游戏编程模式

什么是好的软件架构？**架构是关于改动的。**

重访设计模式

命令模式

该模式的定义是：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。

```
class Command
{
  public:
    virtual ~Command(){}
    virtual void execute(Actor& actor) = 0;
};

class JumpCommand : public Command
{
  public:
  virtual void execute(Actor& actor)
  {
    actor.jump();
  }
};
//----  类似的命令  -----

Command* InputHandler::handleInput()
{
    if(isPressed(BUTTON_X)) return buttonX;
    if(isPressed(BUTTON_Y)) return buttonY;
    return NULL;
}

Command* command = inputHandle.handleInput();
if(command)
{
    command->execute(actor);
}
```

撤销和重做

在编辑器开发中，这种功能很常见。

```
class Command
{
```

```
  public:
    virtual ~Command(){}
    virtual void execute() = 0;
    virtual void undo() = 0;
};

class MoveUnitCommand : public Command
{
  public:
    MoveUnitCommand(Unit* unit, int x, int y) :
unit_(unit),xBefore(0),yBefore(0),x_(x),y_(y)
    {}

    virtual void execute()
    {
      xBefore = unit_->x();
      yBefore = unit_->y();

      unit_->moveTo(x_, y_);
    }

    virtual void undo()
    {
      unit_->moveTo(xBefore, yBefore);
    }
};
```

使用一个命令队列进行顺序执行。

游戏的重放功能就是如此，储存游戏的每帧的指令，播放即可。

享元模式

将公共的部分提取出来进行共享，不一的地方各自拥有各自的版本。类似于实例渲染。

```
class World
{
  public:
    World() : grassTerrain_(1, false, GRASS_TEXTURE), hillTerrain(3, false, HILL_TEXTURE),
riverTerrain_(2, true, RIVER_TEXTURE)
    {}
  private:
    Terrain* tiles_[WIDTH][HEIGHT];
    Terrain grassTerrain_;
    Terrain hillTerrain_;
    Terrain riverTerrain_;
};

void World::generateTerrain()
{
  for (int x = 0; x < WIDTH; x++)
  {
    for (int y = 0; y < HEIGHT; y++)
```

```
    {
      // 加入一些丘陵
      if (random(10) == 0)
      {
        tiles_[x][y] = &hillTerrain_;
      }
      else
      {
        tiles_[x][y] = &grassTerrain_;
      }
    }
  }

  int x = random(WIDTH);
  for(int y = 0; y < HEIGHT; y++)
  {
      tiles[x][y] = &riverTerrain_;
  }
}

const Terrain& World::getTile(int x, int y) const
{
    return *tiles_[x][y];
}
```

优化的金科玉律是**需求优先。**

观察者模式

MVC架构就是观察者模式，使用非常广泛，Java中的Observer，C#中的event。

```
//观察者的代码如下
class Observer
{
  public:
    virtual ~Observer(){}
    virtual void onNotify(const Entity& entity, Event event) = 0;
};

class Achievements : public Observer
{
  public:
    virtual void onNotify(const Entity & entity, Event event)
    {
      case EVENT_ENTITY_FELL:
        if(entity.isHero() && heroIsOnBridge)
        {
          unlock(ACHIEVENMENT_FELL_ON_BRIDGE);
        }
    }
};
```

```
//被观察者的代码如下
class Subject
{
  public:
    void addObserver(Observer* observer)
    {
        // 添加进列表
    }
    void removeObserver(Observer* observer)
    {
        //从列表中删除
    }

  protected:
    void notify(const Entity & entity, Event event)
    {
        for(int i = 0; i < numObservers; i++)
        {
            observers_[i].onNotify(entity, event);
        }
    }
  private:
    Observer* observers_[MAX_OBSERVERS];
    int numObservers;

};
```

需要注意的是不需要的观察者对象被删除之后记得从观察者列表中同时删除。

原型模式

关键思路是：**一个对象可以产出与它自己相近的对象**。如果你有一个恶灵，你可以制造更多恶灵。任何怪物都可以被视为**原型**怪物，产出其它版本的自己。

为了实现这个思路，我们给基类Monster添加一个抽象方法clone()。

```
class Monster
{
    public:
      virtual ~Monster(){}
      virtual Monster* clone() = 0;
};
```

每个怪物子类提供一个特定实现，返回与它自己的类和状态一模一样的新对象。如下。

```
class Ghost : Monster
{
    public:
      Ghost(int health, int speed):health_(health),speed_(speed){}

      virtual Monster* clone()
      {
        return new Ghost(helath_,speed_);
      }
    private:
      int health_;
      int speed_;
};
```

一旦所有的怪物都支持这个，我们不需要为每个怪物类创建生产者类。我们只需要定义一个类。

```
class Spawner
{
  public:
    Spawner(Monster* prototype):prototype_(prototype){}

    Monster* spawnMonster()
    {
        return prototype_->clone();
    }
  private:
    Monster* prototype_;
};
```

为了得到恶灵生产者，我们创建一个恶灵的原型实例，然后创建拥有这个实例的生产者：

```
Monster* ghostPrototype = new Ghost(15,3);
Spawner* ghostSpawner = new Spawner(ghostPrototype);
```

效果如何

在实现clone的过程中，有很多问题，例如深拷贝还是浅拷贝。如果恶魔拿着叉子，叉子是否复制等。同时，这里的例子假设了每个怪物有独立的类作为前提条件，绝大多数引擎不是这样做的。

生产函数

还有一个思路是用函数指针储存生产过程，如下。

```
Monster* spawnGhost()
{
    return new Ghost();
}

typedef Monster* (*SpawnCallback)();
class Spawner
{
```

```
  public:
    Spawner(SpwanCallback spawn):spawn_(spawn){}

    Monster* spawnMonster()
    {
        return spawn_();
    }

  private:
    SpawnCallback spawn_;
};

//这样做即可
Spawner* ghostSpawner = new Spawner(spawnGhost);
```

模版

```
template<class T>
class SpawnerFor : public Spawner
{
    public:
      virtual Monster* spawnMonster()
      {
        return new T();
      }
};
//这样使用即可
Spawner* ghostSpawner = new SpawnerFor<Ghost>();
```

为数据模型构建原型

```
{
  "name": "goblin grunt",
  "minHealth": 20,
  "maxHealth": 30,
  "resists": ["cold", "poison"],
  "weaknesses": ["fire", "light"]
}

{
  "name": "goblin wizard",
  "prototype": "goblin grunt",
  "spells": ["fire ball", "lightning bolt"]
}
```

单例模式

定义：保证一个类只有一个实例，并且提供了访问该实例的全局访问点。

经典的单例模式如下：

```
class FileSystem
{
  public:
    static FileSystem& instance()
    {
        if(instance_ == NULL) instance_ = new FileSystem();
        return *instance_;
    }

  private:
    FileSystem(){}

  static FileSystem* instance_;
};
```

