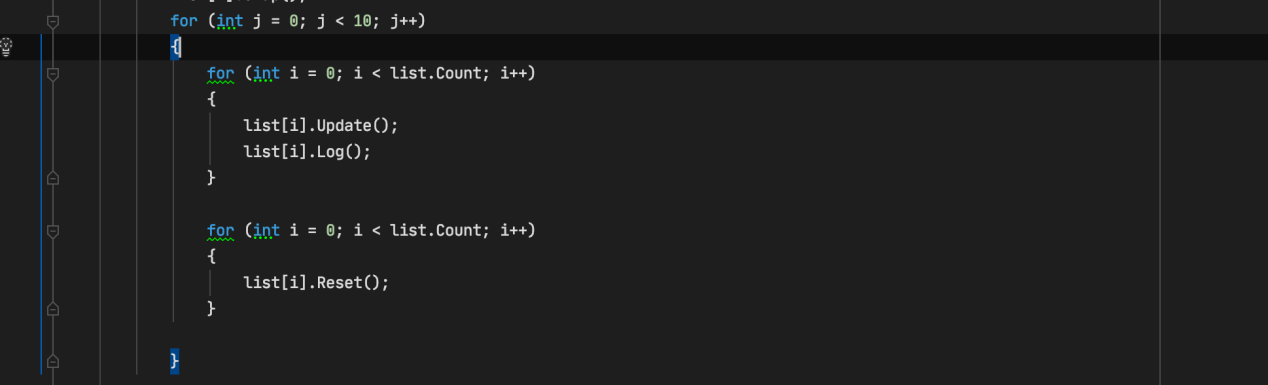
输出结果为 0被打 1被打 2被打 下一帧：0被打 1被打 2被打

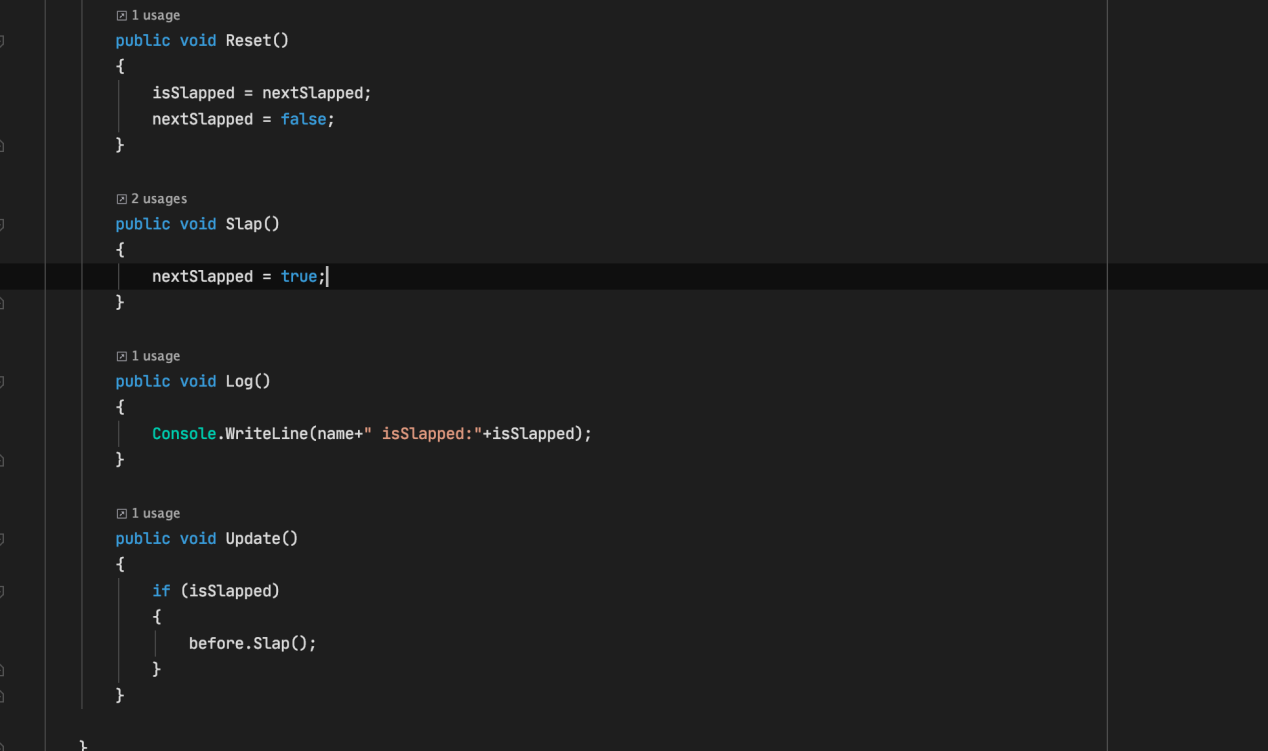
如果初始值为2被打

输出结果为 0没被打 1没被打 2被打 下一帧：0被打 1没被打 2没被打

两个输出结果不一样 与排序顺序相关

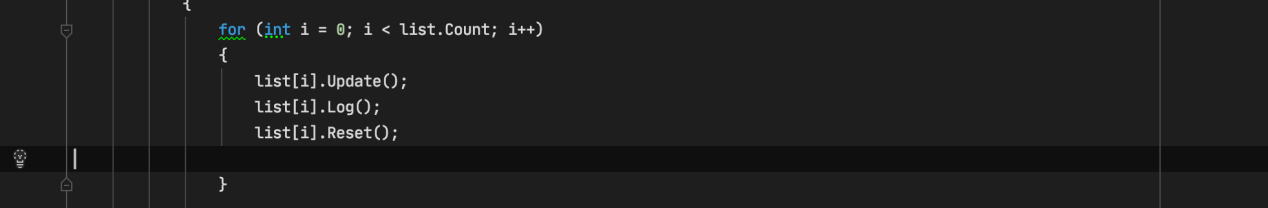
使用双缓冲：





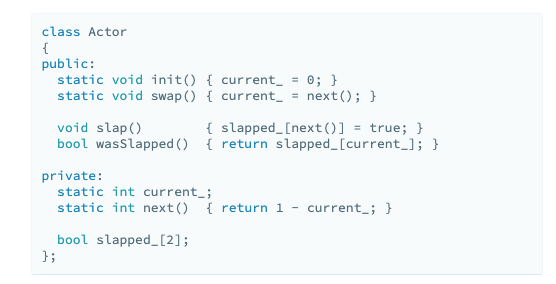
此时无论如何排序 结果都为第二种情况 即每一帧只有一个人被打

错误代码： 将reset与update放入同一个循环中



原因：因为update是遍历当前帧 如果在update后立即Reset将会把当前帧的情况替换为下一帧 将会导致错乱。 例如：2当前帧为被打 他去修改0下一帧的情况，因为0已经将当前帧替换为下一帧，将会导致2修改的是 0 下一帧的下一帧。

此案例优化点：将相同的表现的缓存统一。该示例中，角色需要全体切换一下缓存状态



actor通过静态变量current来获取当前缓存状态。（注意 slapped为常量 为Actor自身变量） 此时Actor整体只需要调用一次swap即可完成状态切换

FrameWork：在自动q

### 事件队列

目的：解耦发出消息或事件的时间与处理他的时间解耦

动机：程序并不想当用户操作时就立即中断当前操作，而是将操作存储当等到处理时再把操作提取出来。

模式：事件队列按先进先出的顺序存储一系列通知或请求。发送通知时，将请求放入队列，处理请求的系统在之后从队列中获取请求并处理。这解耦了发送者和接受者。（即将event存储当想要时再获取event）

设计点：

1.队列中存储的信息

①存储事件：存储事件更像观察者模式，告知这个事情已经发送。 可以有多个监听者。访问队列模块更广。（ps:如果是事件的话我觉得更应该使用事件系统，由监听者自己实现个队列来存储事件）