静态的instance_成员保存了一个类的实例，私有的构造器保证了它是唯一的。比较现代化的方案是如下这种方式。

```
class FileSystem
{
  public:
    static FileSystem& instance()
    {
        static FileSystem* instance = new FileSystem();
        return *instance;
    }
  private:
    FileSystem(){}
};
```

静态局部变量在c++ 11标准中只执行一次，即使在多线程情况下。而前面的例子却不是。

为什么使用单例

- **如果没人用，就不必创建实例。**省那么一丢丢CPU时钟。
- **运行时实例化。**使用静态变量时编译器在main()运行前初始化静态变量。所以依赖次序解决不了。
- **可继承单例。**这个很有用，例如在跨平台的文件系统封装类。为了达到这一点，我们需要它变成文件系统抽象出来的接口，而子类为每个平台实现接口。

如下代码诠释了跨平台接口的开发。

```
class FileSystem
{
  public:
    virtual ~FileSystem(){}
    virtual char* readFile(char* path) = 0;
    virtual void writeFile(char* path, char* contents) = 0;
};

class PS4FileSystem : FileSystem
{
  public:
    virtual char* readFile(char* path){ /* 索尼的接口 */}
```

```
    virtual void writeFile(char* path, char* contents){/* 索尼的接口 */}
};

// 变成单例
class FileSystem
{
  public:
    static FileSystem& instance();
    //...

  protected:
    FileSystem() {}
};

// 这里是主要跨平台的接口
FileSystem& FileSystem::instance()
{
    #if PLATFORM == PLAYSTATION4
        static FileSystem* instance = new PS4FileSystem();
    #elif PLATFORM == WII
        static FileSystem* instance = new WiiFileSystem();
    #endif

    return *instance;
}
```

为什么后悔使用单例

- **理解代码更加困难。**ps：个人感觉并没有。
- **促进了耦合的发生。**如果在物理系统中错误的将音效#include了，则打乱了整个精心设计的架构。
- **对并行不友好。**多线程中需要小心处理。

将类限制为单一的实例

不需要提供对实例的公众、全局访问时，可以提供一个全局接触点消弱整体架构。以下是一种方法。

```
class FileSystem
{
  public:
    FileSystem()
    {
        assert(!instantiated_);
        instantiated_ = true;
    }

    ~FileSystem() {instantiated_ = false;}

  private:
    static bool instantiated_;
};

bool FileSystem::instantiated_ = false;
```

这个实现的缺点是只在**运行时**检查并阻止多重实例化。单例模式相反，通过类的自然结构，在编译时就能确定类是单一的。

为了给实例提供方便的访问方法

通用原则是在完成工作的同时，将变量写的尽可能局部。在我们拿起有**全局**范围影响的单例对象前，先考虑考虑代码中其他获取对象的方式。

- **传进来**。考虑渲染对象的函数，需要接触一个代表图形设备的对象，将其传给所有渲染函数是很自然的。通常是Context之类的参数。但是有一些情况不该传，例如AI中不该有Log相关的参数，得想其它方法。
- **从基类中获得**。代码如下。

```
class GameObject
{
  protected:
    Log& getLog() {return log;}
  private:
    static Log& log;
};
class Enemy : public GameObject
{
  void doSometing()
  {
    getLog().write("i can log.");
  }
};
```

- **从已经是全局的东西中获取**。可以让现有的全局对象捎带需要的东西，来减少全局变量类的数目，如下。

```
class Game
{
   public:
    static Game& intance() { return instance_;}

    Log& getLog() {return *log_;}
    FileSystem& getFileSystem() {return *fileSystem_;}
    AudioPlayer& getAudioPlayer() {return *audioPlayer_;}

  private:
    static Game instance_;

    Log *log_;
    FileSystem *fileSystem_;
    AudioPlayer *audioPlayer_;
};
```

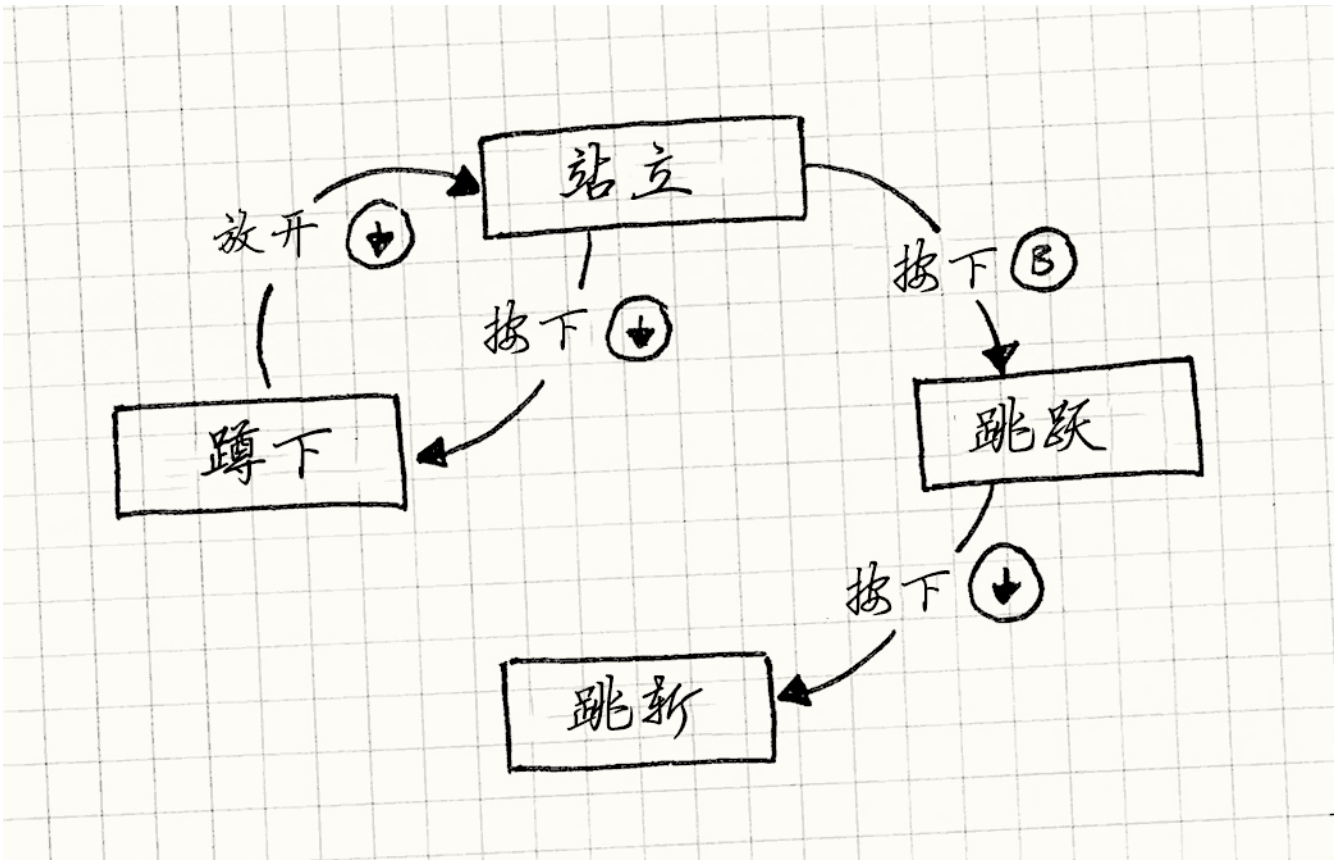
这样，只有Game是全局可见的。函数可使用函数访问其它系统，如。

```
Game::instance().getAudioPlayer().play(VERY_LOUD_BANG);
```

- **从服务定位器中获得**。这是一种模式，参照专门的章节。

状态模式

有限状态机的效果图如下所示。



有限状态机有几个要点。

- **你拥有状态机所有状态的集合。**如上面的站立、跳跃、蹲下等。
- **状态机只能在一个状态。**防止这个是使用有限状态机的理由之一。
- **一连串的输入或事件被发送给状态机。**
- **每个状态有一系列的转移，每个转移与输入和另一状态有关。**

这就是核心部分的全部了：**状态**、**输入**和**转移。**

状态模式的定义：**允许一个对象在其内部状态发生变化时改变自己的行为，该对象看起来好像修改了它的类型。**示例代码如下。

```
//定义状态接口
 class HeroineState
 {
     public:
       virtual void handleInput(Heroine& hearoine, Input input);
       virtual void update(Heroine& heroine);
 };
//为每个状态定义一个类
class DuckingState : public HeroineState
{
    public:
      DuckingState():chargeTime_(0){}
```

```
      virtual void handleInput(Heroine& herine, Input input)
      {
        if(input == RELEASE_DOWN)
        {
          //改为站立
          heroine.setGraphics(IMAGE_STAND);
        }
      }
      virtual void update(Heroine& heroine)
      {
        chargeTime_++;
        if(chargeTime_ > MAX_CHARGE)
        {
          heroine.superBomb();
        }
      }
    private:
      int chargeTime_;
};
// chargeTime移出了Heroine，因为该变量只供该状态使用。
```

状态委托

需要在Heroine类中添加当前状态的指针，放弃switch，转向状态委托：

```
class Heroine
{
  public:
    virtual void handleInput(Input input)
    {
        state_->handleInput(*this, input);
    }
    virtual void update()
    {
        state_->update(*this);
    }
  private:
    HeroineState* state_;
};

// 当状态改变时，将state对象指向对应的状态即可。
```

入口行为和出口行为

当英雄状态改变时，有可能需要改变贴图。现在，那部分代码在它转换前的状态中。代码如下。

```
HeroineState* DuckingState::handleInput(Heroine& heroine,Input input)
{
    if(input == RELEASE_DOWN)
    {
        heroine.setGraphics(IMAGE_STAND);
        return new StandingState();
    }
}
```

我们想做的是，每个状态控制自己的贴图，可以通过添加一个入口实现。

```
class StandingState : public HeroineState
{
public:
  virtual void enter(Heroine& heroine)
  {
    heroine.setGraphics(IMAGE_STAND);
  }
};
```

这样代码就可以简化成如下了。

```
void Heroine::handleInput(Input input)
{
  HeroineState* state = state_->handleInput(*this, input);
  if (state != NULL)
  {
    delete state_;
    state_ = state;

    // 调用新状态的入口行为
    state_->enter(*this);
  }
}

// 俯卧代码就可以简化成如下这样
HeroineState* DuckingState::handleInput(Heroine& heroine,
                                        Input input)
{
  if (input == RELEASE_DOWN)
  {
    return new StandingState();
  }
}
```

当然也可以扩展出口行为，此处并无示例代码。

并发状态机

有时需要拿着枪的时候可以跳跃、蹲下等。问题在于我们将两种状态绑定到了一个状态机上——它做的和它携带的。解决方法就是**使用两个单独的状态机。**

状态需要交互时，可以使用丑陋的if判断，虽然不优雅，但是可以解决问题。

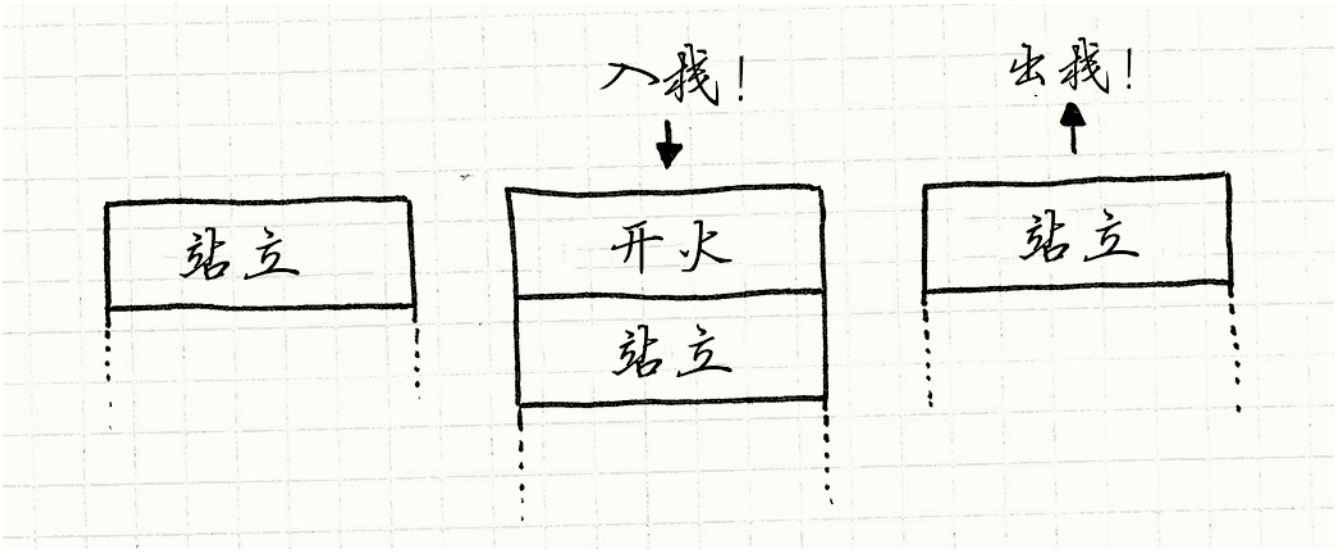
分层状态机

分层状态机可以很好的解决代码冗余问题，实现思路是当一个事件进来，如果子状态没有处理，它就会交给链上的父状态。

```
class OnGroundState : public HeroineState
{
  public:
    virtual void handleInput(Heroine& heroine, Input input)
    {
      if(input == PRESS_B)
      {
        // 跳跃..
      }else if(input == PRESS_DOWN)
      {
        // 俯卧..
      }
    }
};
class DuckingState : OnGroundState
{
  public:
    virtual void handleInput(Heroine& heroine, Input input)
    {
      if(input == RELEASE_DOWN)
      {
        // 站起..
      }else{
        OnGroundState::handleInput(heroine, input);
      }
    }
};
```