②存储消息：消息更像是发送在未来的事情，比如“播放声音”，可以设为服务的异步。监听者应该只有一个。

1. 谁能从队列中读取

①单播：好处：队列读取成了读者该实现的细节。队列更封装。无需担心竞争。

②广播：缺点：需要由广播系统处理队列。需要广播系统进行事件过滤。

项目中：战斗中的trigger。使用简单的事件系统广播，由处理者自己监听并存储消息，在需要时再处理事件。

优点：简单，由处理者自己决定合适使用。

缺点：没使用过，暂且不知。

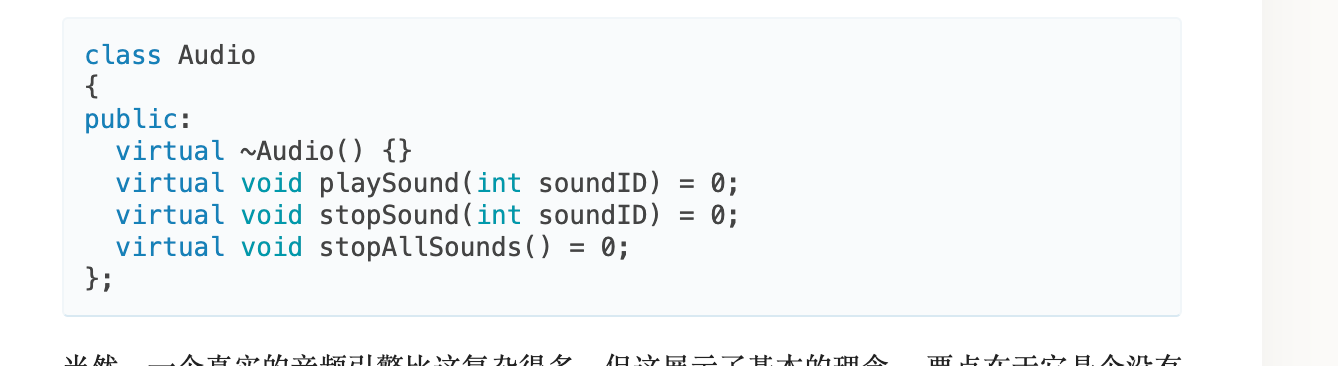
### 服务定位器

目的：提供服务的全局接入点，避免使用者和服务者的具体耦合

动机：有些通用系统需要给外部一个统一的调用接口，类似于静态或者单例。使用者直接通过访问具体的类来调用，倘若服务者由A变为B改变，将会去修改使用者的指向。

模式：服务类定义了一堆操作的抽象接口。具体的服务者实现改接口。分离的服务定位器提供了获取服务类的方法，同时隐藏了服务提供者的具体细节和定位他的过程。

代码：服务类



定位器：



设计点：1.服务如何初始化

①外部代码调用：好处：简单快捷。可以控制如何构造服务者。可以在游戏中改变服务者。缺点：定位器依赖外部，有可能服务没有被初始化。

②在编译时绑定：好处：快捷。可以保存服务不会为空。缺点：无法轻易的改变服务者



③在运行时通过配置设置：好处：更换服务灵活。不是程序员也可以改变服务。缺点：复杂。加载服务需要消耗时间。

1. 服务不存在怎么处理

①使用者处理：好处：由使用者决定如何处理失败。缺点：使用者必须处理失败，可能会有重复代码。

②终止游戏（assert）：好处：服务没有被找到会强迫程序员处理。使用者不用处理失败的情况。缺点：游戏会终止直到被解决，多人合作中会阻碍其他人。

③返回空服务：好处：可以确保服务不为空。服务者不用处理缺失服务。缺点：不易查找偶然bug

项目中：案例 ：dataManager里面存储了所有的Controller，在编译期进行绑定，使用require进行初始化，服务不存在就会终止游戏。

好处：简单，有问题会立即发现

缺点：当有问题会阻塞到其他协作者。无法动态切换服务者（虽然也不需要）。

### 优化模式

内存优化：

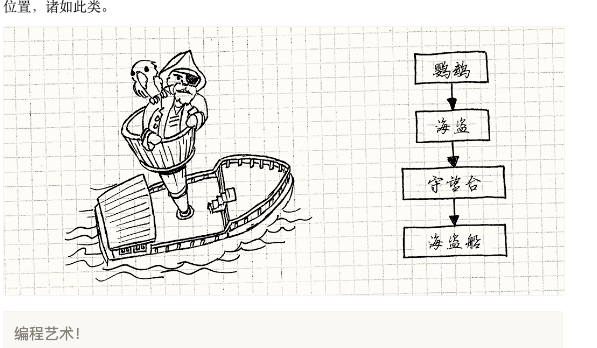
目的：尽量将数据放在一堆，让CPU可以尽快的读取。（不细说，这个涉及到更底层的读取内存，可以优化更好，不能优化也影响不大）

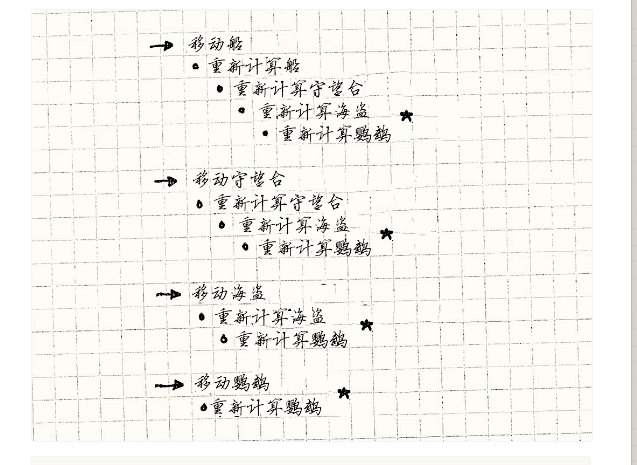
项目中：没有

### 脏标记

目的：将工作延期到需要结果时再执行，避免不要的工作。

动机：每次移动父物体会对子物体造成影响。如果父物体只要一有变动就立即修改子物体这会造成许多不必要的计算。





模式：一组原始数据随着时间变化而变化。使用代价昂贵的过程推导数据。使用脏标记追踪导出数据是否与原始数据一致。它在原数据被修改时设置。当导出时，该数据被设置了才进行推导，并清理标志，否则使用之前的数据。