下推自动机

意思是记录上一个状态，以供状态机可以返回到上一个状态。



如上图这样，当一个状态结束时，自动切换到之前的状态。

结论

由于各种复杂的情况，产生了行为树和规划系统以供工程师使用。不过有限状态机在以下的情况有用：

- 有个实体，它的行为基于一些内在状态。
- 状态可以被严格地分割为相对较少的不相干项目。
- 实体响应一系列输入或事件。

在游戏中，状态机因在AI中使用而闻名，但是它也常用于其他领域， 比如处理玩家输入，导航菜单界面，分析文字，网络协议以及其他异步行为。

序列模式

双缓冲模式

意图是：**用序列的操作模拟瞬间或同时发生的事情。**双缓冲是为了解决场景必须快速流畅得更新而设计。

计算机图形系统是如何工作的（概述）

从帧缓冲中按照从左到右读取像素，写到屏幕上。如果同一帧缓冲发生了同时读写操作，那么就会造成撕裂问题。

模式

定义**缓冲类**封装了**缓冲**：一段可改变的状态。 这个缓冲被增量地修改，但我们想要外部的代码将修改视为单一的原子操作。 为了实现这点，类保存了*两个*缓冲的实例：**下一缓冲**和**当前缓冲。**

当信息从缓冲区中读取，它总是读取当前的缓冲区。 当信息需要写到缓存，它总是在下一缓冲区上操作。 当改变完成后，一个**交换**操作会立刻将当前缓冲区和下一缓冲区交换， 这样新缓冲区就是公共可见的了。旧的缓冲区成为下一个重用的缓冲区。

当满足以下要求时，使用这个模式：

- 需要维护一些被增量修改的状态
- 修改到一半时，状态可能会被外部请求
- 想要防止请求状态的外部代码知道内部的工作方式
- 想要读取状态，而且不想等着修改完成

需要记住的是**交换本身需要时间**，不过通常只是交换一个指针的时间（需要注意这个操作是原子性的）。另一个是**需要保存两个缓冲区**，这必然引起内存占用两份。

缓冲区是如何被交换的?

因为需要锁定两个缓冲区的读写，因此这个过程是越快越好。所以交换缓冲区的指针或者引用是惯例，因为：

- **速度快。**
- **外部代码不能存储对缓存的永久指针。**
- **缓冲区中的数据是两帧之前的数据，而不是上一帧的数据。**因为只能保存前一帧的数据，所以如果需要更旧的帧，就需要另想办法。

游戏循环

意图是：**将游戏的进行和玩家的输入解耦，和处理器速度解耦。**

事件循环

游戏和应用的区别是：游戏处理用户输入，但是不等待它。循环总是继续旋转：

```
while(true)
{
    processInput();
    update();
    render();
}
```

两个因素决定了循环运转的多块：**每帧做的工作**和**底层平台的速度。**所以游戏循环的另一个关键任务是：**不管潜在的硬件条件，以固定速度运行游戏。**

拥有游戏循环的是你，还是平台?

使用平台的事件循环：

- **简单。**
- **平台友好。**
- **失去了对时间的控制。**

使用游戏引擎的循环：**不必自己编写。**

自己写的话，虽然可以完全控制，但是需要与平台交互。难点是应用框架和操作系统需要时间片去处理自己的事情。

如何管理能耗?

原则就是在保证游戏性的同时，让CPU尽可能的休眠。移动游戏会用固定频率运行，这样当消耗的时间小于两帧的时间差，就让CPU休眠一会儿。

如何控制游戏速度?

固定时间步长，没有同步：

- 简单。
- 游戏速度直接受硬件和游戏复杂度影响。

固定事件步长，有同步：

- 电量友好。
- 游戏不会运行的太快。
- 游戏可能运行的太慢。

动态时间步长：

- 能适应并调整，避免运行太快或太慢。
- 让游戏不确定并不稳定。

固定更新时间步长，动态渲染：（它以固定的时间步长更新，如果需要赶上玩家的时间，可以扔掉一些渲染帧）

- 能适应并调整，避免运行太快或太慢。只要能实时更新，游戏状态就不会落后于真实时间。
- 更复杂。

更新方法

意图是：**通过每次处理一帧的行为模拟一系列独立对象。游戏世界**管理**对象集合**。每个对象实现一个**更新方法**模拟对象在**一帧**内的行为。每一帧，游戏循环更新集合中的每一个对象。

更新方法试用以下情况：

- 游戏有很多对象或系统需要同时运行。
- 每个对象的行为都与其它的大部分独立。
- 对象需要跟着时间进行模拟。

更新方法在哪个类中?

- **实体类中**。当类的种类很多时，一有新行为就建Entity子类来实现是痛苦的。所以业界已经远离这种方法了。
- **组件类。组件让你进一步解耦了单一实体中的各部分**。渲染，物理，AI都可以自顾自了。
- **委托类**。还可将类的部分行为委托给其他的对象。

如何处理隐藏对象?

游戏中各种原因会产生不需要更新的状态，怎么处理很重要。一种方案是**使用单个包括了所有不活跃对象的集合**，不过这种方案浪费时间。要么检测是否启用的标识，要么调用一些啥都不做的方法。

如果你使用单独的集合保存活动对象：

- **使用了额外的内存管理第二个集合**：当你需要所有实体时，通常需要一个更大的集合。在速度比内存重要时，这样做是值得的。
- **得保持集合同步**：当对象创建或销毁时，得修改全部对象集合和活跃对象集合。

方法选择的度量标准是不活跃对象的可能数量。 数量越多，用分离的集合避免在核心游戏循环中用到它们就更有用。

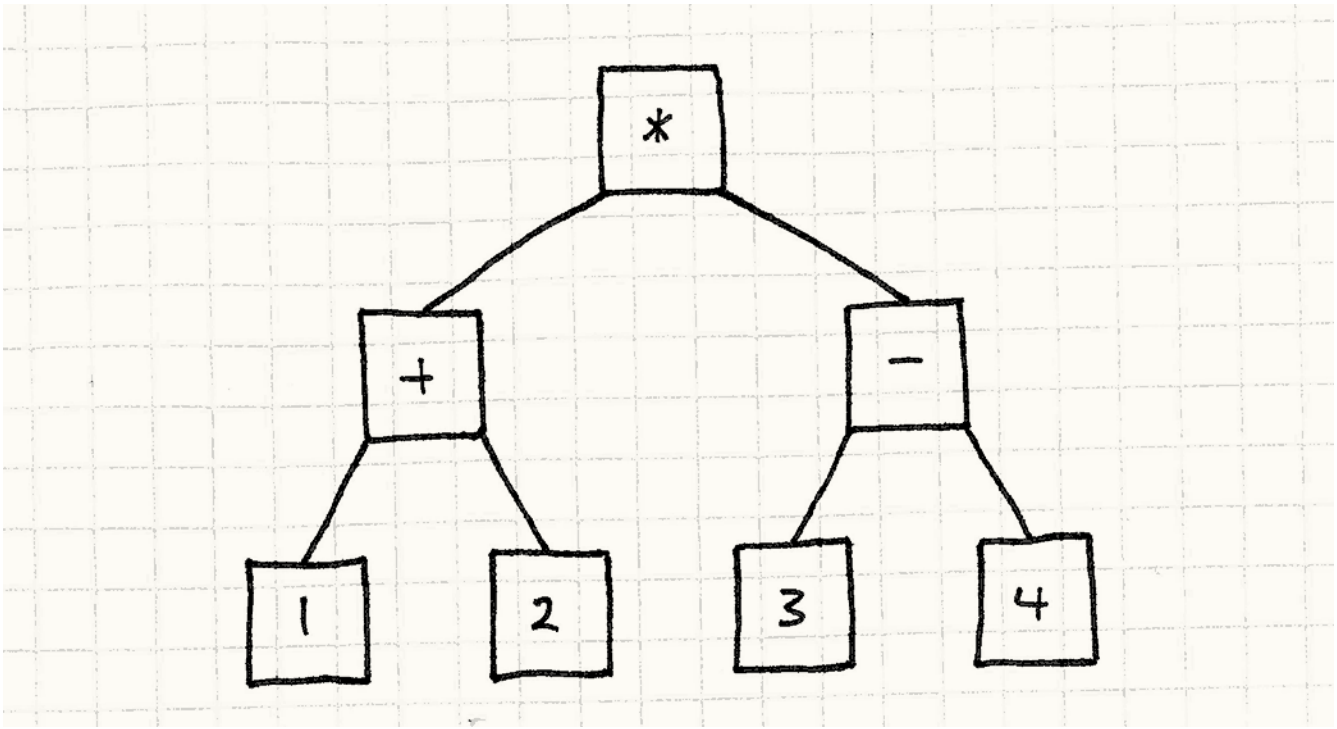
行为模式

行为，就是告诉游戏中每个实体做什么的剧本。**类型对象**定义行为的类别而无需完成真正的类。**子类沙盒**定义各种行为的安全原语。最先进的**字节码**，将行为从代码中剥离，放入数据文件中。

字节码

意图是：**将行为编码为虚拟机器上的指令，赋予其数据的灵活性。**

解释器模式



计算（1+2）*（3-4），解释器模式就是用来执行这棵树的。用面向对象的思路解决它们每一个都是一个对象，示例如下。

```
class Expression
{
  public:
    virtual double evaluate() = 0;
};

//数字
class NumberExpression : Expression
{
  public:
    NumberExpression(double value) : value_(value){}

    virtual double evaluate()
    {
      return value_;
    }
  private:
    double value_;
};

//加法
virtual double evaluate()
{
    double left = left_->evaluate();
    double right = right_->evaluate();
    return left + right;
}
```

解释性模式会带来一些问题：

- 从磁盘上加载它需要实例化并连接成吨的小对象。
- 这些对象和它们之间的指针会占据大量的内存。
- 数据缓存不友好，虚函数调用使之对指令缓存不友好。

虚拟的机器码

任何软件执行时都一样，都是机器码。包括CPU也一样，有这么几点好处：

- 密集。
- 线性。
- 底层。
- 速度快。

字节码+虚拟机等于自定义的一门语言，例如Lua。

模式

指令集定义了可执行的底层操作。一系列的指令被编码为**字节序列**。**虚拟机**使用**中间值栈**依次执行这些指令。通过组合指令，可以定义复杂的高层行为。

何时使用

子类沙箱

意图是：**用一系列由基类提供的操作定义子类中的行为。**

模式是：**基类**定义抽象的**沙箱方法**和几个**提供的操作**。将操作标为**protected**，表明它们只为子类所使用。每个推导出的**沙箱子类**用提供的操作实现了沙箱函数。

何时使用。

- 有一个能推导出很多子类的基类。
- 基类可以提供子类所需要的所有操作。
- 在子类中行为重复，想要更容易的在它们之间分享代码。
- 想最小化子类和程序的其它部分的耦合。

一句话总结就是一个合理继承。

类型对象

意图是：**创造一个类A来允许灵活地创造新类型，类A的每个实例都代表不同的对象类型。**

动机

每种怪物都有不同的品种，每个品种就是一种怪物。面向对象的思路就是“龙”是“怪物”，也就是Dragon是Monster的子类，解决思路如下。

```
class Monster
{
```

```
  public:
    virtual const char* getAttack() = 0;

  protected:
    Monster(int startingHealth) : health_(startingHealth){}
  private:
    int health_;
};

// 不同品种的getAttack返回的字符串不一样
class Dragon : public Monster
{
public:
  Dragon() : Monster(230) {}

  virtual const char* getAttack()
  {
    return "The dragon breathes fire!";
  }
};
```

这种模式的问题之一就是不能应对大量的品种。

为类型建类

就是建立一个Breed类，这个类其实就是怪物的**类型**，建立一个怪物的对象类，对象类引用类型类即可。主要解决的思路是这样可以实现数据驱动，配置表就可以生成大量不同的怪物。

模式

定义**类型对象**和**有类型的对象类**。每个类型对象实例代表一种不同的逻辑类型。每种有类型的对象保存**对描述它类型的类型对象的引用**。

何时使用

- 你不知道你后面还需要什么类型。（例如有新的怪物品种）
- 想不改变代码或者重新编译就能修改或添加新类型。

记住

- 需要手动追踪类型对象

  这个是必然了，需要使用类型对象，自己需要对它的引用负责。

- 更难为每种类型定义行为

  数据驱动的弊端就是行为难定义，可以通过预先定义好行为的方式解决。或者彻底解决的方案是：**解释器模式和字节码模式**让我们定义有行为的对象。

示例代码

```
class Breed
{
  public:
    Breed(int health, const char* attack) : health_(health),attack_(attack){}
```

```
    int getHealth(){return health_;}
    const char* getAttack(){return attack_;}

    // 让类型对象更像类型
    Monster* new Monster()
    {
        return new Monster(*this);
    }

  private:
    int health_;
    const char* attack_;
};

//怪物使用方式
class Monster
{
  friend class Breed;

  public:
    const char* getAttack()
    {
        return breed_.getAttack();
    }
  private:
     Monster(Breed& breed) : health_(breed.getHealth()),breed_(breed){}
    int health_;
    Breed& bread_;
};
```