设计点：

1.什么时候处理脏标记

①当结果被请求时：好处：可以最少的避免计算。缺点：如果计算消耗过大会导致可察觉的卡顿。

②在精心设计的检查点：好处：不会影响到玩家体验。缺点：在工作时失去控制权，你不能保证玩家一定会满足检查点。

③后台处理：好处:可以控制工作频率。缺点：需要做更多的冗余工作。需要异步工作，保证玩家可以同时继续做事。

1. 脏追踪的粒度大小

①粒度更细（一个变化对应一个对象）：好处：你只需要正在改变的数据 缺点：在存储标记上消耗更多。准备工作开销变多。

②粒度更大（一个变化对应一群变化）：好处：存储消耗变少，准备工作变少。 缺点：你需要处理没有变化的数据。

项目中做法：使用应该简单的bool类型用于标记状态是否改变，用于减少网络请求和数据更新。

### 对象池

目的：提高性能和内存通过一个池子重用对象，而不是单独的创建和回收对象。

动机：频繁的创建和销毁对象会导致内存中产生过多的内存碎片，对于内存较大且频繁创建和销毁的物体会导致创建耗时过长。

模式：定义一个池对象，其中包含了一组重用对象。其中每个重用对象都支持一个状态表明其是否正在使用。当你需要对象时，像池子要一个。它找到一个可用对象，初始化并返回。通过这种方式可用快速的创建和销毁对象而不必分配内存和其他资源。

缺点：①对象池可能会在不需要的对象上浪费内存。

②重用对象不会自动清理。

设计点：

1.对象是否与对象池耦合

①耦合：好处：实现比较简单。你可以确保对象都是由对象池创建。可以避免一个显式的active（可通过其他状态来判断）

②不耦合：好处：通用，可以使用泛型。

2.谁来负责初始化

①对象池：好处：对象池可以完全的封装对象。缺点：对象如果有多个初始化方法对象池需要提供多个接口。

②外部：好处：对象池提供的接口简单。缺点：外部会与对象耦合。

FrameWork中做法：使用泛型对象池，内部使用List存储所有对象，对象实现通用的接口，实现Clear和Init方法，创建和回收时由对象池对其进行操作。每次寻找时找到第一个没有正在使用的，没有则创建一个默认的。回收时，判断对象池是否超过设置的最大存储值，如果超过则释放掉没有使用的对象且长期没有使用的（对象中有个时间计数 记录最后一次使用的时间）。其中池子中有个update 当自动清理时间到了也会进行清理。

好处：具有自动释放，更智能，可以确保List不会过大。对对象进行了封装，外部不会涉及到对象的初始化和回收。更适合大项目。

缺点：实现复杂。对象池中会提前初始化，占用内存空间。

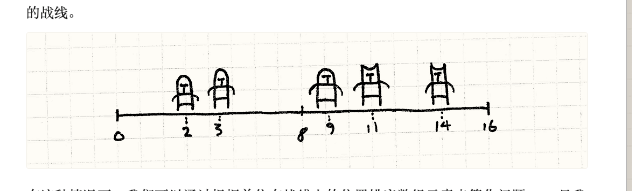
项目中做法：使用普通对象池，内部使用栈存储未被使用的对象。当对象需要时进行弹出，没有则新建一个。

好处：实现简单，更适合充当临时的对象池，如滑动列表。不会有没被使用过的对象。

缺点：不会自动释放对象，在极端情况下对象池会很大。初始化和清理状态都由使用者自己处理。

### 空间分区

目的：将对象根据它们的位置存储在数据结构中，来高效地定位对象。

示例：通过优先队列来存储位置 可让人物更快的找到附近的敌人。

文章中使用了二维的例子，使用二维数组将地图分格，然后根据玩家位置指定到对应的格子。格子中的数据使用链表结构，可存储所有在次格子中的玩家。

由于实现比较困难且目前用不上，暂且忽视。

想要学习的话，常见的算法有：

[Grid](http://en.wikipedia.org/wiki/Grid\_(spatial\_index))

[Quadtree](http://en.wikipedia.org/wiki/Quad\_tree)

[BSP](http://en.wikipedia.org/wiki/Binary\_space\_partitioning)

[k-d tree](http://en.wikipedia.org/wiki/Kd-tree)

- 每种空间划分数据结构基本上都是将一维数据结构扩展成更高维度的数据结构。知道它的直系子孙有助于分辨它是否适合解决当前的问题：

网格其实是持续的[桶排序](http://en.wikipedia.org/wiki/Bucket\_sort)。

BSP，k-d tree，和层次包围盒是[线性搜索树](http://en.wikipedia.org/wiki/Binary\_search\_tree)。

四叉树和八叉树是[多叉树](http://en.wikipedia.org/wiki/Trie)。