把Monster的构造函数声明为私有，这样只能通过对象类型类生成。

```
Monster* monster = someBread.newMonster();
```

通过这种拆分的方式，可以让Bread中的**怪物对象由程序员进行内存管理**，例如用池缓存起来等。

还有两种方式可以更优雅的进行处理：

- 使用继承复用类型对象的数据
- 使用JSON

设计决策

- 类型对象是封装的还是暴露的?
    - 如果类型对象是封装的
        - **类型对象模式的复杂性对代码库的其它部分是隐藏的。**
        - **有类型的对象可以选择性地修改类型对象的重载行为。**因为对象持有类型对象的引用，所以调用自身方法时，可以方便地加入自己的逻辑。
        - **我们得为每个类型对象暴露的方法写转发。** 如果类型对象有很多方法，对象类也得为每一个方法建立属于自己的公共可见方法。

    - 如果类型对象是暴露的
        - **外部代码可以与类型对象直接交互，无需拥有类型对象的实例。**
        - **类型对象现在是对象公共API的一部分了。**
- 有类型的对象是如何创建的?
    - **构造对象然后传入类型对象。外部代码可以控制内存的分配**，因为调用代码也是构造对象的代码。
    - **在类型对象上调用"构造器"函数。类型对象控制了内存分配。**
- 能改变类型吗?

  有时候怪物死去可以复活为僵尸等等，各种情况都有。
    - 如果类型不改变
        - 编码和理解更容易
        - 更容易查找漏洞
    - 如果类型可以改变
        - 需要创建的对象更少。改变一个声明即可。
        - 我们需要小心地做约束。
- 支持何种继承?
    - 没有继承
        - 简单
        - 带来重复的工作。一个血量不同就被干翻
    - 单继承
        - 相对简单
        - 查询属性更慢。
    - 多重继承
        - 可以避免大多数代码重复
        - 复杂。一般都禁止多重继承。

解耦模式

能让我们更好地适应变化的工具是**解耦。组件模式**将一个实体拆分为多个，解耦不同的领域。**事件序列**解耦了两个互相通信的事物，稳定且及时。**服务器定位**让代码使用服务而无需绑定到提供服务的代码。

组件模式

意图是：**允许单一的实体跨越多个领域而不会导致这些领域彼此耦合。**

模式是：**单一实体跨越了多个领域。**为了保持领域之间相互分离，将每部分代码放入**各自的组件类**中。实体被简称为**组件的容器。**

在以下情况可以考虑使用组件模式：

- 有一个涉及了多个领域的类，而你想保持这些领域相互隔离。
- 一个类正在变大而且越来越难以使用。
- 想要能定义一系列分享不同能力的类，但是使用继承无法让你精确选取要重用的部分。

设计决策

引擎越大划分的组件越细。

对象如何获取组件?

- **如果对象创建组件**
  - **保证了对象总能拿到需要的组件。**
  - **重新设置对象比较困难。**变相的变得耦合。
- **如果外部代码提供组件**
  - **对象更加灵活。**对象变成了容器，可以添加各种各样的组件。
  - **对象可以与具体的组件解耦。**

组件之间如何通信?

- **通过修改容器对象的状态**
  - **保持了组件解耦。**共同使用的变量被改变了，它们互相不知道谁改的。
  - **需要将组件分享的任何数据存储在容器类中。**将可能共用的信息存入容器类中，会让所有组件都获得这样的消息。
  - **让组件的通信基于组件运行的顺序。**例如渲染总是在物理和动画之后。
- **通过它们之间互相引用**
  - **简单快捷。**
  - **两个组件紧绑在一起。**又纠缠在了一起。
- **通过发送消息**
  - **这是最复杂的选项。**需要建立一个消息系统，伪代码如下。

```
class Component
{
 public:
    virtual ~Component(){}
    virtual void receive(int message) = 0;
};
class ContainerObject
{
 public:
    void send(int message)
    {
        for..
        {
            if(components_[i] != NULL)
            {
                components_[i].receive(message);
            }
        }
    }
}
```

这样就造成了一些结果：

- **同级组件解耦。**唯一的耦合是信息。
- **容器类很简单。**容器类无需知道组件使用了什么数据，它只是将消息发送出去。这可以让组件发送领域特有的数据而无需打扰容器对象。

事件队列

意图是：**解耦发出消息或事件的时间和处理它的时间。**

音效是游戏中的一部分，如果点击按钮就在线程中播放音效，会怎样呢?

```
class Menu
{
 public:
    void onSelect(int index)
    {
        Audio::playSound(SOUND_BLOOP, VOL_MAX);
    }
};
```

这样做很容易就会造成几个问题。

- **API在音频引擎完成对请求的处理前阻塞了调用者。**因为涉及到加载，IO是很缓慢的过程。
- **请求无法合并处理。**因为一般分配了多个系统，例如渲染、物理、AI等。并且是多线程运行，代码没做线程同步。
- **请求在错误的线程上执行。**调用播放音频时，要求时立即执行。然而游戏系统在方便时调用播放，但是音频引擎不一定能方便地去处理这个请求。所以需要**将接受请求和处理请求解耦。**

模式

事件队列在队列中按先进先出的顺序存储一系列**通知或请求。**发送通知时，将**请求放入队列并返回。**处理请求的系统之后稍晚**从队列中获取请求处理。**这**解耦了发送者和接收者**，既**静态**又**及时。**

为了解决API阻塞引起的问题，可以推迟这项工作，为了达到这一点，需要**具体化**播放声音的请求。

```
struct PlayMessage
{
    SoundId id;
    int volume;
};

class Audio
{
 public:
    static void init()
    {
        numPending_ = 0;
    }
    //将播放放在update中
    static void update()
    {
        for (int i = 0; i < numPending_; i++)
        {
          ResourceId resource = loadSound(pending_[i].id);
          int channel = findOpenChannel();
          if (channel == -1) return;
          startSound(resource, channel, pending_[i].volume);
        }
        numPending_ = 0;
    }

 private:
```

```
    static const int MAX_PENDING = 16;
    static PlayMessage pending_[MAX_PENDING];
    static int numPending_;
};

void Audio::playSound(SoundId id, int volume)
{
    assert(numPending_ < MAX_PENDING);
    pending_[numPending_].id = id;
    pending_[numPending_].volume = volume;
    numPending_++;
}
```

环形缓存

保留了数组所有优点（**没有动态分配、没有为记录信息造成的额外开销、缓存友好**），同时能不断从队列的前方移除对象。

```
class Audio
{
 public:
    static void init()
    {
        head_ = 0;
        tail_ = 0;
    }
 private:
    static int head_;
    static int tail_;
}

void Audio::playSound(SoundId id, int volume)
{   // 确保不会收尾相接
    assert((tail_ + 1) % MAX_PENDING != head_);

    pending_[tail_].id = id;
    pending_[tail_].volume = volume;
    tail_ = (tail_ + 1) % MAX_PENDING;
}

void Audio::update()
{
    if(head_==tail_) return;
    //其它代码
    head_ = (head_ + 1) % MAX_PENDING;
}
```

合并请求

如果同时播放同一个音效，会导致音量过大的问题，所以一般解决办法是取这个音效的最大音量值进行播放，代码如下。

```
void Audio::playSound(SoundId id, int volume)
{
    for(int i = head_; i != tail_; i = (i + 1) % MAX_PENDING)
    {
        if(pending_[i].id == id)
        {
            pending_[i].volume = max(volume, pending_[i].volume);
            //无需入队
            return;
        }
    }
}
```

分离线程

在多核硬件上，最通用的策略是将每个单独的领域分散到一个线程——音频、渲染、AI等等。很容易做到的原因是以下几点：

- 请求音频的代码与播放音频的代码解耦
- 有队列在两者之间整理它们
- 队列与程序其它部分是隔离的

队列中存储了什么?

- **如果存储事件**
  - **很可能允许多个监听者。**队列中存储的是已经发生的事情，发送者不关心谁接受它。
  - **访问队列的模块更广。**事件队列通常**广播**事件到任何感兴趣的部分。为了尽可能允许所有感兴趣的部分访问，队列一般是全局可见的。
- **如果存储消息**

  消息或请求描绘的是想要发生在未来的事情。
  - **更可能只有一个监听者。**存储的消息只请求**音频API**播放声音。

谁能从队列中读取?

- **单播队列**

  在队列是类API的一部分时，单播是很自然的。
  - **队列变成了读取者的实现细节。**发送者知道的所有就是发条消息。
  - **队列更封装。**
  - **无需担心监听者之间的竞争。**
- **广播队列**

  大多数"事件"系统都是这样。一个事件进来，所有监听者均能看到。
  - **事件可能无人接收。**如果没有监听者，事件就消失了。
  - **也许需要过滤事件。**广播队列经常对程序的所有部分可见，很多事件乘以很多监听者，数量恐怖。
- **工作队列**

类似广播队列，有多个监听者。不同之处在于队列中的每个东西只会投到监听器**其中的一个**。常应用于将工作打包给同时运行的线程池。

- **需要规划。** 由于一个事物只有一个监听器，队列逻辑需要指出最好的选项。

谁能写入队列?

这个模式兼容所有可能的读/写设置：一对一、一对多、多对一、多对多。

- **使用单个写入器**
  - **隐式知道事件是从哪里来的。**只有一个对象可以向队列中添加事件，任何监听器都可以安全地假设那就是发送者。
  - **通常允许多个读取者。**
- **使用多个写入器**

  上例就是这样，代码库的任何部分都能给队列添加请求。
  - **得更小心环路。**
  - **很有可能在事件中添加对发送者的引用。**

对象在队列中的生命周期如何?

在C或C++中，需要开发者保证对象活得足够长。

- **传递所有权。**当消息入队时，队列拥有了它，发送者不再拥有。被处理时，接收者负责销毁。
- **共享所有权。**例如使用shared_ptr。
- **队列拥有它。**类似于对象池。

服务定位器

提供服务的全局定位点，避免使用者和实现服务的具体类耦合。

模式是：**服务**类定义了一堆操作的抽象接口。具体的**服务提供者**实现这个接口。分离的**服务定位器**提供了通过查询获取服务的方法，同时隐藏了服务提供者的具体细节和定位它的过程。

记住

- **服务真的可定位。**如果使用单例，获取实例肯定能用。然而**定位**服务有可能失败。
- **服务不知道谁在定位它。**服务应该在任何情况下都正确工作。所以如果你的类只在期望的上下文中使用，避免模式将它暴露给整个世界更安全。

一个简单的定位器

示例代码如下

```
class Locator
{
 public:
    static Audio* getAudio() {return service_; }
    static void provide(Audio* service)
    {
        service_ = service;
    }
 private:
    static Audio* service_;
```

```
};
// 使用服务
Audio* audio = Locator::getAudio();
audio->playSound();

//在游戏开始部分，提供服务
ConsoleAudio* audio = new ConsoleAudio();
Locator::provide(audio);
```

这里需要注意：定位器类没有与具体的服务提供者耦合。

这里还有更高层次的解耦：Audio接口没有意识到它在通过服务定位器来接受访问，因为它只是常见的抽象基类。这意味着我们可以将这个模式应用到**现有的**类上。

一个空服务

如果我们在服务提供者注册前使用服务，它会返回`NULL`。 如果调用代码没有检查，游戏就崩溃了。代码如下。

```
class Locator
{
public:
  static void initialize() { service_ = &nullService_; }

  static Audio& getAudio() { return *service_; }

  static void provide(Audio* service)
  {
    if (service == NULL)
    {
      // 退回空服务
      service_ = &nullService_;
    }
    else
    {
      service_ = service;
    }
  }

private:
  static Audio* service_;
  static NullAudio nullService_;
};
```

如果我们想暂停使用系统，空服务也可以派上用场。

服务是如何被定位的?

- **外部代码注册**:
  - **简单快捷**。getAudio()通常会被内联。
  - **可以控制如何构建提供者**。
  - **可以在游戏运行时改变服务**。
  - **定位器依赖外部代码**。这是缺点，相当于使用时必须已经提供服务了。

- **编译时绑定：**

  思路是使用预编译器，如下。

  ```
  class Locator
  {
  public:
    static Audio& getAudio() { return service_; }

  private:
    #if DEBUG
      static DebugAudio service_;
    #else
      static ReleaseAudio service_;
    #endif
  };
  ```

  像这样定位暗示了一些事情。

  - **快速。**编译时完成。
  - **能保证服务是可用的。**编译通过就肯定能用。
  - **无法轻易改变服务。**由于绑定发生在编译时，任何时候想改变都得重新编译并重启游戏。
- **在运行时设置：**
  - **可以更换服务而无需重新编译。**
  - **非程序员也可以改变服务。**
  - **通用的代码库可以支持多种设置。**
  - **复杂。**
  - **加载服务需要时间。**

如果服务不能被定位怎么办？

- **让使用者处理它：**
  - **让使用者决定如何掌控失败。**
  - **使用服务的用户必须处理失败。**冗余代码太多，不推荐。
- **挂起游戏：**例如使用断言，这样保证了不会在正式版本找不到服务。
- **返回空服务：**
  - **使用者不必处理缺失的服务。**
  - **如果服务不可用，游戏扔将继续。**

优化模式

在游戏中，性能优化是一门高深的艺术。**数据局部性**介绍了计算机的存储层次以及如何使用其以获得优势。**脏标识**帮你避开不必要的计算。**对象池**帮你避开不必要的内存分配。**空间分区**加速了虚拟世界和其中元素的空间布局。

数据局部性

意图是：**合理组织数据，充分利用CPU的缓存来加速内存读取。**

确实芯片的速度越来越快，但是内存的速度却没有跟上，所以目前的窘境是**可以更快地处理数据，但是不能更快的获取数据。**

解决这个问题的思路是现代CPU提供了**多级缓存**。将所需数据的附近的数据加载到缓存中，如果下次读取数据时在缓存中，称之为**缓存命中**，反之CPU空转，等待几百个周期直到读取到内存中的数据，这称之为**缓存不命中（cache miss）**。

现代的CPU有**缓存来加速内存读取**。它可以更快地读取最近访问过的内存的毗邻内存。通过**提高内存局部性**来提高性能——保证数据以处理顺序排列在连续内存上。

记住

软件体系结构的特点之一是**抽象**。解耦意味着容易改变。使用接口意味着使用指针或者引用访问对象，使用指针就会造成在内存中跳跃，带来了这章想要避免的缓存不命中。

如果加上**解析**指针的代价，将数据移来移去的代价将小于缓存不命中的代价。

最缓存友好的数据结构就是**数组**。

脏标识模式

意图是：**将工作延期至需要其结果时才去执行，避免不必要的工作。**

节点位置变化后，就将标志位设置dirty为真，这样当需要计算子节点位置时，先访问父节点的标识，如果dirty为真，则先计算父节点的位置。

记住

延迟太久是有代价的，因为通常需要的时候**现在就要**，可能引起卡顿。另一个问题是：如果有东西出错了，可能根本无法弥补，例如文档编辑过程中电脑突然断电。

设计决策

什么时候清空脏标识：

- **当结果被请求时** —— **如果不需要结果，可以完全避免计算。**如果原始数据变化的速度比推导数据获取的速度快得多，效果就会很明显。**如果计算消耗大量时间，这会造成可察觉的卡顿。**这部分工作一般足够快，不会影响体验。
- **在精心设计的检查点处**—— 例如同步点被触发时，或者加载画面或者过场动画之后。
- **在后台处理**—— 通过在第一次启动时启动一个计时器，进行一段时间所有的变化。这样做可以控制工作频率。同样，后台处理也需要支持**异步操作**，这样就需要考虑保持并行修改的安全性了。

脏标识的粒度问题：

如果允许玩家个性化一些节点，那就需要将变化发送到服务器储存。这就涉及到粒度问题了。

- **如果粒度更细** —— 即每个节点都有脏标识，那么只需要处理变化的节点就可以了。
- **如果粒度更粗** —— 假设为一组节点的父节点设置脏标识，那么**最终需要处理没有变化的数据**、**用在储存标识的内存变少**、**固定开销花费的时间更少**。

对象池模式

意图是：**放弃单独地分配和释放对象，从固定的池中重用对象，以提高性能和内存使用率。**

通过游戏开始时分配一大块内存，在结束后释放，可以解决运行时生成对象和内存碎片话问题。

模式

定义一个**池**对象，其包含了一组**可重用对象。**其中每个可重用对象都可查询是否使用中。池初始化时，就创建了若干对象（通常是若干连续分配，这还缓存友好），然后初始化为不在使用中状态。当需要对象时，像池子要一个。

在以下情况中使用对象池：

- 需要频繁创建和销毁对象。
- 对象大小相仿。
- 在堆上进行内存分配十分缓慢或者会导致内存碎片。
- 每个对象都封装了像数据库或者网络连接这样很昂贵又可以重用的资源。

同时只能激活固定数量的对象

某种情况下可能从池子中重用对象会失败，因为都在使用中，这里有几个常用对策：

- **完全阻止这点。增加对象池的大小**，对于比较重要的对象。
- **就不要创建对象了。**例如满屏的粒子系统不差这一个。
- **强制干掉一个已有的对象。**例如当前有多个声音在播放，那么选一个最轻的声音干掉替换。

空闲列表

当查找当前不在使用的例子对象时，有很多种策略，空闲列表这种可以实现常量时间的查找和删除。代码大致如下。

```
class Particle
{
    public:
      Particle* getNext() const {return state_.next;}
      void setNext(Particle* next) {state_.next = next;}

    private:
      int frameLeft_;
      union
      {
          struct
          {
              double x,y;
              double xVel,yVel;
          } live;

          Particle* next;
      } state_;
};

// 粒子池实现如下
ParticlePool::ParticlePool()
{
    firstAvalible_ = &particles_[0];
    for(int i = 0; i < POOL_SIZE; i++)
    {
        particles_[i].setNext(&particles[i+1]);
    }
    particles_[POOL_SIZE-1].setNext(NULL);
}

//创建粒子
```

```
void Particle::create(double x, double y, double xVel, double yVel, int lifeTime)
{
    assert(firstAvailable_ != NULL);
    Particle* newParticle = firstAvailable_;
    firstAvailable_ = newParticle.getNext();

    //初始化粒子
}

//当粒子不使用时，将其放到链表的头部
void ParticlePool::animate()
{
    for..
    {
        if(particles_[i].animate())
        {
            particles_[i].setNext(firstAvailable_);
            firtstAvailable_ = partlcles_[i];
        }
    }
}
```

设计决策

设计对象池时，有以下几个问题需要考虑。

如果对象与池耦合

- **实现更简单。**对象中放置"使用中"状态。
- **可以保证对象只能被对象池创建。**让池对象成为对象类的友类，将对象的构造器设为私有。

```
class Particle
{
  friend class ParticlePool;

private:
  Particle()
  : inUse_(false)
  {}

  bool inUse_;
};

class ParticlePool
{
  Particle pool_[100];
};
```

- **也许可以避免显式存储"使用中"的标识。**很多对象自身就可以证明是否在使用，这样可以省掉对象池中的状态存储的内存。

如果对象与池不耦合

- **可以保存多种类型的对象。**这也就是通用对象池。
- **必须在对象的外部追踪使用中状态。**

```
template<class TObject>
class GenericPool
{
    private:
     static const int POOL_SIZE = 100;

    TObject pool_[POOL_SIZE];
    bool    inUse[POOL_SIZE];
};
```

空间分区

意图是：**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。**

模式是：对于一系列**对象**，每个对象都有**空间上的位置。**将他们存储在根据位置组织对象的**空间数据结构**中，让你**有效查询在某处或某处附近的对象。**对对象位置改变时，**更新空间数据结构**，这样它可以继续查找对象。